INAUGURAL - DISSERTATION
zur
Erlangung der Doktorwürde
der
Naturwissenschaftlich-Mathematischen Gesamtfakultät
der
Rupert - Karls - Universität
Heidelberg

vorgelegt von
M.Sc. Guillermo Aníbal Marcus Martínez
aus Buenos Aires, Argentina

Tag der mündlichen Prüfung: 27.01.2011

Acceleration of Astrophysical Simulations with Special Hardware

Gutachter:	Prof. Dr. Reinhard Männer
Zweiter Gutachter:	Prof. Dr. Ralf Klessen

To my family

Zusammenfassung

In dieser Arbeit werden die raceSPH- und raceGRAV-Beschleunigungs-Bibliotheken vorgestellt, die eine Anbindung astrophysikalischer Simulationen an Spezialhardware ermöglicht.

Die raceSPH-Bibliothek dient der Beschleunigung von 'Smoothed Particle Hydrodynamics' (SPH), einer Methode zur Bestimmung hydrodynamischer Kräfteinteraktionen. Untersucht wurde die Verwendung von Vector-Einheiten in Mikroprozessoren, Field Programmable Gate Arrays (FPGAs) und Grafikkarten (GPUs). In synthetischen Messungen wurden Beschleunigungsfaktoren von 1,2 bis 28 erreicht, in astrophyikalischen Simulationen von 6 bis 19. Für die gesamte Berechnung wurden Beschleunigungsfaktoren von 1,6 bis 2,4 erreicht, die nahe an dem theoretisch erreichbaren Faktor 2,5 liegen.

Die raceGRAV-Bibliothek dient der exakten Berechnung von Gravitationskräften und wurde entworfen, um den GRAPE-Beschleuniger zu ergänzen. Bei direkter Aufsummierung ist die Performance gleich auf mit den Vector-Einheiten der CPU und bei Normierung auf die Anzahl der Pipelines vergleichbar mit GRAPE-6.

Für die Entwicklung dieser Bibliotheken wurde eine Reihe zusätzlicher Module entwickelt, wie bspw. ein PCI-Treiber für aktuelle Linux-Kernel, eine MPRACE-Bibliothek zur Kommunikation mit FPGA-Karten und eine Buffer-Management-Bibliothek, die effiziente Datentransfers ermöglicht.

Abstract

This work presents the raceSPH and raceGRAV accelerator libraries, designed to interface astrophysical simulations with special-purpose hardware. The raceSPH focuses on the acceleration of Smoothed Particle Hydrodynamics (SPH), a method for approximating force interactions in fluid dynamics. Accelerators used range from vectorizing units on the microprocessors to Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs), and speed-ups range from 1.2x to 28x when measured in a synthetic benchmark and from 6x to 19x when used inside astrophysical simulations, for a total wallclock time speed-up of 1.6x to 2.4x, close to the theoretical maximum of 2.5x.

The raceGRAV library computes gravitational force with high accuracy and is designed to complement the GRAPE accelerator. In direct summation tests, it provides performance on par with vectorizing units of the processor and comparable to the GRAPE-6 when normalized against number of pipelines.

For the development of these libraries, a set of supporting modules were developed, including a PCI driver for modern Linux kernel versions, an MPRACE library for the communication with FPGA boards and a buffer management library for the efficient handling of data transfers.

Contents

Introduction

Astrophysicists are in a difficult position. While in other sciences it is possible for the scientists to perform experiments in order to explore concepts and prove or disprove conjectures, astrophysicists do not have the benefit of pocket universes or stars laying in their labs, waiting to perform the next test. They must therefore rely on observations and knowledge, of physics and chemistry and sometimes other fields, to build theories and explain what is observed. But even then, their observations are limited to snapshots of time, as the evolution of most celestial bodies takes place in a time scale that surpasses the life span of any person. Moreover, because of the huge distances between Earth and other stars and galaxies, the time it takes for the light to arrive to Earth has to be considered. Every observation that goes farther from Earth also looks deeper into the past. For these reasons, astrophysicists resort to simulations in order to better understand the processes that lead to the phenomena they observe.

Computers provide an environment where these simulations can be done efficiently and with great flexibility. Their programmability allow the scientists to experiment with different models and conditions and study their evolution and characteristics. The computer becomes their virtual laboratory.

One of such models is the N-Body simulation of gravitational interactions. Modelling the system as a collection of particles with a defined mass, their force interactions can be computed by the law of universal gravitation. In its most basic form, every particle interacts with each other, so the computation of these forces scales with $O(n^2)$. If we consider that galaxies contain between 100 billion and 1 trillion stars (our galaxy is estimated to have at least 200 billion[1]), computing n^2 interactions is on the order of $10^{22} - 10^{24}$, quite a big number.

The computation of gravitational interactions is such an important component that great efforts have been taken to accelerate its calculation. One of such efforts is the GRAPE, a family of processors specifically designed to compute them, providing speedups by about two orders of magnitude compared to regular processors (at the time of its release).

After the computation of gravity is accelerated other forces, like the one produced by the pressure of interstellar gas, consume a significant portion of the remaining computing time. One method to compute these forces is Smoothed Particle Hydrodynamics (SPH), a meshless particle method that approximates the pressure by interacting only with a group of local neighbouring particles. A good portion of the current document will be devoted to the design and integration of accelerators for the computation of

[1] `http://www.seds.org/messier/more/mw.html`

SPH into current simulation programs used by astrophysicists.

For this purpose, different technologies are used for the design of the accelerators, covering from vector units in modern microprocessors to Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs). FPGAs are reconfigurable arrays of logic elements, allowing the creation of hardware designs and architectures by simply programming the device. GPUs are basically graphic cards that have evolved into fully programmable, massively parallel processors.

The requirements of these technologies from the programming point of view are quite dissimilar, with specialized languages and technical constraints to follow on each case. While scientific applications in CPUs are most commonly programmed in Fortran, C or C++, FPGAs are programmed in Verilog or VHDL, while GPUs use CUDA, BrookGPU or more recently, OpenCL. We will show how to isolate these factors from the main application without affecting significantly the resulting performance.

In the following work we will present a couple of libraries, raceSPH and raceGRAV, that respectively abstract the computation of SPH and gravitational forces from the accelerator used. Their applicability is shown by integrating them with actual astrophysical applications. Accelerators presented include CPU optimizations, FPGA accelerators and GPU coprocessors, that provide different degrees of flexibility, speed and accuracy.

In particular, FPGA accelerators require the development of specialized access libraries for low-level communication. Since boards developed in our research department were used, these libraries had to be developed and they will also be covered in detail. More specifically, the consist of a low level driver (the PCI driver), an abstraction layer for IO functions (the MPRACE library), and a transfer optimization library (the Buffer Management library). Together, they form the core for our generic framework for FPGA application development.

The document follows an organization based on chapters, with each chapter covering a particular module. Because most results are quite specific to the developments on a given chapter, they are presented and discussed in the same chapter instead of a separate section of the document. Is the opinion of the author that this organization makes the content more readable and builds on results from previous chapters as the document progresses.

The document is therefore organized in three main parts, covering the background knowledge (Part I), the supporting libraries (Part II) needed to implement the accelerator libraries, and the accelerator libraries themselves (Part III). Conclusions (Chapter 9) and Appendices (Part IV) round up the final sections.

Part I consists on two chapters, with Chapter 1 dedicated to the astrophysical concepts involved in the simulations and Chapter 2 focusing in the technologies involved in the development of high performance accelerators.

Part II is divided in three chapters, each one describing one module at the lowest level of our software stack. Chapter 3 covers the development of the PCI driver that used to interface the hardware with the user level program through the Linux kernel. Chapter 4 provides high-level IO functionality as well as the communication with the hardware Direct Memory Access (DMA) engine. Chapter 5 documents the Buffer Management library, a set of abstractions for the efficient communication between the

target application in an accelerator board and the application in the host.

Part III is composed of two chapters. Chapter 6 describes the acceleration of SPH computations with a variety of different hardware and documents its use with two astrophysical simulations. Chapter 7 follows a similar structure as the previous chapter for the acceleration of gravitational forces.

This work was done in the frame of the GRACE project, a collaborative effort between the Astronomisches Rechen-Institut (ARI) and the Department for Computer Science V of the University of Heidelberg funded by a Volkswagen Foundation (VWF) grant. Being an interdisciplinary project, the fruitful collaboration between astrophysicists and computer scientists allows the combination of our expertises for a common goal.

Part I

Background Knowledge

Chapter 1

Astrophysical Simulations

The following chapter covers the basic concepts on astrophysical simulations as related with the development of hardware accelerators. They are provided as a guide aimed at a reader with a background in Engineering and no special formation on astrophysics. Most sections follow the development by Aarseth in his book *Gravitational N-Body Simulations*[6], with the exception of the SPH section that follows the work of Monaghan[72, 73]. The reader is encouraged to look at those sources for a more in depth treatment.

1.1 Gravity

The best known description of the gravitational force between two bodies is Newton's law of universal gravitation, first described in its *Principia Mathematica*[77]. When stated as a vector equation, it is:

$$\mathbf{F} = -\frac{Gm_1m_2}{r^2}\hat{\mathbf{r}} \tag{1.1}$$

where $\mathbf{F}$ is the resulting force, m_1 and m_2 are the masses of both bodies, $\hat{\mathbf{r}}$ is the unit vector between object 1 and object 2 and G is the gravitational constant. The resulting force is a vector in the same direction as the distance vector with a magnitude directly proportional to the mass and inversely proportional to the square of the distance. Depending on the direction of the unit vector, both forces are of equal magnitude and in opposite directions. When considering a system with more than two bodies, the force is best computed as a field, where the force at any given point is determined as the sum of the contributions of the field of the other bodies, thus

$$\mathbf{F}_i = m_i\ddot{\mathbf{r}}_i = -m_iG\sum_{j=1;j\neq i}^{N}\frac{m_j(\mathbf{r}_i - \mathbf{r}_j)}{|\mathbf{r}_i - \mathbf{r}_j|^3} \tag{1.2}$$

is the force experienced by the body i with mass m_i at position $\mathbf{r}_i$ for a system with N-bodies, each with mass m_j and position $\mathbf{r}_j$. The energy of such system can be written as

$$E = T + U + W = \frac{1}{2}\sum_{i=1}^{N} m_i \mathbf{v}_i^2 - \sum_{i=1}^{N}\sum_{j>i}^{N} \frac{G m_i m_j}{|\mathbf{r}_i - \mathbf{r}_j|} \tag{1.3}$$

with T the total kinetic energy of the system, U the potential energy and W the energies from external sources. Since the system is closed and has no external interactions, $W = 0$. The kinetic energy is the first term of Eq. 1.3, the sum of the kinetic energy of every body, while the potential energy is the second term, the sum of each unique pairwise combination in the system. These set of equations define the basics for direct gravitational interactions in an N-body system, and allow us to present the following three equations which are at the core of the gravitational force accelerators involved in the following chapters

$$\mathbf{a}_i = \sum_j \frac{m_j \mathbf{r}_{ij}}{(r_{ij}^2 + \varepsilon^2)^{3/2}} \tag{1.4}$$

$$\dot{\mathbf{a}}_i = \sum_j m_j \left[\frac{\mathbf{v}_{ij}}{(r_{ij}^2 + \varepsilon^2)^{3/2}} - \frac{3(\mathbf{v}_{ij} \cdot \mathbf{r}_{ij})\mathbf{r}_{ij}}{(r_{ij}^2 + \varepsilon^2)^{5/2}} \right] \tag{1.5}$$

$$\phi_i = \sum_j \frac{m_j}{(r_{ij}^2 + \varepsilon^2)^{1/2}} \tag{1.6}$$

defining the accelerator, its derivative and the potential for a body i based on the other bodies of the system. The only new addition is the factor ε, referred to as the gravitational softening and used to avoid problems with collisions and near collision situations when $r \to 0$. For a more detailed discussion of gravitational softening, the publication by Shirokov[87] is an interesting reading.

The acceleration (1.4) is directly related to Eq. 1.2, and together with its derivative are used in a Taylor series expansion of Eq. 1.2 for the integration of the position in predictor-corrector methods. The potential ϕ_i (Eq. 1.6) is similar to the second term of Eq. 1.3, but instead of computing the potential for the system is the potential over i.

1.2 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) is a meshless particle method first used by Lucy[62] and Gingold & Monaghan[31]. The method approximates the value of a continuous function by sampling it at particle positions and distributing this value over a certain area (or volume), defined by the SPH kernel. Then, the function at any point in space is the sum of the contributions of each particle at that point. By example, when approximating the density by this formulation we get

$$\rho(\mathbf{r}_i) \simeq \langle \rho(\mathbf{r}_i) \rangle = \sum_{j=1}^{N} m_j W(\mathbf{r} - \mathbf{r}_j, h) \tag{1.7}$$

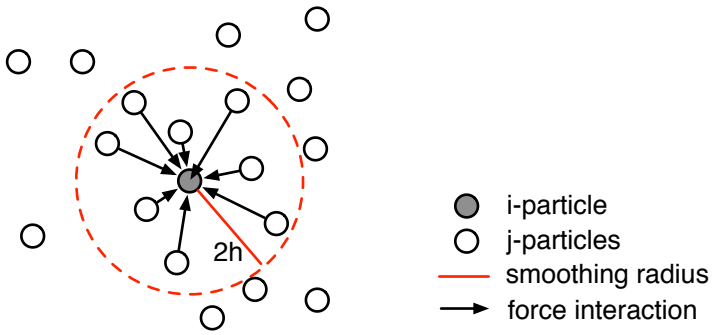


Figure 1.1: SPH approximation of the density at point $\mathbf{r}_i$ for a smoothing length h in two dimensions. Particle i sum the force interactions from $j - particles$ inside the smoothing radius $2h$.

where $\rho(\mathbf{r}_i)$ is the continuous density function, $\langle\rho(\mathbf{r}_i)\rangle$ denotes its SPH approximation and $W(\mathbf{r}, h)$ is the SPH kernel, with $\mathbf{r}_j$ the particle position and h defined as the *smoothing length*. The density is therefore approximated with N points with mass m_j. Graphically, it can be represented as in Fig. 1.1.

The kernel function has several interesting properties. Because the continuous function is approximated by sampling, one can consider the kernel as a function of the family of generalized functions described by the Dirac delta $\delta(x)$, which are defined by the constraints

$$\int W(\mathbf{u}, h)d\mathbf{u} = 1 \tag{1.8}$$

$$\lim_{h\to 0} W(\mathbf{u}, h) \longrightarrow \delta(\mathbf{u})$$

where the last limit defines the kernel as being an approximation. Since W is a part of this family, an infinite number of functions exist that can be used as kernels. A more detailed demonstration can be found in the review by Dalrymple[20].

Another important property of the kernel is that the *smoothing* of the particle is confined by the smoothing length. Therefore, in order to compute the approximation at $\mathbf{r}_i$ only the particles inside a radius $r < kh$ need to be considered and while the summation at Eq. 1.7 is in the range $j = 1...N$, only the interactions inside this radius are non-zero and the summation can be reduced to only n elements, defined as the neighbours of i. Most kernels, including the proposed spline kernels by Monaghan[31, 72, 73] normalize r against h and define the range to be $x < 2$, or $(r < 2h)$.

In order to simulate the system appropriately, the acceleration of the particles is needed. The Euler equation (Eq. 1.9) relates the acceleration to the pressure and the density for an inviscid flow:

$$\frac{d\mathbf{v}}{dt} = -\frac{1}{\rho}\nabla P + g \tag{1.9}$$

with $g = 0$ as no external forces are present. Using the relation

$$\frac{1}{\rho}\nabla P = \nabla\left(\frac{P}{\rho}\right) + \frac{P}{\rho^2}\nabla\rho \tag{1.10}$$

and approximating each term with the SPH formulation as done in Eq. 1.7

$$\nabla\left(\frac{P}{\rho}\right) \simeq \left\langle \nabla\left(\frac{P}{\rho}\right)\right\rangle = \sum_{k=1}^{N} m_k \frac{P_k}{\rho_k^2}\nabla W(\mathbf{r}-\mathbf{r}_k, h) \tag{1.11}$$

$$\nabla\rho \simeq \nabla\langle\rho\rangle = \sum_{k=1}^{N} m_k \nabla W(\mathbf{r}-\mathbf{r}_k, h) \tag{1.12}$$

the acceleration can therefore be written as

$$\frac{d\mathbf{v}_i}{dt} = -\sum_j m_j \left(\frac{P_j}{\rho_j^2} + \frac{P_i}{\rho_i^2}\right)\nabla_i W_{ij} \tag{1.13}$$

$$= -\sum_j m_j \left(\frac{P_j}{\rho_j^2} + \frac{P_i}{\rho_i^2}\right)\nabla_i W(\mathbf{r}_i-\mathbf{r}_j, h) \tag{1.14}$$

with $\nabla_i W_{ij}$ the gradient of the SPH kernel relative to i.

1.2.1 Artificial viscosity

As cited by Monaghan[72], it was identified in simulations without viscosity that when two clouds of gas collide, their flow pass right through them. Therefore, Eq. 1.14 was modified and an additional term Π_{ij} added in order to include an artificial viscosity in the equation, so the acceleration can be written as:

$$\frac{d\mathbf{v}_i}{dt} = -\sum_j m_j \left(\frac{P_j}{\rho_j^2} + \frac{P_i}{\rho_i^2} + \Pi_{ij}\right)\nabla_i W(\mathbf{r}_i-\mathbf{r}_j, h) \tag{1.15}$$

the artificial viscosity term has several formulations, but a common one used here as well as in several other codes[72, 74, 89, 96] is

$$\Pi_{ij} = \begin{cases} \dfrac{-\alpha c_{ij}\mu_{ij} + \beta\mu_{ij}^2}{\rho_{ij}} & \mathbf{v}_{ij}\mathbf{r}_{ij} \leq 0 \\ 0 & \mathbf{v}_{ij}\mathbf{r}_{ij} > 0 \end{cases}$$

$$\mu_{ij} = \frac{h_{ij}\mathbf{v}_{ij}\cdot\mathbf{r}_{ij}}{\mathbf{r}_{ij}^2 + \eta^2 h_{ij}^2} f_{ij}$$

with the smoothing length h, the sound of speed c and the Balsara factor f symmetrized as their mean values

$$h_{ij} = \frac{h_i + h_j}{2} \quad c_{ij} = \frac{c_i + c_j}{2} \quad f_{ij} = \frac{f_i + f_j}{2}$$

1.3 Integration techniques

After computing all the interacting forces present over a particle (gravity and hydrodynamics being two of these forces), the positions are computed by the numerical solution to the differential equations resulting from integrating the acceleration. Generically, if the position $\mathbf{r}$ at time t_n is denoted as $\mathbf{r}_n$ and $t_{n+1} = t_n + \Delta t$ denotes a time interval Δt between t_n and t_{n+1}, then we want to compute $\mathbf{r}_{n+1} = f(t_n, \mathbf{r}_n)$, with f being a generic function.

The easiest of these methods is the Euler method for integration of ordinary differential equations. From the definition of a derivative we have:

$$\frac{dy}{dt} = \dot{y} = \lim_{\Delta t \to 0} \frac{y(t + \Delta t) - y(t)}{\Delta t}$$

and the derivative can be approximated as

$$\dot{y} \approx \frac{y(t + \Delta t) - y(t)}{\Delta t}$$

Finally, separating the term of interest and rewriting according of the definitions of the first paragraph, y_{n+1} is

$$\begin{aligned} y(t + \Delta t) &= \Delta t \dot{y} + y(t) \\ y_{n+1} &= y_n + \Delta t \dot{y}_n \end{aligned}$$

The approximation of the derivative can also be interpreted as the first two terms of the Taylor series expansion of function y around t_n and evaluated at t_{n+1}. That is,

$$\begin{aligned} y(t) &= \sum_{n=0}^{\infty} \frac{y^{(n)}(a)}{n!}(t-a)^n \\ y(t) &\approx y^{(0)}(a) + y^{(1)}(a)(t-a) \\ y(t) &\approx y(a) + \dot{y}(a)(t-a) \\ y(t + \Delta t) &\approx y(t) + \dot{y}(t)(t + \Delta t - t) \\ y_{n+1} &\approx y_n + (\Delta t)\dot{y}_n \end{aligned}$$

This usage of Taylor series expansions allows the construction of several other solutions by using additional terms from the expansion. It also has the advantage that the error introduced by the approximation can be computed as the solution to the maximum bound to the infinite series of the truncated section (the remainder).

An important family of numerical solvers is composed by the predictor-corrector schemes. In these methods, two phases are necessary, with the first phase (prediction) computing an initial estimation which is used on the second phase (correction) to improve over the predicted result.

One such scheme is the Hermite scheme proposed by Makino[64], that according to Aarseth[6] provides several improvements when used in conjunction with special hardware accelerators. In this topic, it is also worth noting the investigations of Nitadori[79] regarding the applicability and possible benefits of higher-order integration schemes.

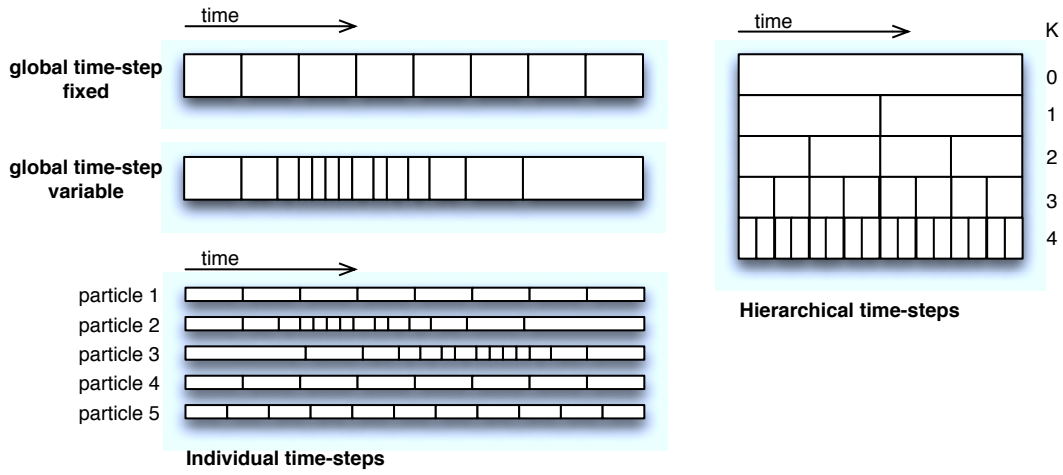


Figure 1.2: Collection of time-step schemes. Global timesteps are shared by all particles. Individual timesteps are unique for each particle and required interpolation to interact with other particles. Hierarchical time-steps assign particles to slots of fixed size and moves them as their time-steps changes. Particles are interpolated to the actual slot being computed.

1.4 Time-steps schemes

While the previous discussion covered how to compute the next timestep, this section focuses on different strategies to determine, when to compute a timestep. The basic assumption from section 1.3 was that all timesteps must be computed on every iteration, in order to produce a sequence of positions $\mathbf{r}_0, \mathbf{r}_1, ...\mathbf{r}_{n-1}, \mathbf{r}_n, \mathbf{r}_{n+1}, ...$ for every particle in the system.

The first and simplest scheme to consider is to use one single, fixed time-step for all particles, or $t_s = \Delta t = k$, with k constant. This is very simple, requiring no special considerations besides the actual value to use. It is however a very wasteful scheme, as it computes every timestep at the maximum time resolution even when some particles are stationary ($\mathbf{v} = 0$) or moving very slowly in comparison to the fastest particle in the system.

Therefore, it can be improved with a single variable, global timestep. In this scheme, the timestep is chosen so it ensures the accurate resolution of the fastest moving particle in the system, by example, so the remainder of the Taylor series expansion is bounded by the desired error function. In other words, it is possible to define a function $t_{s,i}$ that computes the timestep for a particle i, and then set the global timestep $t_s = \min(t_{s,i})$. The main drawback of this scheme is that the resolution is driven by the fastest particle in the system. This might not be an issue for some problems, but astrophysical simulations have particles that span several different time scales, with some area highly dynamic and others fairly stationary. Even a single particle crossing the system at high speed will nullify the advantage of this scheme.

However, it is also possible to assign individual time-steps to every particle. If every particle keeps track of its own variable time-step, it will be computing its position with the best accuracy possible and at the optimum rate, which means no force computations are wasted. But then, the particle time t_i will not match with the individual time of other particles, which is a problem as all particles must be at the same time t to compute forces for that particular time. A solution proposed by Aarseth[6] is to determine the next required global time t based in the smallest $t_{s,i}$, compute the force for particles with comparable time-steps scales, and interpolate the rest. There is an additional cost when computing such interpolations, but it is still less than the cost of computing force interactions.

An alternative to this scheme is to use hierarchical individual time-steps, which defines a hierarchy of possible time-steps instead of allowing them to vary arbitrarily. One possible hierarchy is $t_s = 2^{-k}, x = 0, ..., K$, with $K + 1$ partitions, each half of the previous. Besides being easier to implement, it provides additional advantages e.g. when saving data at regular intervals, as there are time-step multiples when all particles are on the same particle time, so all particles are synchronized and can be dumped easily. Since the possible time-steps are now fixed particles can be grouped by time-step. This leads to the concept of active and inactive particles, where active particles compute the force interactions while inactive particles are interpolated to the next active timestep.

Chapter 2

Hardware Accelerators

With the creation of general purpose Central Processing Units (CPUs) an era of generic processors and generic computing was started. The new CPUs, which were made popular with the Intel 4004[94] design released in 1971 and latter models like the 8008 in 1972 and the 8080 in 1974, shown that a single-chip processor design can be used to perform a great variety of tasks effectively, spawning the introduction of several other similar platforms like the Motorola 6800 also in 1974 and the Zilog Z80 in 1976 and together defining the start of the microprocessor era.

These new, general purpose hardware architectures became the perfect match for the contemporary developments in higher level languages, as the hardware provided the ability to execute a variety of different programs while the higher-level language provided an adequate abstraction of the platform specifics. Languages like BASIC and Pascal became very popular with the commercial computers created around these microprocessors, while FORTRAN, an earlier language until then used mainly in mainframes and supercomputers, remained oriented to scientific applications. Ultimately, as microprocessors became commodity hardware and their performance increases, networks of off-the-shelf microprocessors replaced the specially designed vector units used on the supercomputers as their main processing elements[19].

As the microprocessors advanced technologically and they became faster and more complex, their use broadens, covering also scientific applications previously reserved for supercomputers. For certain specific tasks, it was clear that adding specialized hardware units would speed them up significantly. These hardware units took several forms: from external floating point units (FPUs) in early designs to vector units that can be used for a variety of applications. Other approaches will use custom add on boards, as was the case of graphic cards that function first as simple frame buffers and later as image coprocessors.

In the following sections, we will explore different technologies used to extend the GP-CPUs performance for specific applications, from improvements that have been added into the CPUs themselves to specially designed chips (ASICs). We will focus on technologies useful for scientific applications, particularly those that require floating point operations.

2.1 General Purpose CPUs and Streaming Instructions

All modern CPUs use a variety of techniques to improve performance. Several of these techniques involve circuit design improvements in order to increase the operating frequency, reduce the operating voltage and improve the thermal dissipation of the chip. In this section, we will concentrate in architectural improvements that led to performance gains regardless of the physical characteristics of the device.

CPUs are divided in sections or stages, which each section responsible for a specific task in the processing of an instruction. Therefore, the modern basic 5-stage division is Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM) and Write Back (WB), with more added as the architectures evolved. However, in the first implementations a single clock cycle was used to perform all 5 stages for every instruction, leading to several stages being idle for a good portion of the cycle and only one instruction being finalized or *dispatched* per cycle. Very soon it was realised that by inserting registers between stages, it was possible to increase the operating frequency of the stages at the cost of latency: the stages will execute faster, but it will take 5 cycles to dispatch one instruction.

As a consequence of this separation, it will also be possible to keep all stages busy on every clock, by *pipelining* the stages with different instructions. Assuming non-blocking instructions, if one instruction is pushed into the pipeline per clock cycle, on every clock the instruction will advance to the next stage. Following the same 5-stage example, the first instruction will have a latency of 5 cycles, but subsequently one instruction will be dispatched on each clock, effectively increasing the performance provided by the pipeline by a factor of 5.

One important assumption of the last paragraph was that instructions were non-blocking. Instructions can block each other if a dependency or *hazard* exists between them, by example an instruction that reads the value calculated by the previous instruction and tries to operate over it. Dealing with these dependencies in hardware significantly increases the complexity of the architecture. An in-depth analysis of the subject is out of the scope of this overview, but the de-facto reference for this subject is the work by Hennessy and Patterson[41].

Another improvement used in modern microprocessors is to make them *superscalar*, in order to increase the number of instructions dispatched per clock cycle. The simplest way to make a pipelined design superscalar is to add another pipeline. Instructions must be fetched from the same stream and dispatched in order, to maintain the integrity of the program. In more complex designs, the units are not fixed in a position the pipeline, but can be used as they become idle. A scoreboard is used to keep track of the units being in use and the instructions being processed, in order to dispatch them appropriately. By being able to dispatch more than one instruction per cycle, the utilization of the execution units can be improved and the performance of the increased.

An alternative to scoreboards, where the hardware processor schedule instructions on the fly, is to make the execution units visible on the instruction set and let the compiler make the decisions for instruction scheduling. This leads to architectures like Explicitly Parallel Instruction Computing (EPIC) and Very Long Instruction Word (VLIW), were more than one operation is encoded into a single instruction for parallel

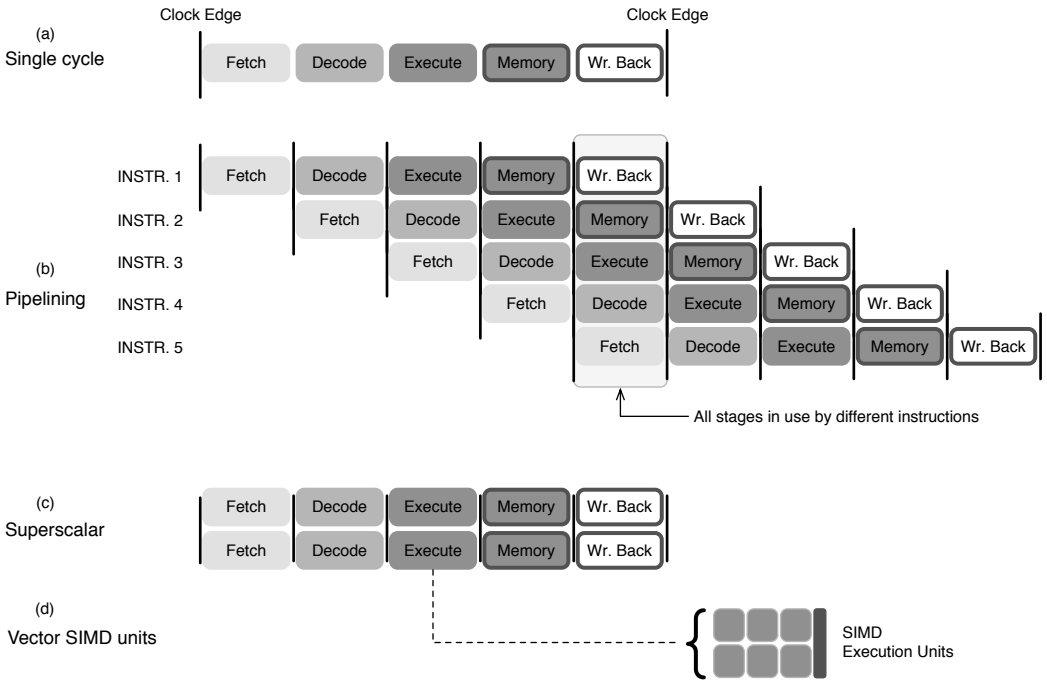


Figure 2.1: CPU techniques overview

execution. Therefore, the decoding of an instruction already provides the information needed for every execution unit as decided by the compiler.

One way to further increase performance is to combine these concepts. By duplicating execution units multiple times, more work can be done per clock cycle. However, more units means longer (wider) instructions, as more bits are needed to decode the function of the additional units, which in turn means more data needs to be fetched from memory on every instruction fetch. But if the units are identical, the Single Instruction, Multiple Data (SIMD) programming model from the vector supercomputers can be used, were each unit performs the same operation over different data. By limiting the operation performed, only one decode unit is needed. Additional data types can be defined that further limit the bits needed for addressing the data used. Because the parallelism is based on the number of execution units used by the SIMD instructions, the size of the vectors used is fixed. This is a critical distinction from vector computers, where the size of the vectors is variable and defined by the value of a register, independent of the number of execution units available.

Independently from the architectural improvements, the area occupied by the processors keep shrinking as a consequence of other technological improvements like better lithography processes that allow smaller transistors to be etched in the silicon die. Therefore, designers were faced with a decision in how to better utilize the additional area. For a time, it was thought that bigger caches will consume the vast majority of the surface available[84], but instead the market took the direction of multiplying the

number of processors per die. In this way, multicore processors were introduced and the multithread programming model already available in modern operating systems made possible the utilization of the additional cores fairly easy. At the present date (end of 2010), six cores per processor are available commercially and eight cores are in the horizon. While the power consumption and silicon processes provide a physical limit on the form of etching resolution and power dissipation, it is nonetheless expected to see commercial manycore processors with tenths or hundreds of processing cores in a single die in the next decade.

2.2 Graphic Processors as Scientific Coprocessors

The first graphic cards were little more than frame buffers which provided a computer with a canvas area that a processor could use to draw information that would be displayed to the user in a display monitor. Over time, these graphics cards gained more functionality to aid the processor in the display of increasingly complex data: first by handling character maps to speed up the display of text and color, later by assisting the host processor in the drawing of 2D display objects needed for graphical user interfaces (GUIs) and emerging mechanical and architectural applications.

As the computational power increased and surpassed the requirements of 2D, 3D graphics became a growing need and manufacturers started to support 3D operations on graphic cards as well. Given the complexity of transforming a 3D scene into a 2D raster image for display, the necessary operations are normally arranged in a graphics pipeline. Current pipelines represent objects as a collection of polygons and are capable of handling object geometry, lightning, reflections and texturing, where each operation is described in general form as a *shader* operation. Two standard pipeline models are OpenGL[53] and Direct3D[71]. While the first graphics processors operated as a fixed pipeline over a stream of polygons, concentrated in processing as many polygons as fast as possible, recent additions to these models allow the use of programmable shaders, which adds a degree of flexibility to an otherwise static process.

Several languages were created to use the capabilities of the programmable shaders, the most important being C for Graphics (Cg) by NVIDIA, OpenGL Shading Language (GLSL) by the OpenGL ARB and High Level Shading Language (HLSL) by Microsoft. While each language has its specific characteristics, all of them support some form of floating point operations. Because they are intended to program shaders and output the result to the display, they do not provide easy communication with the host, specially in the device-to-host direction. However, clever researchers realized that it was possible to read the Frame Buffer Objects (FBOs) from the graphics card memory (i.e. the final image created for display in a monitor, used for off-screen rendering in OpenGL) and establish a processing workflow where data was sent to the card, the graphic processor perform certain computing tasks coded as shader operations, and the results read from the FBO.

Being focused in graphics operations, shader programming languages lack some of the flexibility needed for general purpose programming. Recognizing this interest, ATI released as beta the Close-to-Metal (CTM) language, that allowed the generic programming of the Graphics Processing Unit (GPU) hardware and gave access for the

first time to the underlying architecture of the GPUs. Very soon afterwards, Stanford University released the BrookGPU[1][16] programming language, designed for generic stream processing in multiple platforms. The Brook compiler is capable of transforming Brook code into a multitude of backends, including CTM, OpenCL and DirectX backends.

The use of graphics processors as General Purpose Graphics Processing Units (GPGPUs) became mainstream with the introduction of the G80 architecture from NVIDIA and its Compute Unified Device Architecture (CUDA) programming language. The G80 architecture was one of the first graphic cards to switch from a dedicated graphic pipeline to a fully generic architecture that can cover the requirements of both a graphic pipeline as well as general purpose computing. Its more significant architectural characteristics can be summarized as follow:

- **Hundreds of Processing Elements (PE)**. The first generation of fully programmable GPUs, the G80, contains up to 128 processing units, organized as 16 Multiprocessors (MP) with 8 Processing Elements (PE) each. The Multiprocessors (MPs) are actually SIMD processors with 8 elements-wide vectors, with each PE being an element of the vector. Therefore, the Processing Elements (PEs) are not fully independent, but the software architecture ensures this is not a limitation for most operations. Newer architectures increase the number of PEs up to 240 cores in the G200 and up to 512 cores in the latest GF100 architectures.

- **Higher memory bandwidth**. In order to keep as many PEs as possible occupied, data needs to be fetched from memory at a higher rate than on a normal CPU. While a modern CPU like an Intel i7 has a peak data transfer rate of 25.6 GB/s, the G80 GPU has a raw bandwidth of 86.4 GB/s. Newer architectures increase even further, with a raw bandwidth of 141.7 GB/s for the G200 and 177.4 GB/s for the GF100.

- **Non-cached, Non-coherent memory model**. An important difference with a normal CPU is the memory model on the GPUs. Most modern CPUs have a complex memory hierarchy with several levels of memory cache (L1, L2, L3) between the cores and the main memory of the system. Because every memory cell consumes at least 6 transistors[93], their size occupies a significant area of the chip, regularly over 50%, with researchers predicting in excess of 90% dedicated to caches in future designs. GPUs like the G80 remove this limitation caused by caches by forcing accesses from main memory on every transaction, while providing a small shared area on each MP that can be used as scratchpad by the execution units (the shared memory). Removing the cache memory enable the designers to allocate much more execution units into the chip.

  In addition, a normal CPU dedicates a good deal of hardware to keep coherence between memory operations. It involves ensuring all transactions propagate the correct value needed to perform the required operation transparently, that is, that the program is not aware of these dependencies. By example, when a read-after-write occurs, it has to guarantee that the read value is correct for all subsequent

[1] See `http://www.graphics.stanford.edu/projects/brookgpu/`

read operations after the write is issued. This is a problem that requires extra logic even in a simple pipeline (e.g. a write operation followed by a read operation, while the write has not been dispatched yet). The task is further complicated when taking into account the memory hierarchy (L1, L2, L3 caches and main memory) and multi-threading programs with multiple CPUs and cores present, which requires a specialized protocol to synchronize the data state among all the hardware units and cores. All these requires a significant amount of hardware design that increases the complexity, consumes chip area and ultimately reduces the operating frequency of the CPU.

On the other hand, GPUs relax memory coherence in order to reduce the complexity of the cores. Therefore, it is not guaranteed that a read after a write will return the correct value, when a read is performed by a different PE (coherence is maintained on each PE and its local data). The memory hierarchy is different on GPUs, with cached and non-cached regions, as well as shared memory areas where coherence can be ensured by explicit memory fences. These shared memory areas are accessible to all threads executing in the same MP. It is therefore the responsibility of the compiler (or the programmer) to maintain the coherence of the data being processed, instead of being guaranteed by the hardware. The new GF100 architecture adds additional caches as part of the shared memory area to relax some of these memory restrictions, as well as the capability to do system-wide (across MPs) memory fences, giving the option to synchronize memory accesses over the whole GPU.

- **Thread execution manager in hardware**. Another important difference with most modern CPUs is the handling of concurrent threads. A normal CPU is capable of handling one or two threads per core (Intel calls this feature *hyperthreading*), with the notable exception of Sun UltraSPARC T1 and T2 processors, which support up to 32 concurrent threads. The scheduling of these threads is usually handled by the operating system, which maps them as virtual cores. On the other hand, a G80 GPU supports a high number number of concurrent threads, 768 threads per MP or up to 12,288 parallel threads in-flight (resident) in a single GPU. The assignment of threads is done as a combination of hardware and software. Part of the scheduling is done in software by grouping the threads into blocks, with each block being assigned to a MP. Depending on the resources available, up to 8 blocks can be assigned to a single MP. The partition in blocks is done by the programmer at design time, and will be described in detail in the next section, but their assignment to available MPs is done at runtime. Threads inside a MP are grouped in fixed segments called *warps*, and executed in SIMD fashion by the MP and its PEs. With only 128 PE available, many of these threads are in practice waiting to be scheduled, but the thread manager uses the delays between warp executions to hide long latency operations and greatly increase the pipeline efficiency of the MPs. Also, the use of warps to handle the scheduling makes the cost of context switching virtually zero.

Since architectural differences are bound to occur as new versions of GPUs are released, the CUDA platform defines Compute Capabilitys (CCs) in order to group

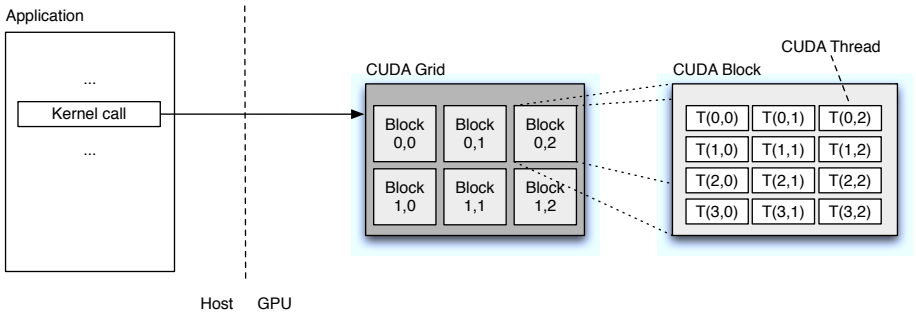


Figure 2.2: CUDA Thread hierarchical organization

different devices according to the functionality available and made the programs aware of them, so optimizations can be made on each case.

As it was mentioned earlier, the success of the GPUs as mainstream coprocessors was a combination of the new architecture and the introduction of the CUDA programming language. A variant of the C programming language, the most important feature of the language is its massively parallel programming model based on threads. With a hierarchical organization, thousands or even millions of threads are supported within a single program.

CUDA follows a Single Instruction, Multiple Thread (SIMT) paradigm, where each thread executes the same code, called a *kernel*. As a coprocessor, kernels must be invoked (*launched*) by the application running on the host to start the execution on the GPU. Threads are organized hierarchically in a generic grid of blocks, where each block is a collection of threads (see Fig. 2.2). Every block in the grid can execute independently of each other, and no communication is allowed between them. Blocks are assigned for execution to a single MP, and they remain resident in the MP until all threads in the block finish execution. Provided that enough resources are available, multiple blocks can be assigned to one MP, but they remain isolated of each other. Threads inside a block are executed concurrently in SIMT fashion, with one thread being processed by one PE. Every thread in the system is aware of its position inside the grid and the block, by means of predefined variables provided by the runtime environment.

One important consequence of the independence of each block is that the platform becomes scalable. The effect is similar to dispatching units of work to workers, where idle MPs consume blocks, and new blocks are assigned to MPs as they become available. This is repeated until the grid is exhausted and no more work is pending. From the programming point of view, programs are mostly unaware of the number of MPs present, and this is why GPU programmers prefer to have many blocks with many threads, so programs can be executed mostly independent from the number of MPs, distributing work both among PEs inside an MP as well as among as many MPs as possible. This leads to another advantage from a business point of view for NVIDIA, as they can customize the hardware offerings to different market segments by just adjusting the number of MPs and the amount of memory included. In this way, the GPU

	Size	Latency	Location	Cached	Access	Scope
Register	8K regs	1	On-chip	no	R/W	per thread
Local	global	400/800	Off-chip	no[a]	R/W	per thread
Shared	16 KB	2[b]	On-chip	no	R/W	per block
Global	≤1.5 GB	400/800	Off-chip	no	R/W	system
Constant	64 KB	1/800	Off-chip	yes	read only	system
Texture	global	1/800	Off-chip	yes	read only	system

a Local memory accesses are cached for devices with Compute Capability ≥2.x
b If there is no bank conflicts. Otherwise accesses are serialized and the latency increases

Table 2.1: Memory Regions available in a GPU. Sizes and latencies are representative for a G80 architecture. System scope represents all threads in the GPU plus the host system.

becomes modular and the MPs are the basic building block.

CUDA also makes a distinct separation of the memory regions available within a GPU, which are Register, Global, Shared, Local, Constant and Texture memory (see Tab. 2.1 for a summary). Registers are used by automatic variables and regular variables per thread, but the total amount of registers available inside a MP is fixed. Therefore, the number of registers which can be used by a thread is a function of the number of threads in the MP, which is also dependent of the number of threads per block and the number of blocks assigned to the MP. On the extreme case where 768 threads are assigned to the MP (being one block of 768 threads[2], 3 blocks of 256, or 8 blocks of 96), every thread has only $\lfloor 8192/768 \rfloor = \lfloor 10.\bar{6} \rfloor$, or 10 registers available per thread.

Global memory is the main GPU memory, with up to 1.5 GB for the G80 architecture and available to all threads in the MPs as well as the host system. Global memory is not cached, so any access by a thread implies a latency of 400-800 cycles before the data is available. However, the memory controller is capable under certain conditions to process multiple requests from multiple threads simultaneously. These conditions are in general described as *coalesced* accesses, and they group a set of memory access patterns where a set of active threads access a contiguous set of addresses. One simple example is when each thread in a block reads an element in an array, and the index of the element being read corresponds to the thread id of the requesting thread (i.e. `x=array[thread_id]` on every thread). Depending on the Compute Capability of the GPU, other access patterns support shifting or interleaving of the addresses as variations of this basic pattern. Non-coalesced accesses require additional memory transactions that severely affect the memory performance of the program. When properly combining the latency of the coalesced memory accesses with the number of threads resident, it is possible to completely hide the memory latency and fully utilize both the memory bandwidth and PEs available.

[2]Not really possible on the G80 architecture, as a block is limited to 512 threads for devices with Compute Capability <2.0

While the CUDA optimizing compiler is quite aggressive in register reuse, it is still possible that not enough registers are available for the execution of a single block, causing *register spilling* into local memory. Local memory can therefore be defined as using global memory for variables that otherwise do not fit or would consume too many resources in the register space. This applies to requiring too many registers in a block, local (per thread) arrays and certain transcendental functions that require local memory. The use of local memory is undesirable, as it has all the drawbacks of accessing global memory and none of the advantages of registers, but is a fallback solution for an otherwise not-executable code.

Shared memory is a scratchpad area, shared among all threads within a block. Therefore, shared memory can be used by the threads inside a block to share data and coordinate work among them. Accesses to shared memory have a latency similar to registers, but they are further constrained by bank conflicts, so caution is advised when designing multithread access to shared memory. When a bank conflict occurs (i.e. two or more threads try to access different addresses in the same bank), the access to the bank is serialized and additional latency is introduced.

Constant memory is a small area of global memory that is optimized for access to read-only data, like system parameters and computing constants, which can only be written by the host. In contrast with global memory, the constant memory is cached by a small, 8 KB size cache on every MP, providing a latency comparable to register access for a cache hit and a worst case of hundreds of cycles for a miss. In practice, the worst case is seen rarely, as constants can be prefetched by the MP and therefore reduce significantly the latency involved.

Texture memory is another redefinition of the global memory. As it name implies, its main purpose is to support texture accesses for the graphics pipeline, so its optimized for 2D and 3D data locality but it has certain features that can be used by generic programs. When part of the global memory is assigned to a texture buffer (a process called *binding*), it is also marked as read-only and cached by the MPs, so it is an alternative for caching data access of big areas from global memory that do not need to be written[3]. In addition, textures make use of the texture units present on each MP, which besides direct data access can perform data conversion (8-bit and 16-bit integers to 32-bit floating point numbers) as well as linear, bilinear and trilinear interpolations with dedicated hardware, independent from the PEs. Because the texture binding is done by the host and the memory is set to read-only, loading a texture in memory can be done only by the host program. Writing to the global memory associated with a texture by the device has undefined results, as it collides with the operation of the texture cache.

During the previous discussion on the memory spaces of the GPU, several constraints and limitations have been shown that affect the performance of a program. Some, like coalesced memory accesses, are runtime limitations; but most of them are related to features like number of threads per block and grid dimensions, which are set at compile time by the programmer, or number of registers per thread, which is a by-product of the compiler after processing the source code. With relatively few of these

[3]In devices with Compute Capability below 2.0. Newer devices have a cache for global memory access, and it is preferable to the texture binding for most situations.

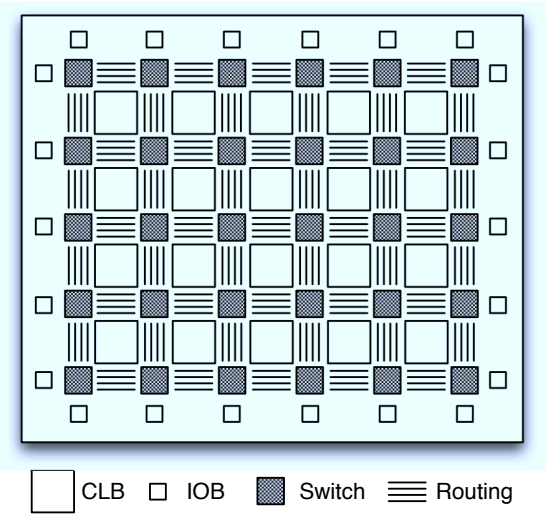


Figure 2.3: An uniform FPGA array with a matrix of 4x5 CLBs and 22 IOBs.

parameters and the NVIDIA *Occupancy Calculator*, a tool which includes the limits for all Compute Capability devices, it is possible to compute easily the percentage of occupancy of the MPs as well as to identify the most restrictive parameters, together with the scaling and variance of them as the number of threads is varied. For a more detailed analysis of the code, the CUDA profiler provides access to the performance counters in the GPU, analysing the code as its executed in debug mode.

For a more in depth study of the CUDA programming platform, the more relevant references are the CUDA C Programming Guide[82] and the book by Kirk and Hwu, *Programming Massively Parallel Processors*[54]. Valuable information on optimization techniques for the platform is also provided by the CUDA Reference Manual[83] and the CUDA Best Practices Guide[81].

2.3 Field Programmable Gate Arrays

Most programmable logic was developed in order to reduce component count in circuit boards and increase density. The most generic device of this family is a simple memory device, where the address lines are the inputs and the data lines the outputs. With m inputs and n outputs, it can map n functions with m variables of arbitrary complexity.

Signetics, once a mayor manufacturer of integrated circuits[104], first introduced a programmable array in 1975 as part of their efforts to further increase density on circuits boards. These arrays where the first to include logic building elements: an array of AND and an array of OR gates that could be interconnected to create logical functions. Monolithic Memories (MMI) created the Programmable Array Logic (PAL) in 1978, introducing the concept of a macrocell that implements a function as a *sum-of-products*.

Xilinx introduced the first commercial Field Programmable Gate Array (FPGA)

in 1985[4]. FPGAs consist mostly of an array of Configurable Logic Blocks (CLBs), IO Blocks (IOBs), and a routing interconnect network, with a generic arrangement depicted in Fig. 2.3. IOBs provide connectivity between the IO pads and elements in the array via the interconnect, while CLBs provide the logic elements. Switching elements provides the configurability, routing signals between elements by switching connections among static routes. The function performed by a CLB, IOB settings and the chosen routes are all configurable by memory locations, so the program of the FPGA can be uploaded or changed by reloading the associated configuration memory. Routing in a FPGA design is static, which means that once a configuration is loaded, the routing does not change. Even new developments like the SpaceTime technology by Tabula[5] use static routing, as this technology is more related to rapid reconfiguration, not dynamic signal routing.

Modern CLBs contain multiple slices that can be combined to create more complex functions. Fig. 2.4 shows the CLB structure from a Xilinx Virtex II FPGA. In this architecture, every CLB contains 4 slices, with each slice containing 2 function generators, 2 Flip-Flops (FFs) and some additional logic, mostly to implement multiplexers, dedicated carry lines and to support specific operating modes of the FF. The function generators are 4-inputs, 1-bit output that similarly to the ROM example, can be used as a small 16x1 RAM, a 16-level shift register, or as a Look-Up Table (LUT).

FPGA manufactures need to balance the amount and distribution of routing and logic elements very carefully, as too few routing elements may lead to all available routes being exhausted quickly, requiring the reuse of logic blocks as pass-through shortcuts. Similarly, too many routing elements may waste chip area that could otherwise be used by logic.

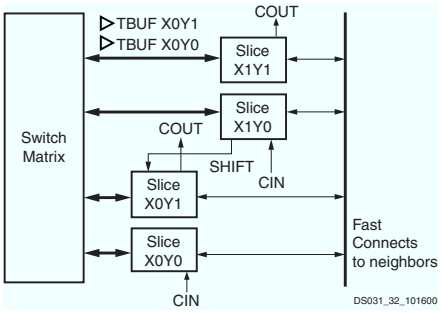
Given their complexity and the proprietary nature of the internal designs, FPGA development is always done in Hardware Description Languages (HDLs), with vendor independent and vendor-specific tools that follow a workflow to generate a programming image (a *bitfile*) which can be uploaded to the configuration memory of the device. The two most common HDL languages are Verilog and VHDL, but many more exist. While these languages are high level with many abstract constructs, they can also be used to describe electronic designs in great detail, therefore used to create behavioural, functional and structural models of components. Abstract constructions are normally used for simulation, very representative of the task, but usually not possible to convert into actual circuits. Special tools are used to convert functional or structural representations into low-level equivalents. Several attempts have been done to use traditional languages like C[61, 56] or Java[11] to describe hardware, with limited success.

The tools in the workflow transform higher-level representations (like HDLs) into simpler representations that can finally be used to generate a wiring diagram between standard building blocks, not unlike many other Electronic Design Automation (EDA) tools for hardware synthesis. The basic steps of this process can be summarized as follows:

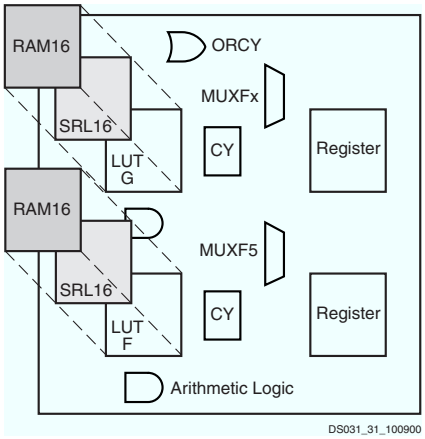
- **Synthesis**. Is the process of converting and optimizing the high level description

[4]XC2064, `http://www.xilinx.com/company/history.htm`

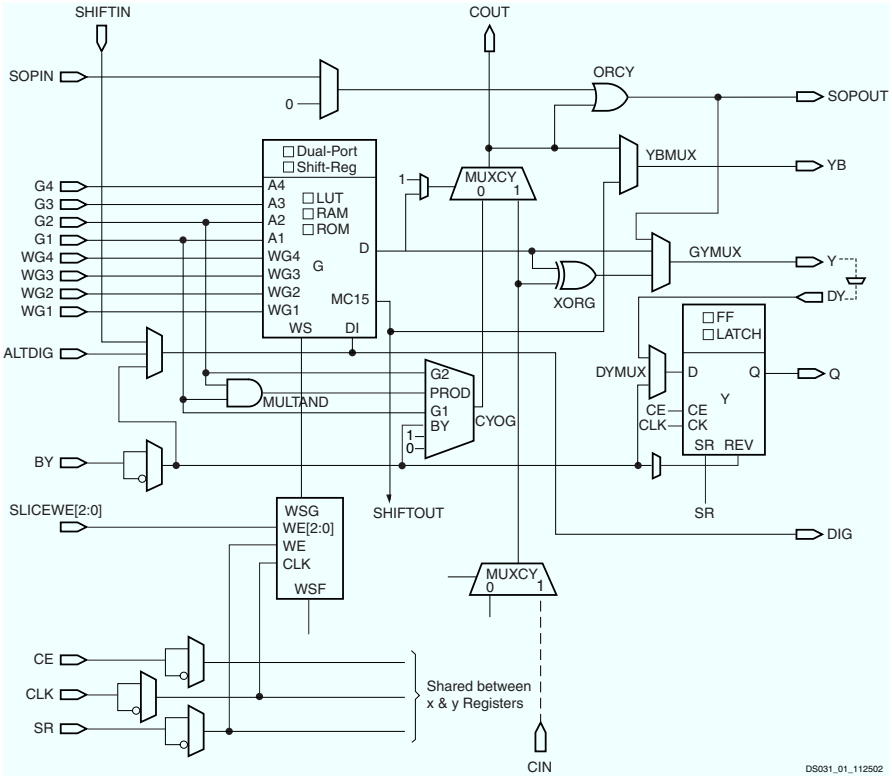
[5]See `http://www.tabula.com/technology/technology.php` for more info



A CLB with multiple slices



A Slice with multiple configuration options



Equivalent schematic of the top section of one slice

Figure 2.4: Virtex-II CLB Element, from [98]

in HDL in order to generate a LUT-level netlist[49]. Its output is typically a logic circuit built with common blocks known to the vendor-specific tools. Several third-party tools exist, that provide varying levels of flexibility, measured on both their ability to recognize and transform abstract constructions into logic circuits, and their capacity to optimize these circuits to meet the required design constraints. The process is similar to an optimizing compiler, which transforms source code into an optimized object code for a target platform.

- **Translation**. Specific to the Xilinx implementation flow, this step converts the LUT-level netlist, generic UNISIM primitives (common to many Xilinx device families) into SIMPRIM primitives, specific to a device, and insert additional timing constraints for the design, together with any black-box definition[49]. Other vendors implement this step into their mapping tool.

- **Mapping**. The Map tool assigns primitive elements to actual element types in the platform device. Logic Functions are therefore assigned to LUTs, registers are assigned to FFs, and so on. At this point, the tool can verify device occupation and add switching time information for the next stage.

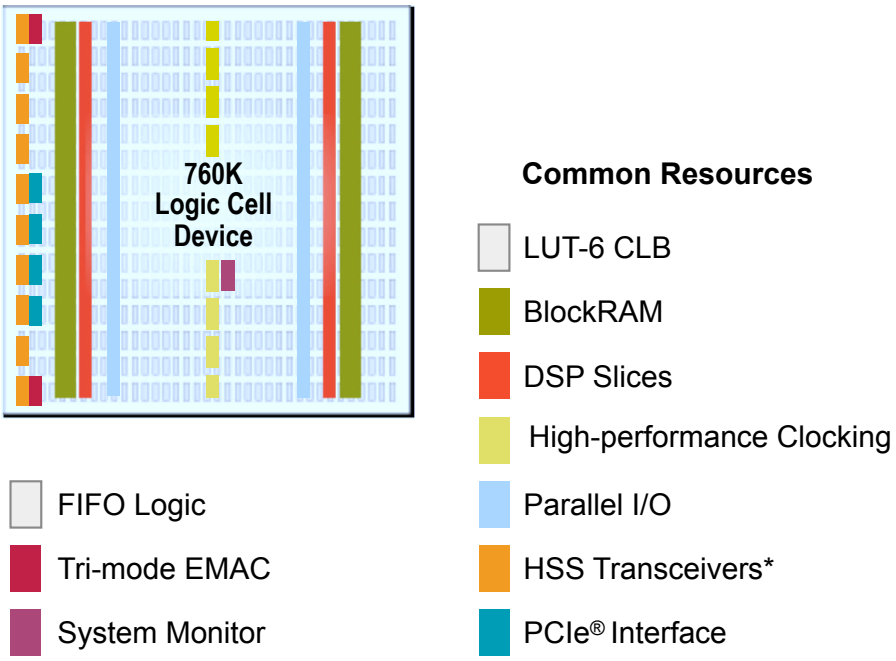
- **Placing and Routing**. While two separate tasks, placing and routing are often performed together. After the design has being mapped, the components must be allocated (*placed*) in the FPGA array. When adding the interconnects, the routes available and the distance between components add delay to the signals, that can (and very often does) fail the required timing. Placing and Routing are difficult, NP-hard problems[24], that is both computationally intensive and time consuming. Many implementations perform iterative processes in order to determine the best implementation (but not necessarily optimal).

- **Bitfile Generation**. Convert the generated implementation into a binary representation suitable for upload to the device configuration memory.

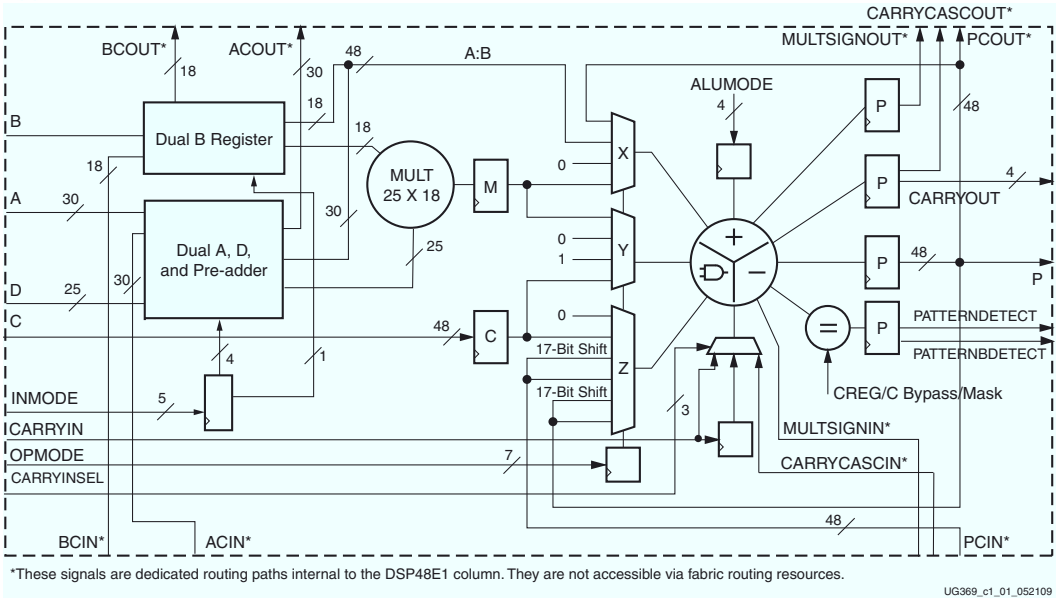
Specialized Elements

As the number of CLBs increased, users started to do more complex designs. From replacing logic elements, FPGAs turned to be ideal for hardware prototyping and to process data streams. As integer adders, multipliers and buffers were more often required, the amount of CLBs needed to implement these design blocks (*cores*) rose exponentially. For this reason, FPGA manufacturers have departed from uniform arrays and added specialized elements to the architecture fabric. Because these blocks are implemented directly in hardware, they are both faster and more compact than their CLB counterparts, allowing a better utilization of the chip area. Four areas summarize the specialized units present in modern FPGAs, with examples depicted in Fig. 2.5 for the Virtex-6 family:

- **Clock Manipulation**. Because routing networks can cause strong variations in skew and signal distances keep increasing as the density increases, newer FPGAs required to implement dedicated clock networks for low-skew signal distribution,



Virtex-6 Architecture Overview



Virtex-6 DSP48E Slice

Figure 2.5: Virtex-6 Overview, from [100]

as well as clock subdomains to localize clock signals to certain areas of the chip. Additionally, dedicated hardware has been added for signal synchronization like Phase-Locked Loops (PLLs) and Digital Delay Lines (DLLs), as well as clock dividers and multipliers.

- **Memories**. Adding dedicated memories, like the BlockRAM in the Xilinx platform, frees many registers from the CLB logic. Memories are primarly used to create buffers, FIFOs and big LUTs, all very common design cores.

- **Arithmetic**. The first arithmetic addition was the dedicated carry chain to the CLB logic, that serves the specific task to speed-up adders/substractors, a very important addition based in the ubiquitous presence of counters and incrementers on any design. The next significant addition was the inclusion of hardware multipliers. While targeted primarly to Digital Signal Processor (DSP) designs, multipliers are useful in a range of applications and critical for the implementation of floating point multipliers units. The DSP support was extended with the inclusion of XtremeDSP slices with the Virtex-4 family and DSP48 units with the Virtex-5, supporting pipelined multipliers and a hardware accumulator and making the implementation of multiply-and-accumulate operations very easy. In order to better utilize the density of newer devices without requiring substantially more routing resources, Virtex-5 devices also introduced 6-input LUTs.

- **Communication**. In communication units, three additions are noteworthy: the management of DDR and differential signals by the IOBs, the inclusion of serial transceivers for high-speed serial communication, and the inclusion in hardware of PCI Express (PCIe) cores in select devices. While the first two are required to support current communication standards, the PCIe core is targeted at interconnection with computing devices, being as PCIe coprocessors in PCs or in embedded systems.

The biggest drawback of the specialized units is that they loose the flexibility of the FPGA. If a design does not use the particular functionality of a unit, it becomes wasted space on the chip that cannot be reused by the design in any other way. For this reason FPGA families became fragmented, offering different mixtures of units to better match the requirements of particular application segments.

Fortunately, the improvements provided by the specialized are very appropriate for the use of FPGAs as coprocessors. The principal limitation of ancient devices for their use in scientific applications was that they required Floating Point (FP) operations and the devices did not have enough CLBs to implement them. As the number of CLBs increased, Floating Point Units (FPUs) became feasible. The addition of hardware multipliers greatly improved both the performance and number of possible FP multipliers. Several libraries were developed that implement a set of FP operations [58][68], some of them parametrizable to obtain the best balance between area, speed and accuracy, depending on the needs of the application. As the number of operators implemented reached hundreds, automatic tools were created for the assembly of complex pipelines from formula descriptions. Much more limited than actual compilers, tools like the Pipeline Generator by Lienhart[60] and the PGPG by Hamada[35] provided a huge

improvement in development time. Other improvements beneficial for the coprocessor design are the PCIe cores for interconnection to the host and the Double Data Rate (DDR) support for faster memory controllers. All combined with the flexibility of re-programmability, modern FPGAs provide a hardware solution to accelerate complex applications.

2.4 Application Specific Integrated Circuits

Application Specific Integrated Circuits (ASICs) are at the other end of the spectrum from CPUs. While CPUs aim at good performance with a wide range of applications, ASICs focus in performing a single task, or a small range of tasks, extremely well. Because they are custom-designed to perform this job, they tend to be very fast and consume less power, as well as being very efficient at it, both in terms of sustained vs. peak performance and performance per watt. For high volume applications, ASIC are also the cheapest solution. The main drawback is its lack of flexibility, so they are best suited to perform tasks that are not likely to be modified. The next important drawback is the cost: design and production costs are very high, and only compensated by a high volume production. In the case of scientific research, because the target is usually a single specialized application that requires a limited number of pieces, the manufacturing costs are very high for small quantities.

There are several ways to achieve the ASIC level of performance. The obvious route is to design a custom chip from scratch, but this is a very long process. Several companies like Mentor Graphics, Cadence and IBM provide Intellectual Property (IP) libraries and EDA tools for the assembly of preoptimized building blocks that can speed up the development and testing phases considerably. Other companies like ARM specialize in custom CPU cores (the ARM architecture) that are available as IP cores and can be extended and modified to suit the needs of the design. In practice, most ASIC designs are the assembly of standard-cell blocks from an IP library, which are already optimized for a specific manufacturing process.

An alternative to the traditional ASIC design flow is the use of a compromise solution, Structured Application Specific Integrated Circuits (s-ASICs). Instead of optimizing every element of the design on the lowest level, s-ASICs use an array of fixed elements and provide an standard interconnect, not unlike an FPGA architecture. Positioned to be an alternative between FPGAs and fully custom ASICs, s-ASICs aim at lower design and production costs while providing better performance, better power efficiency and a high degree of compatibility with FPGA designs. By removing the reconfigurability of the interconnect, several designs elements can be optimized: interconnect networks can be replaced by wires, read-only memories can be replaced vias in the manufacturing process, and so on. Several leading companies in the FPGA sector offer s-ASIC alternatives, like *HardCopy*[1] from Altera and *EasyPath*[2] from Xilinx that provide a migration path from FPGAs into medium volume production.

One example of a successful ASIC that got a lot of attention by the media was Deep Blue by IBM. Developed by Feng-Hsiung Hsu and his team as the next generation from

Deep Thought[6], an FPGA coprocessor for computing chess moves using brute force evaluation, Deep Blue 1 and 2 were improvements in speed and capabilities provided by both the chip and many refinements in the associated software[45]. Mostly regarded as a public-relations stunt by IBM Research[51, 86], it exemplifies well the target for accelerators: computing intensive, massive parallel, and focused in small number of tasks.

Since our scientific interest is in accelerating astrophysical applications, this discussion would not be complete without an overview of the GRAPE hardware. The GRAvity PipE (GRAPE) is an ongoing project led by Jun Makino that uses custom hardware to accelerate the computation of Newtonian forces in N-body systems. Because forces are computed by a very well known formula, the complexity of the computation was high – $O(n^2)$ for direct summation – plus they easily consumed over 90% of the computational time as simulation size increased, these forces where perfect candidates for acceleration with ASICs. The GRAPE-1, their first prototype system from 1989 was capable of a peak performance of 240 Mflops, while GRAPE-4 systems released on 1995 was capable of 1.1 Tflops, being the fastest supercomputer in the world between 1995-1997[46]. Subsequently, the GRAPE-5 and GRAPE-6 designs received 3 Gordon Bell prizes in 1999, 2000 and 2001 for their contributions to high performance computing.

The detailed architecture of the GRAPE-6 chip is described by Makino et al. at PASJ[67], but it is basically an upgrade to the GRAPE-4 in order to improve the performance of the pipelines. Every GRAPE-6 chip contains 6 pipelines to compute force-interaction, 1 pipeline for position and velocity prediction, and a neighbour list unit. The 6 pipelines are further organized as 48 virtual pipelines[64], reducing memory requirements per pipeline. The GRAPE chips provide a modular architecture that is fully exploited in the GRAPE-6, with 16 chips interconnected in big clusters boards while each GRAPE-6A card[28], a smaller PCI version suitable for direct use with PC hosts, contains 4 chips.

According to Makino[67], a critical point in the design of the GRAPE-6 chips is to reduce the amount of communication between the host and the GRAPEs, and to achieve this it is best to send only the particles that have changed in the current time-step. Therefore, the GRAPE-6 has a predictor unit that is able to interpolate the position of the inactive particles.

The GRAPE-6 also has a dedicated unit to compute neighbour lists. While limited to 256 neighbours per list, the units are very useful to a variety of applications. Not only are they useful for neighbour list based gravity schemes [6, 9] but they can be reused to assist in the calculation of SPH [75, 92]. Unfortunately, the relatively slow communication link limits the performance for this use.

The GRAPE family of accelerators is the story of a successful ASIC for scientific computing. By supporting only a very small set of functions, it has been capable to produce enormous gains in performance, enabling scientists worldwide to advance research in ways that otherwise would have been impossible. The latest generation in the family, the GRAPE-DR[65], provides an ASIC with SIMD programmable cores tailored for the computation of cumulative forces, with an architecture not unlike that

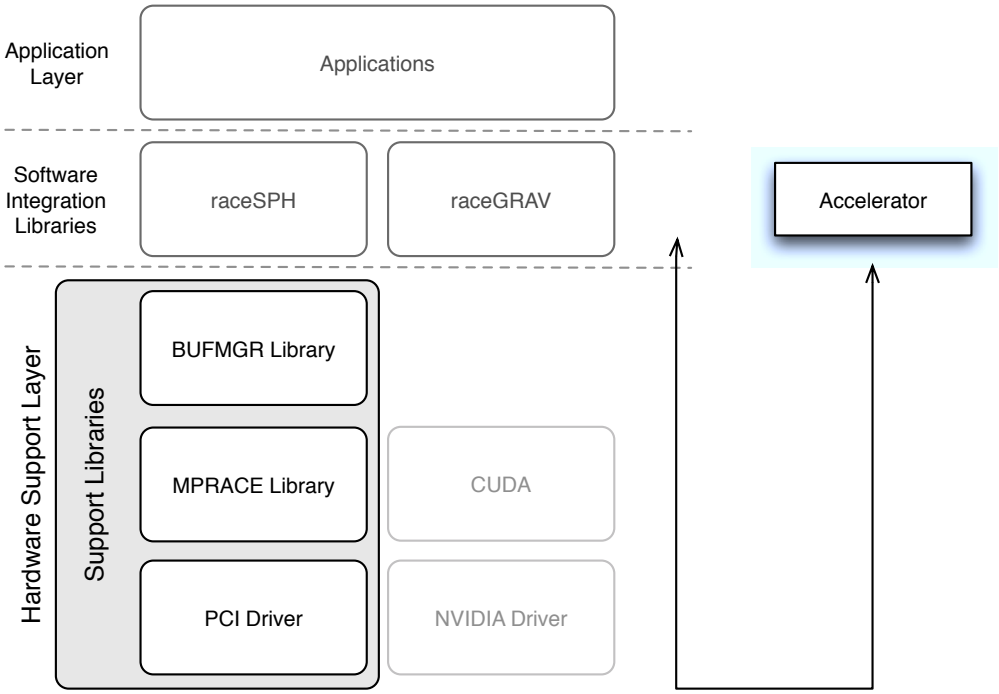
[6] named after the fantastic computer from Douglas Adams' novels[7]

Figure 2.6: The GRAPE-6A board

of modern GPUs.

Part II

Supporting Libraries



The diagram shows a simplified view of the software stack developed. For an application and an accelerator to communicate, they must go over some route through the software stack and into the hardware. Every layer defines certain level of operations, with the application being on top and using some of the higher level functionality exported by the software integration layer, responsible of abstracting the accelerators as computing cores. The hardware support layer is responsible for the communication and functionality needed to access the accelerators in a general level, and it is needed by any accelerator used. Therefore, GRAPE cards need special drivers and libraries, as do the FPGA and GPU cards, that are normally provided together with the boards. In this diagram, *CUDA* and the *NVIDIA driver* blocks represent proprietary components provided by NVIDIA for the use of their graphic cards and used by the integration layer when accessing GPU accelerators.

The following three chapters will describe the development of the components marked as supporting libraries in the hardware support layer, needed in order to access the FPGA boards. They abstracting functionality in the cards and aim at sustaining the available transfer rates of the interconnect between the FPGA and the host, typically a PCIe link or a PCI bus.

Chapter 3

The PCI Driver

In modern operating systems, the function of a device driver is to provide the logic needed to communicate the devices connected to the system with the applications using them. Like any other device in the system, our FPGA boards need a driver to handle the communication with the operating system. The PCI driver fits on our software stack at the lowest level, providing the OS dependent functions that makes most of the remaining libraries OS independent.

Our previous PCI driver, used by boards like the μEnable and the MPRACE-1, was developed under Linux 2.4. When the Linux kernel 2.6 finally arrived, patches were made to continue usage and development. However, the kernel 2.6 brings significant changes in device administration, memory handling and DMA configuration, as well as new features like SysFS. It was clear that a new driver was needed to support the requirements of the new MPRACE-2, as well as the ABB and future boards.

The driver has to support the needs of several ongoing projects in our research groups, namely the ROBIN boards for the CERN, the ABB board for the CBM, and the MPRACE-2 for the GRACE project. Each one adds a certain range of restrictions: the CERN platform runs mainly in Scientific Linux CERN v4 (SLC4), a variant of Red Hat v4 running kernel 2.6.9. The Titan cluster of the GRACE project, where the MPRACE-2 has to be installed, runs a distribution with kernel 2.6.13; while the CBM project required at least kernel 2.6.18 to properly support the runtime extensions and infiniband cards used.

In addition, the kernel development switched from a version-based release schedule to a features-based, continuous development model: instead of limiting major version numbers for feature addition and minor numbers for fixes, each version release might contain new features. This has the advantage that the kernel evolves quickly, but adds significant maintenance cost to out-of-tree code –each release might break existing code.

Therefore, the driver has to be general enough to support a range of platforms and distributions: from very ancient kernel releases up to the latest version, while providing the functionality required to communicate with our FPGA boards.

The driver needs to support multiple, generic PCI devices in 32- and 64-bits systems, and provides mapping of PCI(e) base address registers (BARs), access to the device configuration space, memory management for kernel- and user- space memory, hot-plug capability, dynamic chardev allocation, SysFS accessibility and a C / C++ user

interface, as well as a compatibility layer for applications using our old driver.

3.1 Architectural Overview

The Linux Architecture defines several memory spaces in order to create protective barriers. The most generic setup has a device memory space, a kernel memory space, and user memory space. These spaces map to the different buses (the PCI bus in our case) and to the physical memory address space. Both kernel and user memory are mapped to separate areas of physical memory, see Fig. 3.1. The kernel space is on a fixed area, while the memory manager of the kernel takes care of dynamically mapping the user space assigned to each process to the available physical memory (or eventually, the swap file). The user memory space is virtual, so each user process has an independent space. Memory pages are assigned on demand. The IO-MMU takes care of mapping addresses between the device memory space and the physical memory space, and to add a bounce buffer if required.

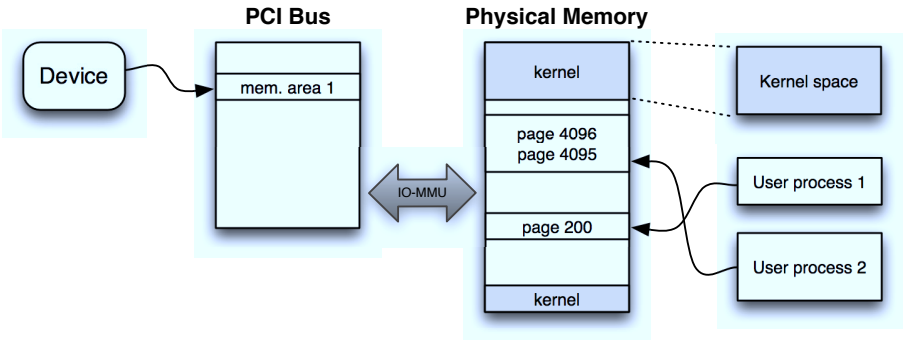


Figure 3.1: Memory Space

The basic infrastructure of the driver follow the guidelines of the book Linux Device Drivers[18]. Traditional Linux device drivers can be described as monolithic drivers: They encapsulate all their functionality inside an opaque interface, exporting information to the device and application using well defined interfaces. The driver code executes exclusively in kernel space, and is expected to manage all operations related to the device from this position. This is the approach taken by some of the other accelerators, like the driver used by the GRAPE cards.

In contrast, our device driver seeks to be a generic PCI driver. We try to leave most of the logic in user space, exporting the required kernel structures and functions when needed. This creates in practice an hybrid driver, with support functions in kernel space for operations that cannot be done otherwise, and leaving all device logic on user space. This approach has given good results in the previous version[44], and is the method used by commercial solutions like Jungo, PLX, and the video drivers of both NVIDIA and AMD used by the GPUs. In theory, we could have used the Jungo driver to provide a functionality similar to the one provided by this driver, but we had two reasons not to do so: after a code review, we concluded that the implementation

of the Jungo driver bypasses many kernel functions and accesses the kernel structures directly (by far, not the recommended way); and it was not clear the implied licensing cost for some organizations like the GSI. It was finally decided to implement our own and provide a generic open source solution for other researchers in the same situation.

The driver structure can be divided in two big, separate blocks: a kernel driver, which we call the driver, and a user space interface which we call the API. The driver takes care of device initialization, memory and IO mapping to both device and user space, interrupts and SysFS interface. The API abstracts the functionality provided by the kernel driver, making the interface platform and version independent.

This approach brings some advantages as well as some disadvantages. On the plus side, the API provides a platform independent layer which allows an easy path for platform migration, as most operating systems provide similar functionality to their devices; the interface also isolates kernel changes from the rest of the software stack; and finally, it provides a cleaner, function driver functionality. On the other side, the export of kernel structures to the system can potentially reduce the system stability, as it gives the application in user space the capability to issue commands that can crash the system (i.e. a wrong DMA descriptor list can cause memory corruption), and forces several operations to be done potentially out of specification. This risk is minimized by the fact that the driver API is accessed in our software stack only by the mprace library, which provides the device logic.

3.2 Kernel Memory

Kernel memory is allocated inside the kernel, and mapped into the user and device spaces. Kernel memory is a single block of contiguous memory, which is most useful for small transfers or devices that do not allow scatter / gather lists. Its setup time is low compared to user memory, and as it is allocated from kernel space, it is quite scarce. Contiguous blocks decrease in size and availability as the system uptime increases, with a maximum size of 8 MB on boot and typical sizes of a few kilobytes.

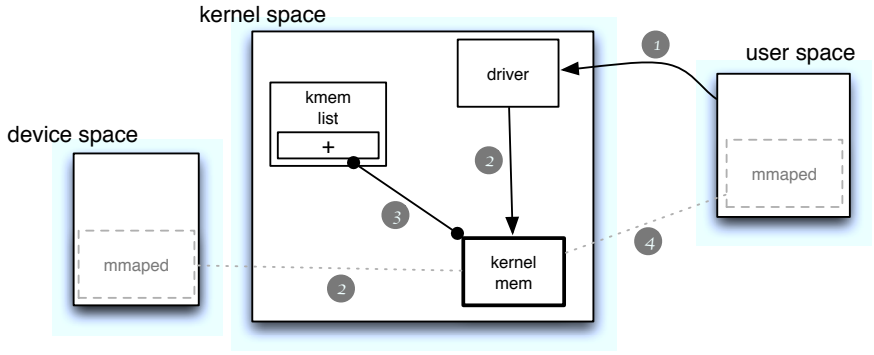


Figure 3.2: Typical kernel memory allocation

In Fig. 3.2 we can see a simplified representation of a kernel memory allocation.

First, (1) the application running in user space requests the driver a new kernel memory buffer of a fixed size. Then, (2) the driver allocates the memory, and during the same process maps the area to the device space, making the buffer available to the device. Next, (3) it adds a new entry to the `kmem_list` list for management purposes. The driver finally returns to the application with the device address (to pass via commands to the device) and an ID of the buffer. Finally, in an separate step, (4) the buffer is mapped to the user space for access by the application.

The dual mapping allows both the device and the application to access the same memory area from their respective address range. In order to guarantee that any modification to a buffer is visible to the other part, the driver provides a `sync` function, that acts as a memory barrier, ensuring all pending operations are committed. The `kmem_list` list serves to keep tracks of allocated resources. It serves to locate buffers quickly when referenced by ID, and can also be used to release allocated buffers manually when an application exists unexpectedly or automatically when the driver is unloaded.

3.3 User Memory

User memory, as its name suggests, is a buffer which is allocated in user space. Memory allocated in user space differs significantly from kernel memory in that it is not a contiguous block, but a collection of segments of physical memory (as small as a page) ordered virtually in user space via mapping. The list describing the mapping of the segments is called a scatter/gather (SG) list. From the user point of view, this mapping is transparent: the application accesses the memory as a contiguous block in its user space. It is even possible (actually, quite common), that a page initialization being delayed by the OS until first accessed (via a minor page fault) or that a page be stored swap memory on disk.

However, in order for the device to access an user space buffer, all pages of a buffer must be available, mapped into memory and into device space. But the mapping to device space is not contiguous as in user space, mostly because some platforms share the memory and device bus. This means the device needs a scatter/gather list for device space, which in turn means each physical memory segment must be mapped to the device space in order. The list is then transferred to the device as needed for accessing the buffer. This is done by the device driver following the procedure in Fig. 3.3, detailed as follows:

On the first place, (1) the buffer is allocated by the application or by the driver library on user space. Next, (2a) the buffer pages are locked in memory and mapped by the driver into device space and (2b) a new scatter/gather list is created, along an entry into the `umem_list`. The driver then returns to the library basic information, like the size of the SG list and handle ID. Afterwards, (3) the library calls the driver in order to retrieve the SG entries of the list. This list is (4) copied into another structure inside the library, and then discarded. The application can now pass to the device, as needed, the provided SG list which contains device addresses for every entry.

The driver requires to store and copy the SG list several times. First, the driver requires a copy as provided by the kernel in order to release it correctly when required.

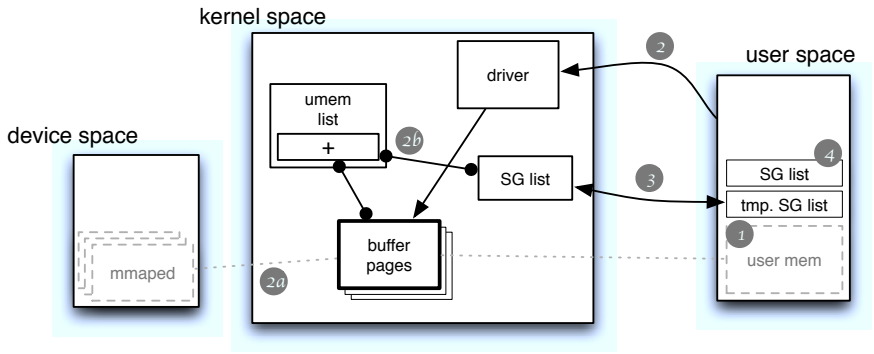


Figure 3.3: Preparation of a user memory buffer for access by the device

But the list provided by the kernel is wasteful and unnecessarily long, because older kernels give a page-by-page list, instead of a merged list. Therefore, when the library requests the SG list of a buffer, it has the option to merge consecutive entries that will make a linear block. By this very simple procedure, the size of the SG list is reduce on average a factor of 6, and the performance of transfers is improved by the use of longer segments. On the worst case scenario (when no segments can be merged), the list remains unchanged and the additional processing time is not significant.

3.4 Interrupt Handling

An interrupt is a signal originating in hardware or software that can be issued by a device in order to raise attention of the CPU and the OS. An interrupt triggers the execution of an interrupt handler, which is a small function registered by drivers with the kernel. Since interrupts are asynchronous and the raising of an interrupt can occur at almost any time, the interrupt handler must execute in kernel space, and once inside a handler, the control must be returned as soon as possible to the normal OS thread.

In our driver architecture, the application waits for an interrupt to be raised. This is correct in the application level, as an interrupt can be handled by a separate process. It is implemented as a function which once called, only returns after the requested interrupt has been raised. However, it poses a problem at kernel level. The kernel expects the interrupt handler to acknowledge the interrupt to a device very quickly, and return the control to the kernel. However, the interrupt handler must at the same time enable the application to continue execution by returning from the wait function. This involves a cross in domain from interrupt-handler mode in kernel level to user code in application level. Some of the most interesting developments in this direction come from the User-Level Device Drivers project of the Gelato Federation, using file system hooks and events for rapid interrupt dispatching. But some of our compatibility requirements clash with their developments, in particular the handling of interrupts over a user-space mapping via SysFS, which is not available in some of the platforms we need to support. The future of the project itself is another matter of concern, as they focused primarily in the Itanium architecture and the development

group seems to have disbanded. So we had to implement a compromise solution.

Our driver registers an interrupt handler for each device during initialization, and creates per device a wait queue for each possible interrupt source (every device is assigned a maximum number of sources). When the application requests to wait for an interrupt, it makes an IOctl call to the driver. The driver then waits for an event in the corresponding event queue, and sends the process to sleep. When the correct interrupt source arrives, the interrupt handler sends an event to the wait queue. This will awake the process, but not immediately. Instead, it will reschedule the process, which will be executed on the next scheduler round after the interrupt handler is finalized. This allows for the handler to finalize fast and cleanly, and to resume the application process without complications. The drawback is that the interrupt acknowledge is not generic, it has to be modified to support every additional device, as every device requires an unique response. This is the only critical point in the driver that requires device specific code, beside the normal initialization IDs, but provides very fast release of interrupts, which is critical for several of the applications we intend to support, like high-throughput data acquisition.

3.5 SysFS Interface

SysFS is a virtual file system that exports system and device driver information. It allows any user with sufficient permissions to communicate directly with the kernel subsystems and device drivers. It replaces some functionality assigned earlier to *procfs*. In our case, we use SysFS to provide a direct way to interact with the driver from a program or the command line without the need for an additional utility. This is useful in order to monitor and debug applications, as well as to recover from crashed programs. Scripts can be written that monitor an application, warn on non-caught interrupts, keep track of allocated buffers or clean up mappings that are not properly released upon exit.

Entries are listed under `/sys/class/fpga/fpgaXX`, where `XX` is the device number assigned to each device by the driver upon initialization. Every device has its own set of entries. Table 3.1 shows a summary of the SysFS entries available for a device, sorted per association. Kernel Memory entries list, allocate, release and reference kernel buffers. Similarly, User Memory entries allow listing, unmap and reference of user memory buffers; no allocation or release is possible, as it must be handled internally by the user process. Interrupt entries provide statistics on the interrupts handled by the driver. Additionally, the SysFS provides links to the entries provided by the PCI subsystem.

In our original design for the PCI driver, no IOctl functions were used, and all kernel driver functionality was provided using SysFS entries. However, early during development we found out that SysFS was not present in versions earlier than 2.6.11, and that several additional functions of SysFS (like the file operation mmap) were not present until 2.6.13. We found also some inconsistencies depending on the distributions, as some decided to incorporate SysFS earlier than others. Therefore, we had to reduce the importance of SysFS from our original design to a mainly informational source and keep usage of IOctl functions as the main communication channel.

Kernel Memory	
`kbuffers`	List the currently allocated kernel buffers
`kmem_alloc`	Receives the size, returns the ID of the new buffer
`kmem_free`	Receives the buffer ID, and releases it
`kmem_count`	Returns the value of the internal buffer counter
`kbufXX`	Access the buffer with ID XX

User Memory	
`umappings`	List the mmapped user buffers
`umem_unmap`	Receives the buffer ID, and unmaps it
`umemXX`	Access the buffer with ID XX

Interrupts	
`irq_count`	Return the total count of interrupts received
`irq_queues`	List the number of pending interrupts in the wait queues

Table 3.1: SysFS entries

3.6 PCI Driver API

The PCI driver API provides an abstraction of the functionality provided by the kernel driver. Its main purpose is to give a layer of abstraction which isolates the specifics of the kernel and allows multiplatform support. In addition, it also provides locking mechanisms for certain operations that have to be performed atomically. Most of the time, the driver interface is used only by a support library that provides higher level functionality. In this sense, the PCI driver API is not intended for direct use, but it is needed if a new platform is to be supported. The PCI Driver API provides similar functionality in diverse interfaces. The C++ interface is the main and most used, as is the one used directly by the mprace library. The C interface is provided as convenience for future developments, and the compat interface is used for compatibility with the uelib, as a replacement of the original PCI driver interface.

3.6.1 C++ Interface

The C++ interface reorganizes the IOctl calls around 3 basic classes: `PciDevice`, `KernelMemory` and `UserMemory`. Both `KernelMemory` and `UserMemory` represent the memory buffers which can be created, and provide functions to get the mapping information needed for a device to access them. However, their creation is handled directly by the `PciDevice` class, which also provides all other functions, like IO BAR mappings and PCI configuration space RW. Fig. 3.4 shows a class diagram of the whole C++ API.

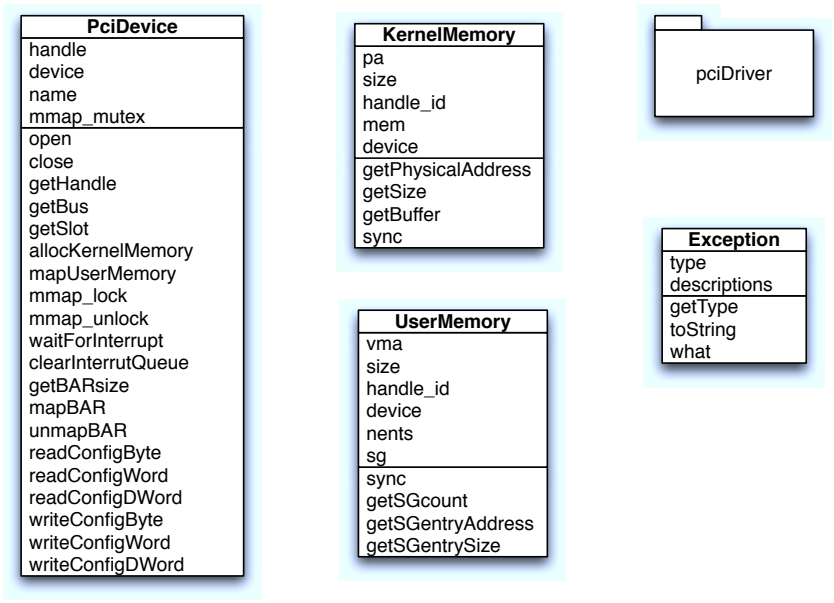


Figure 3.4: Class Diagram of the C++ API

3.6.2 C Interface

The C interface is a reimplementation of the C++ interface in C. It provides similar structures as the C++ classes, and same functions. While it could have been implemented as a wrapper of the C++ interface, it was decided to reimplement them fully, as the complexity is not high and the overhead of a C++-to-C wrapper would have been too high. Tab. 3.2 summarizes the C API.

3.6.3 Compat Interface

The Compat interface (short for Compatibility), reimplements the old PCI Driver interface using the new driver. It is intended to be a drop-in replacement for older code, enabling the use of the uelib library and all their code base for MPRACE-1, μEnable and their applications.

Structures	
`pd_device_t`	Represents a PCI device
`pd_kmem_t`	Describes a Kernel Memory buffer
`pd_umem_t`	Describes a User Memory buffer
`pd_umem_sgentry_t`	Represent a single SG list entry

Device Functions	
`pd_open`	Open the device
`pd_close`	Close the device
`pd_mapBAR`	Map a BAR to user space
`pd_unmapBAR`	Unmap a BAR from user space
`pd_getBARsize`	Return the size of a BAR area
`pd_readConfigXX`	Read XX from the PCI config. space
`pd_writeConfigXX`	Write XX to the PCI config. space

Kernel Memory	
`pd_allocKernelMemory`	Allocate a Kernel Memory buffer
`pd_freeKernelMemory`	Release a Kernel Memory buffer
`pd_syncKernelMemory`	Synchronize the content of a Kernel Buffer

User Memory	
`pd_mapUserMemory`	Map User Memory to the device
`pd_unmapUserMemory`	Unmap User Memory from the device
`pd_syncUserMemory`	Synchronize the content of a User Buffer

Interrupts	
`pd_waitForInterrupt`	Wait for an Interrupt from a device
`pd_clearInterruptQueue`	Clear the interrupt Queue for a device

Table 3.2: C interface

Chapter 4

The MPRACE library

The MPRACE library is the next level library in our software stack. It sits directly above the PCI driver, and provides additional, higher functionality to our FPGA boards. Its main purpose is to provide common operations used by applications, operations like register IO, DMA transfers and FPGA configuration. These are functions specific to the developed hardware and must be put together for every new board developed. In this sense, the MPRACE library is the successor of our *uelib* library, which provided similar functionality to our older boards: the μEnable, ATLANTIS and MPRACE-1.

With the new MPRACE library, we aimed at supporting the MPRACE-2 and ABB boards, with the option to extend to others. The main reason to start a new development, instead of extending the old one to support the new hardware, is that a lot of the old code was specific to PLX-based PCI controllers, like the PLX9656 on the MPRACE-1. Since the new boards are PCIe-based and use a mixture of software and hardware cores inside the FPGAs, little code could be reused. In addition, many FPGA families of the old boards are no longer supported by the development tools, so it was decided to be phased out and clean the code, in order to avoid the false impression that some new designs could be supported by boards of the very old series. Nevertheless, if some are eventually needed, we can still use the PCI driver wrapper to support them using the old *uelib* library.

From the application point of view, the library abstracts the functionality of most of the common cores added to our designs, allowing the user to concentrate in the development of FPGA applications and software instead of routine tasks. While register IO is a relatively simple task, boards like the MPRACE-2 have a DMA engine capable of scatter/gather transfers, an operation that requires significant initialization on the software host side.

4.1 Architectural Overview

The architecture of the library is centered around the `Board` class and its subclasses. All other elements are support components to the main interface provided by these classes. In a general sense, classes in the library represent either physical components (like an MPRACE-2 board), logical components (like a DMA engine) or software components

(like a DMA buffer). Fig. 4.1 shows a class diagram of the main elements of the library. We can highlight the different boards available, the use of the DMA Engine in them, and the relationships with the PCI driver library objects.

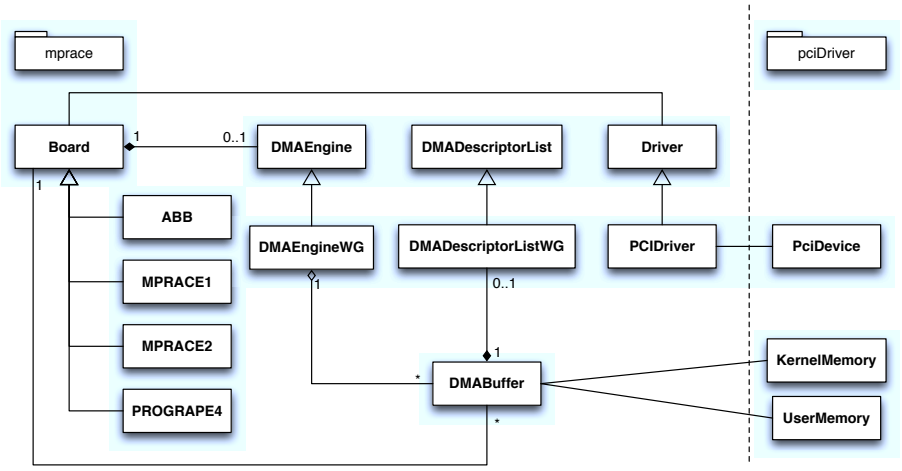


Figure 4.1: MPRACE library class diagram

4.2 Register Mapping

Register read and write is the most basic IO operation that any system can perform, but even such a simple task can be performed in several different ways. In the most basic mapping, a register behaves exactly as a memory location: writing to it sets the register to a new value, reading from it returns its current value. Some architectures handle this as a completely separate address space (i.e. the SPARC architecture), while others handle registers as memory mapped areas, where a memory address is catched and redirected to access the register instead. A similar method is used to map whole areas of a PCI device into the PCI bus: the device provides one or more base address registers (BARs) and their sizes in the device configuration area, each which are assigned (mapped) during boot to an non-colliding address range in the PCI bus address space. Then it is the task of the operating system and the driver to map the BARs and make them accessible to the application. This library maps these areas into the application memory space and uses wrapper functions to provide a clear and consistent interface for the user: namely a `getRegister()` and `setRegister()` pair of functions.

While this method is sufficient for most operations, other registers (by example, IO pins), might require separate addresses for read or write, a direction and/or a high-impedance control, each one which might be mapped to an additional register. This arrangement is very common in small microcontrollers, like those of the PIC[4] and AVR[3] families (from Microchip and Atmel, respectively), because of their simple implementation and ease of use. For the same reasons, they are also used sometimes in FPGA designs. As an example, our designs use this approach to map the SelectMAP

interface on the MPRACE-2 board, which permits the programming of the Main FPGA from the Bridge. In order to make the access of any of these configurations easier for the potential developer, a set of classes are provided that implement the respective methods.

4.3 DMA Buffers

A `DMABuffer` class gives a consistent interface to two basic, dissimilar entities: the Kernel Memory buffer and the User Memory buffer, both structures provided by the underlying PCI driver. While the mapping to user space is very clear and comes in the form of a regular pointer, the additional data required by the DMA Engine to perform a transfer is very different on each case. The `DMABuffer` class provides encapsulation of this data and a convenient way to pass it to the library on each transfer request.

Because a kernel buffer is by definition a contiguous area, an user pointer, a physical address and a size is enough to completely describe the buffer and its mapping into user space, and all of them are provided by the `KernelMemory` class of the PCI driver. On the other hand, an user memory buffer is composed of a collection of memory pages. From user space it is a single contiguous area, but the sequence of the physical pages involved is defined in the scatter/gather list. However, the scatter/gather list provided by the driver is not guaranteed to be neither available nor arranged in a way compatible with whatever structure is required by the DMA Engine used by the board. Therefore, we abstract this additional mapping with a new `DMADescriptorList` class.

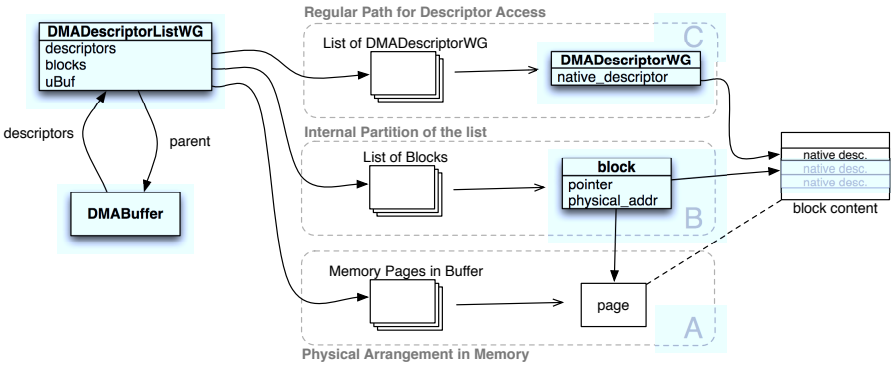


Figure 4.2: DMA Descriptor List diagram

The `DMADescriptorList` class is a virtual class, which simply abstracts the access methods for an array of hypothetical `DMADescriptor` elements. What really interests us is the `DMADescriptorListWG` class, which implements it for the `DMAEngineWG`, as shown in Fig. 4.2. Each element of the `DMADescriptorListWG` array is of type `DMADescriptorWG`. The class actually composites two arrays. One array is a linear array of `DMADescriptorWG` elements, where each element references to a native descriptor (section C). This is the array that is accessed regularly when manipulating the descriptor list. The second array is used to store the native descriptors as required by the board, and organized as a collection of blocks where each block is a single memory page

from an independent DMA Buffer, this is the section A. This means the native descriptor list is stored in a DMA buffer which is accessible by the board. Each block links a native descriptor to an element of the scatter/gather list, providing the board access to the list required for processing a DMA transfer, while at the same time linking the necessary data structures to the `DMABuffer` being transferred. The `DMADescriptorListWG` limits the number of `DMADescriptorWG`s that can be stored per block in order to guarantee that there is neither a partial descriptor in a block nor the descriptors cross a page memory boundary. This relationship is depicted in section B. Section B is initialized by the `DMAEngineWG`, and once setup, allows for fast manipulation of the descriptors.

4.4 DMA Engine

The DMA Engine is a module that handles transfers between the device and host memory autonomously, offloading the CPU from this task. The DMA Engine is located in the board, and it can be part of an integrated circuit, like the PLX9656 in the MPRACE-1, or part of an FPGA design like the DMA Engine WG in the MPRACE-2 (WG stands for Wenxue Gao, the main developer of the module). Each of them needs a software counterpart in the library that implements the necessary logic to initialize, monitor and finalize each transaction. The virtual class `DMAEngine` defines the minimum interface needed to perform these operations, while the `DMAEngineWG` class implements it for the DMA Engine WG module.

In particular, the following sections will describe how descriptor lists are assembled for several types of transfers, using the structures described in the previous section. The *DMAEngineWG* has two independent channels, one for each direction of transfer: Host-to-Board and Board-to-Host, that allows the engine to operate concurrently in both directions.

4.4.1 Descriptor List Assembly

Each native descriptor of the *DMAEngineWG* follows the structure shown in Fig. 4.3. Each DMA descriptor represents a contiguous memory area to be transferred, referred by its source and destination addresses, and its size. The control word has several flags, that are used in different stages of the transfer. The next descriptor field is used to point to the address in host memory with the next descriptor of the list, effectively building a linked list. The engine is capable of fetching the next descriptors from host memory without additional program intervention.

In order to perform a transfer, the host prepares a descriptor list with one or more descriptors, and then copies the first descriptor (the head of the list) to the appropriate channel registers in the board to start the transfer. Depending if the transfer is blocking of non-blocking, the software will wait until the transfer finishes or times-out, or will return immediately and make the application responsible for monitoring the status itself.

The list of descriptors contains three important points to consider: the first, the second and the last descriptor. The first descriptor is the one loaded directly into the engine. This descriptor is normally copied into, not referenced by the board for remote

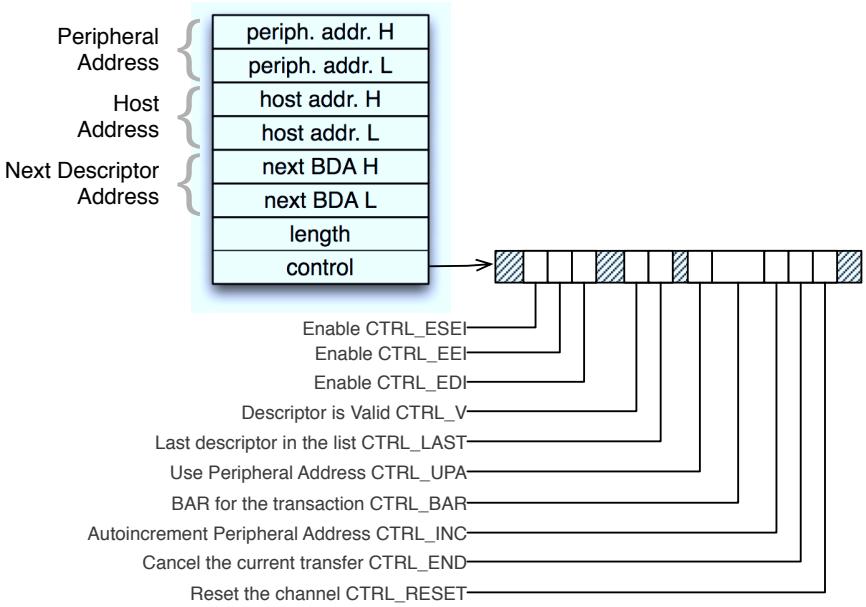


Figure 4.3: DMA Descriptor

access. This is also the only descriptor that defines the peripheral (device) address, as for all other, this address is computed internally by the engine, based on this address and the increment control flag. The second descriptor is the first loaded automatically by the engine. Both the first and the second must contain the appropriate transfer flags for BAR destination, board address increment and interrupt signalling. These flags are not required in the remaining descriptors. The last descriptor must also contain the last descriptor flag, signalling the next descriptor address to be invalid and the end of the transaction.

In theory, the descriptor list could be constructed before each transaction, and destroyed when it's finished. However, building a descriptor list is a time consuming task, especially when the user buffer is very big. A buffer with hundreds of megabytes contains thousands of memory pages, which can be distributed in hundreds of segments and require a list that spans tens of blocks itself. This is summarized in Fig. 4.4, where each sample point is averaged 500 times. The allocation time is plotted against the buffer size as well as the number of descriptors and the number of blocks used. It can be seen that for small buffers, the allocation time is mostly constant, as it is dominated by the algorithm overhead, then it scales linearly. It is worth to remark the jump present on the plot at 32MB buffer size. The best explanation for it is related to the underlying allocation algorithm from the linux kernel, where the memory manager chooses to use bigger consecutive blocks (possibly bigger pages too) to precisely keep the size of the descriptor lists under control, at the expense of a longer allocation time. Finally, the descriptor list remains mostly static because pages are pinned in memory, so only 3 descriptors at most need to be modified per transaction, independently of buffer size. Therefore, the list is built only during buffer allocation, and modified as needed for

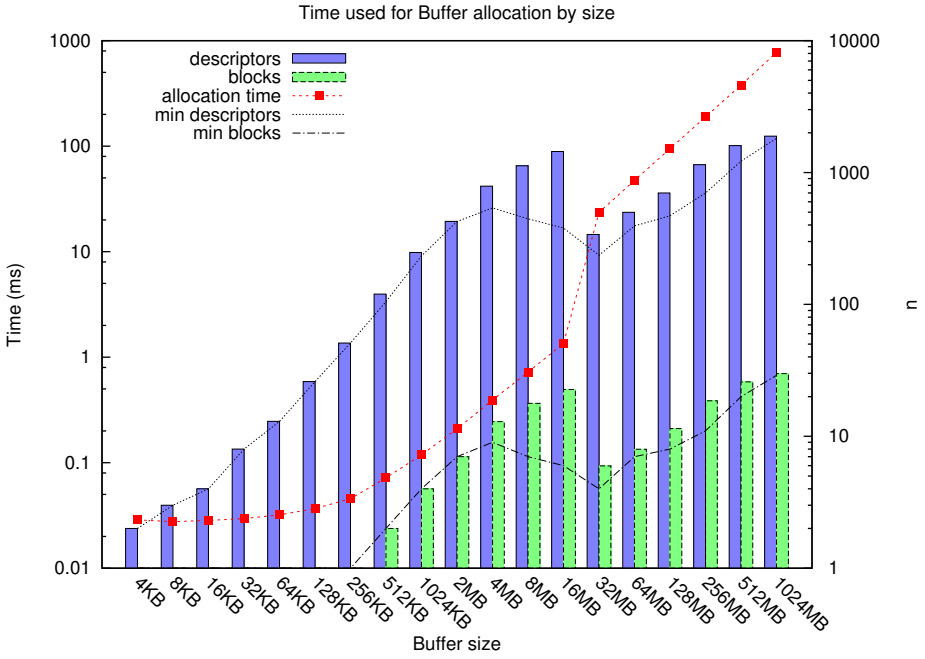


Figure 4.4: DMA Descriptor Characteristics per Buffer Size

each transaction.

We now need to consider 8 cases: two buffer types, Kernel and User Memory, with four possible scenarios each, based in offset and size of the transfer: a full buffer transfer, a partial (smaller than maximum) transfer, with an initial offset, or both.

The simplest case is the transfer of a kernel buffer, because it requires a single descriptor. Therefore, the four cases can be summarized easily in the table 4.1.

parameters		descriptor configuration	
offset	size	hostAddress	count
0	max	physicalAddress	max
0	X	physicalAddress	size
X	max	physicalAddress+offset	max-offset
X	Y	physicalAddress+offset	size

Table 4.1: Kernel DMA Descriptor cases

The user buffer cases are more complex. Since the User Memory buffer has a list of descriptors, modifying the starting or the ending point of the transfer requires walking the descriptor list to identify the first, second and last descriptors that correspond to the transfer setup. Then, their contents are backed up to be restored at the end of the transfer, and finally, the entries and control flags are modified appropriately.

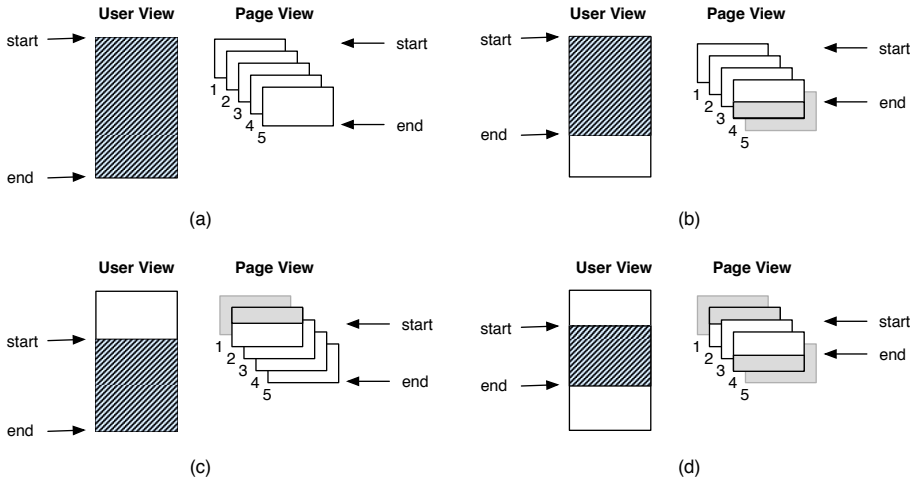


Figure 4.5: User DMA Descriptor cases

Fig. 4.5 summarizes the four cases for the user buffer, as seen by the user buffer and its corresponding page view, which correlates to the descriptor list. The diagrams show how certain areas of the buffer map into the pages. The diagram (a) shows the transaction for the full buffer (offset=0, size=max); the next diagrams are for (b) size $<$ max and (c) offset $\neq 0$; and case (d) when both offset $\neq 0$ and size $<$ max . On the easy case (a), the full buffer is transferred. This means that for the prebuilt descriptor list, the first descriptor is the first of the list, and the last is the last, so little to no modification of the control flags is necessary. On case (b), only the last descriptor needs to be modified. Therefore, the `DMAEngineWG` has to walk the descriptor list until it reaches the requested size for the transfer. Then it saves this descriptor for later restore and modifies it to signal the last descriptor. Similarly, for case (c), it has to walk the list to find the first descriptor, which is going to be copied to the board. Finally, case (d) has to do both: walk the descriptor list to find the first descriptor, then continue until it reaches the required size.

4.5 Performance

In Fig. 4.6 we summarize the DMA performance of the library. For the test, a host with an Intel Xeon E5420 @ 2.5 GHz was used, together with an AVNET Virtex-5 V5LX110T board using a DMA Engine developed by Wenxue Gao[30]. A transfer sweep from 4KB to 64MB gives the transfer function until the bus saturates. This occurs at ∼700 MB/s for DMA writes and at ∼380 MB/s.

The 4-lane PCIe interconnect is capable of 10 Gbit/s transfer on each direction (full-duplex), which leads to 1 GB/s after taking into account the 10/8 bit encoding. This does not take into account the PCIe protocol and OS stack overhead. From the protocol alone, we know that for every packet with 20 words of data, 2 additional words are used as header. So, from the protocol alone, we can estimate roughly 10% overhead. With simpler DMA engines, we have measured maximum performance of 800 MB/s for

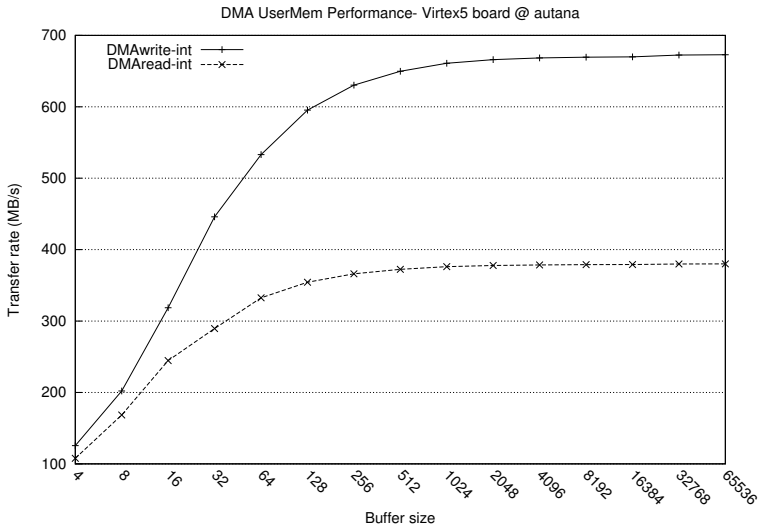


Figure 4.6: DMA Performance in a 4-lane Virtex-5 board

writes and 700 MB/s for reads, dependent on the platform. An additional 10% can be accounted for by the SW IO TLB mechanism, which uses an additional bounce buffer for the device-to-memory transfer. We can conclude that a 700 MB/s for DMA writes is a sensible result.

However, the DMA read performance is below the expected level by about half the bandwidth. We have experienced severe degradation depending on the platform in spurious tests, that is, significant differences between chipsets and between AMD and Intel architectures. However, the results presented here are verified over several platforms and chipsets with less than 5% performance difference. It is therefore attributed to a bottleneck in the FPGA design, that is currently under investigation.

Chapter 5

The Buffer Management Library

To achieve the closest match to the theoretical performance maximum of a communication channel with the minimum resource usage has been for a long time a goal of device driver and application developers. Using diverse buffer management algorithms, like Buffer partition in chunks or Double Buffering to achieve these goals, has been done for years. However, FPGAs pose a particular challenge as their communication patterns are determined by the design loaded, precluding optimizations of data transformation. Then must the application take care of these optimizations. The Buffer Management library provides a framework to facilitate this task.

The Buffer Management Library (`bufmgr` for short) provides optional functionality for efficient use of the DMA buffers and DMA transfers. It provides a common interface to transfer a certain buffer in memory using only a limited amount of buffers efficiently, following certain known buffer algorithms. In addition, it provides the ability to decrease the number of copy operations needed when data has to be converted to a certain format before or after the transfer, by merging the data transformation into the data copy loop of the selected algorithm.

The library fills several needs. For once, it reduces the amount of memory needed as DMA buffer to a fixed amount (for a certain buffer algorithm), independent of the amount of memory being transferred. This means a multi-gigabyte array can be transferred efficiently using only some megabytes of DMA buffers. This saves time and complexity, especially when transfers are of varying size during the life of the application. In addition, the library improves the performance of applications which are transfer-bounded. A transfer bounded application is one where the performance of the algorithm is bounded by the data transfer speed from or to the host, as is the case in our SPH designs (see Chapter 6 for more details). This improvement is based not only in a more optimized data transfer loop, but in the ability to merge the data format transformation into the buffer algorithm, once again, as needed by our SPH designs.

Besides the data transformation, the library provides advantages in the form of reduced DMA buffer usage, which translates in less pinned memory; multiple backends (for the MPRACE and uelib libraries); and a simple and common interface for even complex buffering schemes. These translates into less coding and more efficient transfers without adding complexity to an application.

It is worth mentioning that unlike the PCIDriver or the MPRACE library, the

Buffer Management library is entirely optional. The boards can be used and the applications can be written that take full potential of the FPGAs without using it. As such, this library is provided as a tool to maintain performance and simplify application development.

The library evolved in two parts, which are called version 1 (v1) and version 2 (v2). Version 1 is a class hierarchy of `BufferManager` classes, which implement translation as class composition. This version provides support for the uelib set of functions only, as it was developed before the MPRACE library was finalized. Version 2, in the other hand, is a set of template classes, which uses compile time composition via template parameters to add back-end and translation operations. The rest of the chapter discusses the buffering algorithms implemented, the translation mechanisms used, and the performance of both approaches.

5.1 Buffering Algorithms

The Buffering algorithms is the core of the library functionality. It organizes one or more DMA buffers under a common interface that provides varying degrees of complexity and performance. The simplest buffer manager is just a wrapper around a DMA Buffer. Others provides partitioning, allowing to transfer an unlimited amount of data using either one, two or many DMA buffers. Some of them provide concurrent transfer and copy operations, others do not. Depending on the need of the application, a simpler buffer can provide better performance for small transfers (as the overhead is lower), while a more complex one would be advantageous in complex data transformations. An overview of the buffer structures is shown in Fig. 5.1

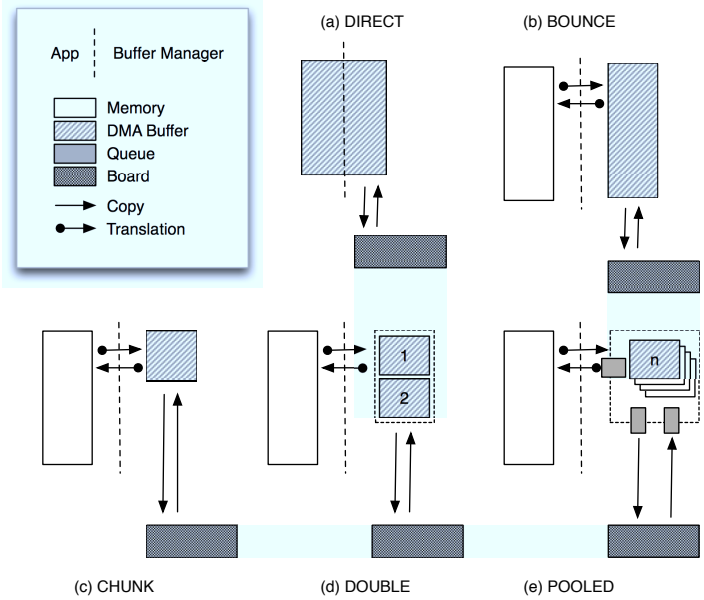


Figure 5.1: Buffer Managers

In the direct buffer scheme, shown in 5.1(a), a memory region is allocated and provided to the application. This memory region is the actual buffer to be transferred, allowing for direct manipulation of the buffer to be sent without the need to copy any data around. This limits the amount of memory that can be allocated for this task and prevents the data from being translated. This buffer is a wrapper for the DMA buffer, and it does not provides any data transformation capability. A major disadvantage is that the memory must be allocated by the manager, preventing or making it very complicated to interact between different programming languages like C++ and Fortran, as pointers are handled differently. This manager is seldom used and is provided as a reference implementation, only in the v1 part of the library.

Bounce buffers, shown in 5.1(b), the buffer manager uses a memory region provided by the application as source or destination, depending if it is a write or read operation. The actual read or write is done over an internal buffer, so the data has to "bounce" over the internal buffer before being transferred. The data has to be copied once between the bounce and the source/destination buffer, but provides the separation of memory regions (the application memory area does not have restrictions on memory type) and translation is also allowed during this copy. It also makes interoperability between languages much easier. This scheme duplicates memory consumption, and still limits the amount of memory that can be transferred, as it cannot be bigger than the internal buffer. Both Direct and Bounce Buffers are implemented by our `SimpleBufferManager` class in the v1 part of the library.

A chunk buffer, shown in 5.1(c), uses a small internal buffer similar to a bounce buffer, but instead of copying all the data at once, data is processed (translated and sent; or received and translated) in small portions, or chunks. Memory consumption is reduced to the size of the internal buffer. The communication pattern changes, as the chunk buffer is now a cycle of translate/transfer operations. The conversion and transfer are done sequentially, as both operate over the same memory region, preventing them from overlapping. This causes the CPU to be idle while the transfer is being performed, wasting some resources, but saving potentially a lot of memory, which will save time if memory would otherwise need to be swapped in and out of memory. This scheme is implemented by our `ChunkBufferManager` class, and by the `TChunkBuffer` template.

Fig. 5.1(d) shows the traditional Double Buffer algorithm. In the double buffer scheme, two internal buffers are used instead of one. While one buffer is being transferred, the other can be filled and translated, allowing for the overlapping of both operations. Memory consumption is doubled compared to the chunk buffer, but resources (CPU and bus transfer) are used more efficiently. In the ideal case, the translation of data takes exactly the same time as the data transfer, allowing both the CPU and the transfer to be pipelined without any contention waiting for the next chunk. This scheme is implemented in the `DoubleBufferManager` class and the `TDoubleBuffer` template.

For the Pool of Buffers scheme, shown in 5.1(e), a variable number of buffers are used. Buffers are allocated from a pool as needed and returned when not used anymore, and transfer and conversion operations are handled asynchronously by means of queues. Three queues are needed: one for writing chunks to the board, one for reading chunks from the board, and another to handle the conversion of read chunks. This provides greater flexibility for the transfer to accommodate under different conditions

of bus contention or CPU load, but the more complex handling increases the overhead, so it is most useful when the data transformation is a complex operation. Memory consumption depends of the amount of buffers present in the pool. This scheme is implemented by the `PooledBufferManager` class and the `TPoolBuffer` template.

5.2 Translation Mechanisms

Most buffer algorithms must implement a copy loop, where data is copied from the source structure in regular memory, into a DMA buffer for its transfer; or being received into a DMA buffer and then copied into the main memory. The translator mechanisms provide the means to integrate the transformation of data into the copy loop of the buffer algorithms. Therefore, the buffer algorithm can use the same copy loop to perform operations over the data as it is read or written, saving one copy operation that would otherwise being implemented as an additional loop or another completely separate operation. Similarly, implementing the optimized version without the `bufmgr` library involves rewriting the algorithm to include the specific translation.

The separation from the translation and buffer algorithms simplifies the development, as the algorithms can be heavily reused, and customized via translators to perform the required customization without loss of generality, and very little to no loss of performance (see Sec. 5.3).

The translation mechanism is implemented different in v1 and v2. v1 uses subclassing and class composition, which allows dynamic change in runtime of the translation mechanism. v2 uses templates and compile time composition, so the transformation is defined at compile time. This was decided to be desirable, as it gives the compiler additional optimization opportunities, and the runtime change is seldom used. In case several different translators are needed, multiple translator implementations can be shared over the same DMA buffer set, as the buffers are not bounded to a single manager. Clearly, they cannot be used concurrently in this case, if the buffer is in use by another manager.

5.2.1 Translation by subclassing

The `Translator` class is an abstract interface that the application extends by creating and using a derived class. It allows the buffer manager to translate a single element from format *In* to format *Out* and to translate a set of elements, with additional parameters to adjust input and output offsets. A simple translator class that multiplies one integer by 2 is shown in Listing 5.1. It extends the Translator class, implementing the methods required by the managers.

The `BufferManager` class relies on this interface to save an intermediate copy operation by adjusting the input and output offsets when dealing with chunks. This separation also makes possible to implement advanced, cache optimized conversions (by example, by using SSE instructions or memory prefetch commands) without any need to modify the buffering scheme. All required classes are included in a library, and the developer implements subclasses of the translators as needed. The application uses a simple read/write interface to the buffer managers, passing arrays of the desired data

Listing 5.1: A Dummy Translator

```
class DummyTranslator : public Translator {
public:
  DummyTranslator()
  {
    inSize = sizeof(int);
    outSize = sizeof(int);
  }

  inline bool convert(void *in, void *out)
  {
    int *i = static_cast<int *>(in);
    int *o = static_cast<int *>(out);

    *o = (*i)*2;

    return true;
  }

  inline unsigned int convertMulti(
    unsigned int count,
    void *in, void *out,
    unsigned int inOffset,
    unsigned int outOffset )
  {
    int *i = static_cast<int *>(in);
    int *o = static_cast<int *>(out);

    for(int j=0; j < count; j++) {
      o[j+outOffset] = i[j+inOffset]*2;
    }

    return count;
  }
};
```

types.

5.2.2 Templatized Translators

In v2, the translators are implemented as methods in a class with an specific signature. This class is used as a parameter when the template is instantiated, and the translation mechanism fixed for this implementation of the template. This approach, as mention earlier, has the advantage of giving the compiler more optimization opportunities at the cost of a small degree of functionality: translators cannot be changed in runtime, without losing any of the main advantages of using the library. Listing 5.2 shows the same example as before, as it is used for a templatized buffer.

In addition, the templatized buffers separate the DMA operations by a similar mechanism, allowing the buffer managers to have a back-end which we use to exchange the library used for communication with the boards: the `uelib` or the `mprace` libraries.

5.3 Profiling and Performance

As the main goal of using this library is to provide better performance for the transfers, we present in the following sections a detail characterization of both versions of the library. We include in the library a set of profiler classes that allow the easy characterization of both classes and templates, so the user can run similar tests for their specific setups.

5.3.1 Performance of the BufferManager classes

A workstation with an Intel Xeon processor at 2.66GHz, a 533MHz FSB and 512KB of L2 cache on a Broadcom ServerWorks chipset motherboard, 2GB of DDR-266 RAM, a mpRACE-1 board and running RedHat Enterprise Linux 4 was used for the tests. The mpRACE-1 board is equipped with a Xilinx Virtex-II FPGA[98] and a PLX Technologies 9656 interface controller [5] capable of 264MB/s using a PCI-X bus at 66MHz. The FPGA was loaded with a simple design that provides a memory region for IO access.

To measure time precisely during program execution, we utilize the Timestamp Counter (TSC) of the processor, allowing the readout of clockticks under normal conditions (non-sleep modes). This provides nanosecond resolution for measurement of transfer times. Each transfer measure is repeated 10 times and the result is averaged to minimize spurious behaviour.

Performance without Translation

First, the performance of direct buffer accesses without any data translation is measured and displayed in Figs. 5.2-5.3. They provide information about the maximum performance that can be achieved, as any scheme ultimately relies on a direct transfer. It also shows the overhead associated with the transfer setup and any other device driver operations involved. In out testbench, we obtain a maximum of 255MB/s from a theoretical maximum of 264MB/s, but only if the transfer is bigger than 32KB. This is one of the important observations: The transfer rate varies over different transfer

Listing 5.2: A dummy translator for the templatized buffer managers. It is functionally equivalent to the DummyTranslator presented in Listing 5.1

```
// Header
class TrDummy {
public:
        typedef int host_type;
        typedef int board_type;

        static void host2board(
                unsigned int const count,
                int *in,
                int *out,
                unsigned int const in_offset,
                unsigned int const out_offset
        );

        static void board2host(
                unsigned int const count,
                int *in,
                int *out,
                unsigned int const in_offset,
                unsigned int const out_offset
        );
};

// Implementation
        void TrDummy::host2board(
                        unsigned int const count,
                        int *in,
                        int *out,
                        unsigned int const in_offset,
                        unsigned int const out_offset
        )
{
        for(int i=0;i<count;++i)
                out[out_offset+i] = in[in_offset+i] / 2;
}

        void TrDummy::board2host(
                        unsigned int const count,
                        int *in,
                        int *out,
                        unsigned int const in_offset,
                        unsigned int const out_offset
        )
{
        for(int i=0;i<count;++i)
                out[out_offset+i] = in[in_offset+i] * 2;
}
```

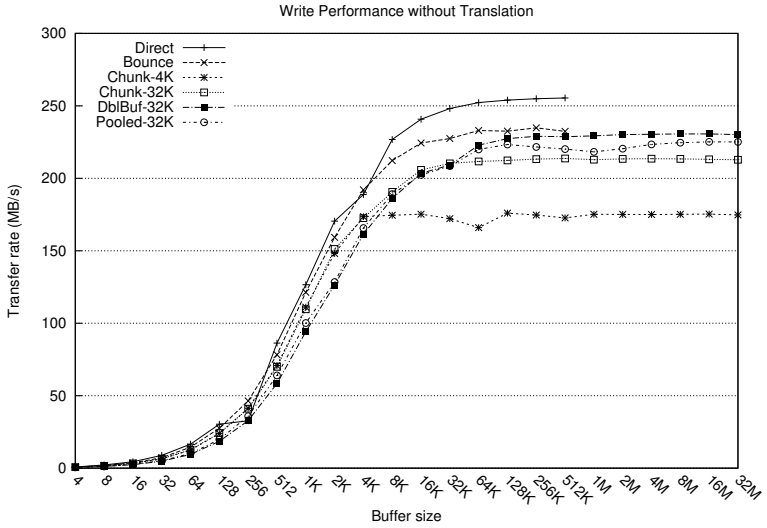


Figure 5.2: Write without translation

	Write (eff)	Read (eff)
Direct	255 MB/s	255 MB/s
Bounce	234 MB/s (91%)	234 MB/s (91%)
Chunk-32K	213 MB/s (83%)	198 MB/s (77%)
DblBuf-32K	230 MB/s (90%)	215 MB/s (84%)
Pooled-32K	225 MB/s (88%)	215 MB/s (84%)

Table 5.1: Transfer Rate Efficiency (w/o Translation)

sizes. This leads to the exploration of the different techniques here presented. The figures translate into 96% efficiency for the direct transfer. From now on, all efficiency numbers are compared to the direct transfer. Efficiency numbers for the cases shown in Figs. 5.2-5.3 are presented in Tab. 5.1.

Using a bounce buffer inserts a copy operation of the full buffer, and provides information about the cost of simply copying the data around from one memory region to another. It also enables the use of translators, so this is the maximum performance that can be achieved when using a translator, which will be discussed in section 5.3.1.

By using chunks, we add two more variables: the size of the chunk, and the number of chunks used. A single chunk is used by the Chunk Buffer (Chunk), two by the Double Buffer (DblBuf), and 32 by the Pooled Buffer (Pooled). Tests were run with chunk sizes between 4KB and 1MB, but only selected cases are presented due to space restrictions. Chunk-4K is presented to show that for small chunk size, the maximum transfer rate is the limiting factor.

What is important to notice from these figures is that for sensible chunk sizes, none of the Buffer Managers present significant performance degradation when operating

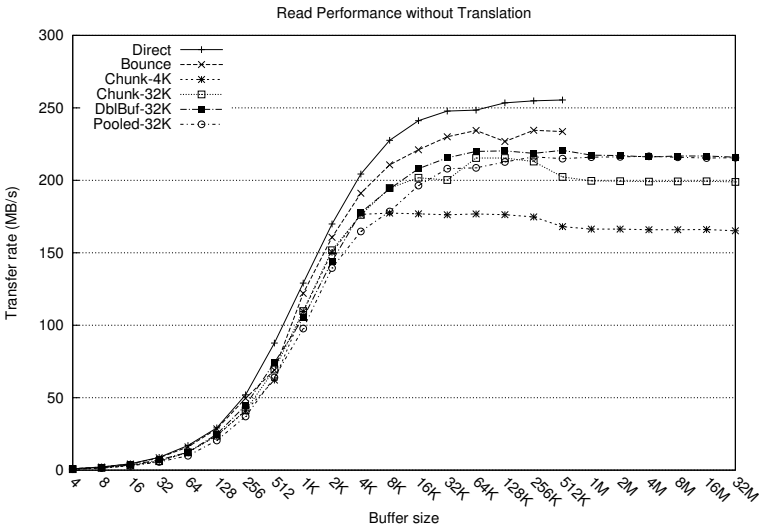


Figure 5.3: Read without translation

without translation. The use of a Double Buffering or a Pooled scheme looks like the best choices, providing 88% efficiency for writing and 84% for reading. The effects of choosing non-optimal chunk sizes will be discussed in section 5.3.1.

Performance with Translation

For the translation tests, a Translator that converts double precision floating point numbers to single precision is implemented. This operation requires the use of the FPU unit of the CPU, so this translator effectively forces the data to be fetched and processed. It also exemplifies a critical conversion operation used by our application libraries interfacing to a custom core that uses floating point arithmetic.

On Figs. 5.4-5.5, write and read performance plots for transfers with translation, and direct access performance is displayed as reference. When using a Bounce buffer, data is translated as it is copied to/from the internal buffer, but can be seen that performance drops dramatically as it approaches 512KB. Evidence suggests that it is the effect of increasing cache misses as the size of the buffer get closer to the physical cache size of the CPU. As a general fact, the translation puts more pressure in the transfer performance for reading data from the board to the host.

The same behaviour can be seen when using a single chunk, where performance shows a drop and stabilizes in after 512KB. The performance improves with the use of double buffers or pooled buffers compared to the single chunk, as both schemes allow for overlapping of transfer and conversion time.

From Tab. 5.2 can be seen that the Double Buffering scheme shows the best efficiency with 89% for writing and 76% for reading, with the Pooled scheme below by a 2%. More important, when compared with the figures of Tab. 5.1, the translation adds only 1% overhead for writing, and 8% for reading.

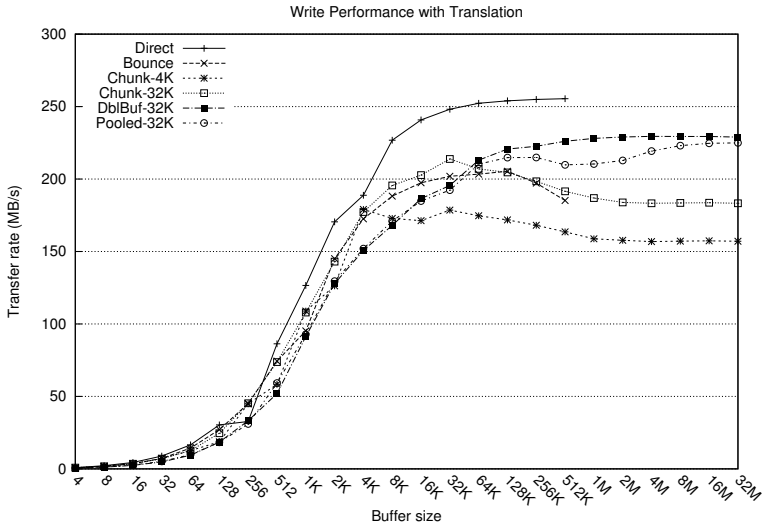


Figure 5.4: Write with translation

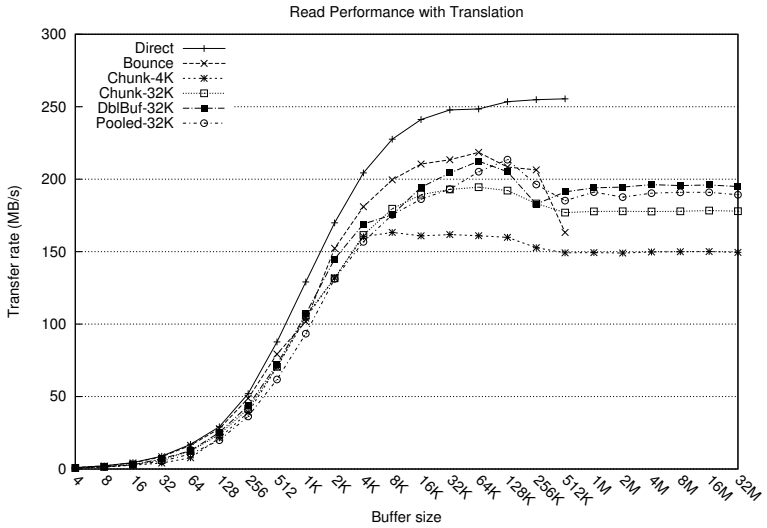


Figure 5.5: Read with translation

	Write (eff)	Read (eff)
Direct	255 MB/s	255 MB/s
Bounce	185 MB/s (72%)	163 MB/s (63%)
Chunk-32K	183 MB/s (71%)	177 MB/s (69%)
DblBuf-32K	228 MB/s (89%)	194 MB/s (76%)
Pooled-32K	224 MB/s (87%)	189 MB/s (74%)

Table 5.2: Transfer Rate Efficiency (w/ Translation)

Non-optimal Chunk Sizes

On the previous sections, chunk sizes were selected for good performance. But an application writer may choose chunk sizes that lead to non-optimal results. In this section we present the consequences of using too small or too big chunk sizes for the different buffering schemes.

The result of choosing a small chunk size has already been presented in Figs. 5.2-5.5, with a size of 4K. When it is below 32K the transfer time is limited by the Direct Transfer plot for that size, so full performance is not possible above this level.

When the chunk size is too big, the observed behaviour depends on the buffering scheme used, and several examples are depicted in Fig. 5.6. For a simple chunk size, the chunk follows the Bounce buffer plot of Fig. 5.2, plus the overhead of the chunk management. With the addition of a translator, an additional overhead from the cache misses can be seen as a gap between plots *Chunk-1M* and *Chunk-1M+TR*.

A double buffering scheme adds a cutting point with the size of the internal buffer. Up to this point, it behaves like a single chunk until it passes the buffer size, then it can take advantage of the additional buffer and the performance improves significantly. If the buffer is too big, this cutting point is already outside the cache of the processor and creates a valley. The cutting points can be clearly seen in the plots *DlbBuf-512K* and *DblBuf-1M* on Fig. 5.6. Then, for this particular system, the advantage of the double buffer lies in a range between 32K and 512K, even while the cutting point is still present in 128K and 256K curves (not shown) but is less critical.

The Pooled buffer tries to resolve this issue and provide a more sustained transfer rate for all transfer sizes. In particular, when the chunk size fits in the cache, a move/translation can be done very fast, but then the processor has to wait until the other buffer finishes the transfer to continue moving/translating data. The pooled buffer makes more buffers available to maintain the processor busy as long as possible. Therefore, the pooled buffer would only exhibit a cutting point when used with big chunk sizes. However, the management of pooled buffers adds some overhead, that may or may not justify the use of this scheme. It will depend on the balance between the transfer speed, the CPU speed, and the complexity of the translation being made.

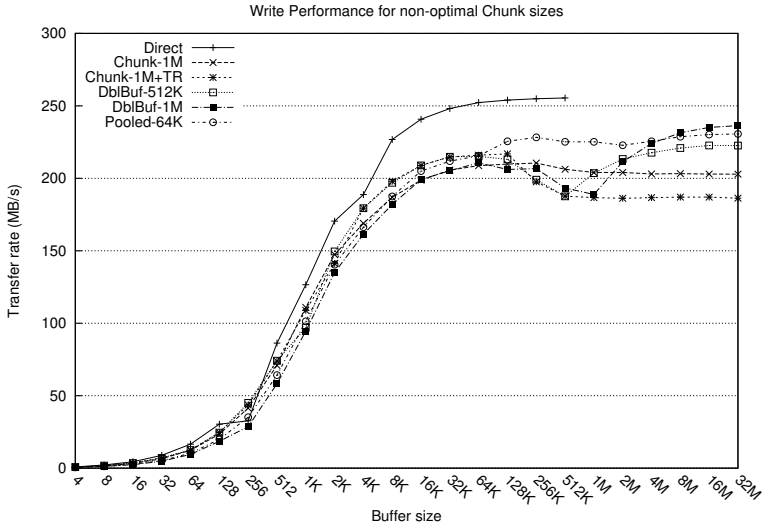


Figure 5.6: Non-optimal Chunk Sizes (write)

A Simple Case Comparison

As a simple case comparison, we performed a test that involves taking a buffer with data and doing the translation externally, comparing the transfer time for the different buffering schemes performing the same translation. The results are summarized in Tab. 5.3. The first two entries (ext-2cp and ext-1cp) are reference cases, performing translations externally, and they will use direct buffer when possible (512KB), otherwise they will use a chunk buffer (32MB). All chunks are of size 32KB, and the pool, when used, contains 32 buffers. 'ext-2cp' creates first a temporary array with the translated values, and then sends this array. 'ext-1cp' uses the direct buffer when possible as the target of the external translation, eliminating one copy. 'tr' uses a bounce buffer for 512KB and a chunk buffer for 32MB, and performs the translation internally. Therefore, 'tr' shows the advantage of using the translation in the loop. The advantages of using chunks, double buffering or pooled buffers have already been discussed and are confirmed by the results in this table, ranging from 23% to 82% improvements.

5.3.2 Performance of the Templatized Managers

A workstation with two Intel Xeon E5420 processors at 2.5 GHz, a 1066 MHz FSB and 6MB of L2 cache on a Intel 5400 chipset motherboard, 4GB of DDR2-667 RAM, and an AVNET V5LX110T Virtex-5 board and running Linux Debian was used for the tests. The Virtex-5 board uses a hardware PCIe Core from Xilinx paired with a custom made DMA Engine from Wenxue Gao[30]. The 4-lane PCIe interconnect is capable of 1 GB/s peak performance. The FPGA was loaded with a simple design that provides a memory region for IO access.

Similarly as with the previous tests, to measure time precisely during program execution, we utilize the Timestamp Counter (TSC) of the processor allowing the readout

	Write		Read	
	512KB (speedup)	32MB (speedup)	512KB (speedup)	32MB (speedup)
ext-2cp	4.231 ms -	252.7 ms -	3.886 ms -	266.4 ms -
ext-1cp	3.181 ms 38%	251.8 ms 0.3%	3.142 ms 23%	264.3 ms 0.7%
tr	3.182 ms 38%	186.0 ms 35%	3.055 ms 27%	214.3 ms 24%
chunk	2.987 ms 41%	185.4 ms 36%	3.031 ms 28%	215.0 ms 23%
double	2.321 ms 82%	145.4 ms 73%	2.593 ms 49%	181.7 ms 46%
pooled	2.587 ms 63%	148.5 ms 70%	2.626 ms 47%	180.4 ms 47%

Table 5.3: Simple Case Results

of clockticks under normal conditions (non-sleep modes). This provides nanosecond resolution for measurement of transfer times. Each transfer measure is repeated 10 times and the result is averaged to minimize spurious behaviour.

Performance without Translation

The DMA performance with templatized buffer managers is summarized in Figs. 5.7 and 5.8. Transfer functions are presented for the mprace library (a direct transfer) as well as chunk and double buffer managers. The first difference with the previous plots is the minimum data amount needed to saturate the channel. While the MPRACE-1 required ∼16 KB to saturate above 250 MB/s, this design requires ∼128 KB to saturate above 600 MB/s. Of course, the bandwidth to reach is higher in this case, and the plots concentrate on the differences above 4 KB transfers.

The next significant difference is the gap between the mprace results and the chunk buffers. The size of the gap represents an additional overhead, until the size of the chunk saturates the transfer into a constant plateau. This gap is reduced when compared with the double buffer performance, because the double buffer overlaps the transfer and copy operations. Also, because the double buffer reaches very closely the mprace performance for big data transfers, we can conclude that most of the copy operation time is hidden by this overlap. Therefore, the gap can be explained by the copy time needed to transfer the data in or out of the buffer, which is more significant in this case because of the higher bandwidth involved.

However, this is only true for big data transfers (>4 MB), and using relatively

big (1 MB) buffers. We are currently experimenting with a templatized pooled buffer manager, with the intention of reducing the performance penalty for smaller transfers.

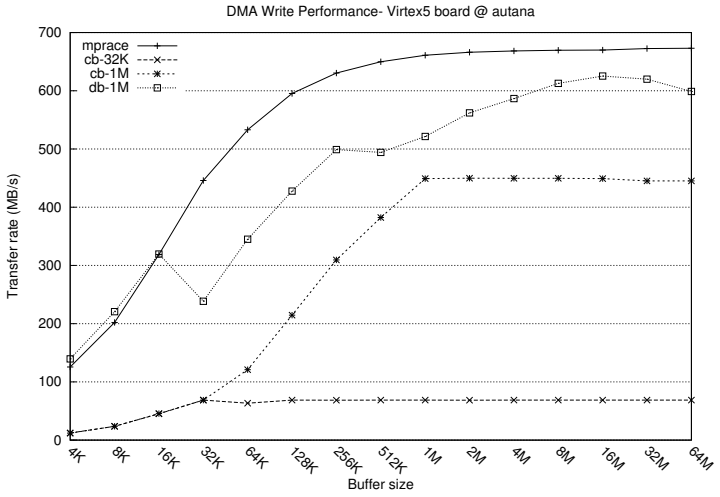


Figure 5.7: Write without translation

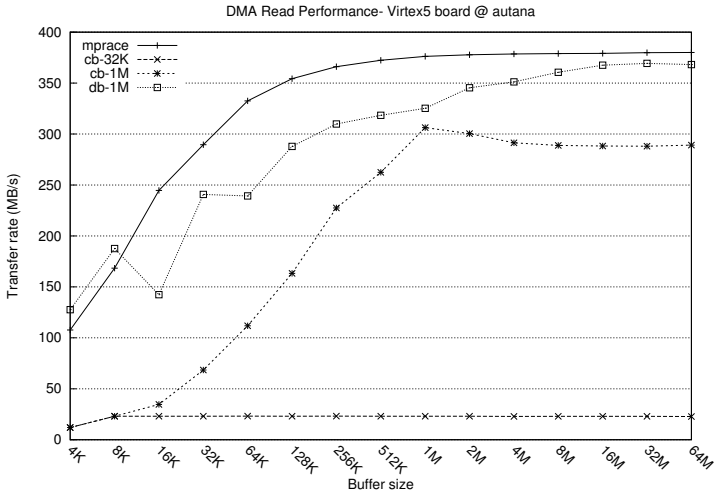
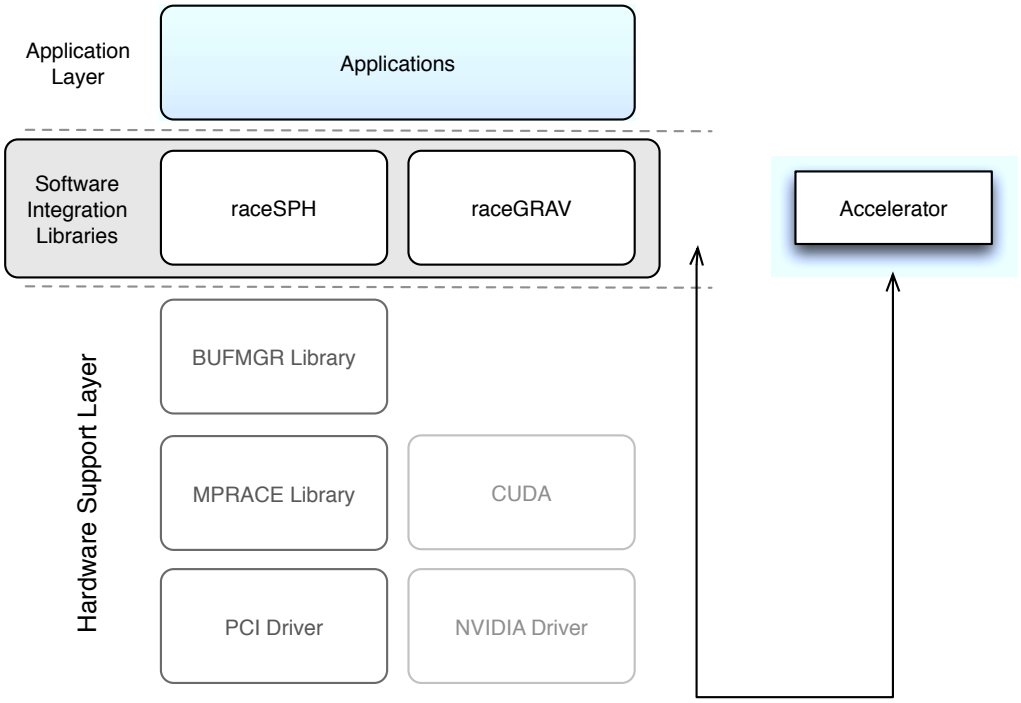


Figure 5.8: Read without translation

Part III

Software Integration



The following two chapters document the software integration libraries, raceSPH and raceGRAV, that abstract the operations needed to compute SPH and gravitational interactions into computing cores used by applications, and use the functionality developed on the previous chapters to interact with FPGA boards. In addition to analyse their results in synthetic benchmarks, they discuss their integration with applications provided by our colleagues at the ARI and the Munich Observatory.

Chapter 6

The raceSPH Library

Integrating a hardware accelerator into an existing high performance application can be a daunting task. First of all, the accelerator normally addresses a very defined part of the algorithm, which may or may not map directly to the implementation being used. Next, the data structures in the implementation typically do not match the data structures in the accelerator. Also, the application is normally optimized to run in a certain platform, and inserting an accelerator collides with this optimization.

In order to address these points efficiently, we created the raceSPH library. The library implements a generic interface to compute 2-step Smoothed Particle Hydrodynamics in its more generic form, following the equations described in 1.2 for the computation of density, pressure and acceleration, as well as entropy, the divergence and curl of the velocity. The equation of state is not computed by the library, but by the application between steps. This is done because the equation of state can vary greatly between one application and another, as it is independent from the SPH equations used.

This coarse separation in step1 and step2 might seem at first too simplistic, as it does not allow any modifications on the SPH model used. However, most of the astrophysical simulations we have encountered do not use other implementations. In particular, VINE[96], NBODY6++ and GADGET2[88] all use the original derivation from Gingold & Monaghan[31] that we implement in the accelerator. In addition, most differences arise from the treatment of boundary particles, or the formulation of the artificial viscosity in step2. However, boundary particles are rarely used in astrophysical simulations, with the notable exception of periodic boundary conditions[55], in which case they can be computed independently on the host, and depending on the formulation, also using the same accelerator. In this case, the application might modify the neighbour list accordingly, or use the accelerator in an independent stage to compute the contributions of boundary particles separately. Finally, different formulations of artificial viscosity can be added for special cases by modifying the processing pipeline of the accelerator to accommodate the new equation, a task that can be achieved easily by the Pipeline Generator used to build it. This pipeline is described in PPL language, and simple or moderate changes can be introduced without much knowledge of hardware design. For more details on the pipeline architecture of the accelerator or the Pipeline Generator, see the work by Lienhart[58, 60].

This high level representation also allow us to implement the library in multiple platforms. In this chapter we will cover a reference implementation, as well as specialized versions for SSE instructions, FPGA and GPUs.

6.1 Motivation

No discussion of this library would be complete without a review of its main motivation. In all astrophysical simulations with self-gravity, specially direct gravitational n-body simulations, the computation of these forces is by far the most time consuming task. As the time required to compute gravity scales with $O(n^2)$ for the number of particles in the system, it becomes dominant very quickly. Therefore, a lot of time and research was spent in ways to accelerate its computation, which eventually led to the development of specialized hardware accelerators like the family of GRAPE cards. These accelerators reduce the order of the gravity computations to $O(n)$ time by means of parallelizing the computations for a good range of the number of particles.

When using such accelerator for gravity with a simple code which also includes gas hydrodynamics with SPH, the gravity computation is no longer the most time consuming part. Fig. 6.1 shows the time fraction spent in the 3 most time consuming tasks for particle number between one thousand and one million. It can be seen that SPH consumes between 50 and 60 percent of the time, while gravity only consumes between 15 and 30 percent. The last component is the generation of the neighbour lists used by the SPH computation, which takes 20 to 30 percent.

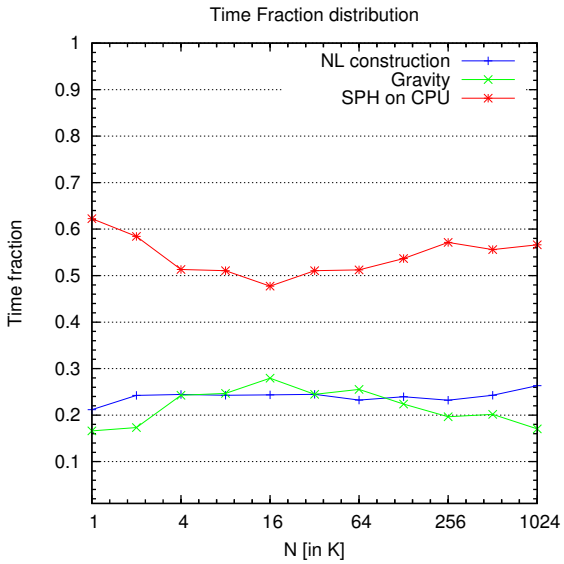


Figure 6.1: Time fraction distribution on CPU, computing gravity with an accelerator and SPH with the CPU. Plot by Peter Berczik, reproduced with permission

From this plot we can also conclude that if the SPH computations take 50-60%, the maximum speed-up over the whole execution time that can be achieved from ac-

celerating this part of the algorithm is in the order of 2x-2.5x speed-up, by Amdahl's law[10, 43]. The first question that comes to mind is, if it is worth the effort to implement an accelerator for such gain.

The answer comes from the application of Gustafson's law[32]. For a given problem size, gravity and SPH can be parallelized by dividing work among processors. The creation of neighbour lists can also be parallelized by following several constraints. Therefore, the serial part of the algorithm is minimal. As we distribute the application among P processors and α is the non-parallel fraction, the speed-up would be $S = P - \alpha(P-1) \approx P$, as α is said to be small. In consequence, we can profit from this speed-up in the local node when running the application in a cluster, as it will be compounded by the number of processors used, and for a given time, allows the simulation of a bigger problem size.

6.2 Previous and Related Work

There have been several developments on the acceleration of SPH, both inside and out of the astrophysical community. One of the first attempts has been the GRACESPH, which uses the capabilities of the neighbour list generation of the GRAPE boards to reduce the computational time of SPH in the host. Introduced by Steinmetz in 1996, GRAPESPH[92] does not compute the SPH forces in special hardware, but instead profits from the dedicated unit in the GRAPE boards to generate the interactions lists, which in turn are used to compute the SPH interactions in the host. While using very different CPUs than at present time, the time fractions presented for non-accelerated code, 50-55% for the force computation, are very similar to ours from Fig. 6.1. He was also one of the first to identify the readout of the neighbour lists from the GRAPE as a main bottleneck for this approach.

An improvement over the single node implementation by Steinmetz is shown by Nakasato et al. in 1997 in conjunction with the Remote-GRAPE[75], where they build over the previous results to conclude that the computation of SPH was the actual bottleneck in the system, proceeding to distribute it among several hosts by their newly introduced Remote-GRAPE interface and therefore improving its performance further.

A significant development in the computation of the SPH forces was the introduction in 2004 by Lienhart[58, 60] of the first SPH pipelines implemented in FPGA, together with the corresponding analysis necessary for the tuning of precision in the floating point operations without affecting the accuracy of the computations significantly.

Another related development is presented by Nakasato[74] et al. in 2007 with the SPH computation using PROGRAPE-3 systems, providing a similar development with the introduction of an automatic pipeline assembler and limited precision floating point units. An interesting variation is the use of multiple FPGAs in a single board (the PROGRAPE-3 has 4 dedicated to computation) for the calculation, and how they compare multiple task distribution schemes for the computation of gravity and SPH. Another important remark is the effect on performance of the data transfers, as their board deviated from the designed speed by a wide margin.

Parallel to these developments, other researchers focused in the acceleration of SPH

computations for other applications. Particularly, fluid dynamics simulations were quite active as for them, this is the most time consuming task in their programs. In addition, SPH is a method also used when creating high speed visualizations of fluids, like water or smoke in a game, as it provides a physical model which is approximate enough and easy to integrate into other particles systems for visualization. One important distinction of the following GPU implementations is that normally have much simpler integration schemes, without distinction of active or inactive particles in their systems.

Focusing in GPUs, one early work using GPUs is by Harada[37] et al. using a first-generation CUDA card for a full implementation of a simulation inside a GPU, that is, all tasks done by the GPU and no work left to perform by the CPU, showing speed-ups of up to 28 times for the computation.

Another important development was the work from van Kooten et al. presented in the GPU Gems 3 book[78], where they do not only compute the SPH forces but also provide a hash method to construct efficiently the neighbours by a grid of neighbouring cells. Unfortunately they work is focused in visualization, so the performance metrics used of frames per second are difficult to compare with our target interest.

More recently but focused in scientific applications of fluid dynamics, it is worth noting the work of Herault[42], Dalrymple[23, 21] and the team of GPU-SPHysics with the simulation of tsunamis and lava flows.

6.3 Formulae

Since some variations of the formulas are possible in the form of different kernel functions or artificial viscosity formulations, it is worth to list the actual implementation used. A W4 kernel function is used, together with the standard formulation of artificial viscosity as shown in 1.2.1. The kernel is normalized against the symmetrized smoothing length (see again 1.2.1), and the scalar part of the gradient denoted as $\Omega(x)$, so they both can be rewritten as follows

$$W'(x) = \begin{cases} 1 - \frac{3}{2}x^2 + \frac{3}{4}x^3 & \text{if } 0 \leq x < 1 \\ \frac{1}{4}(2-x)^3 & \text{if } 1 \leq x < 2 \\ 0 & \text{otherwise} \end{cases}$$

$$\Omega(x) = \begin{cases} -1 + \frac{3}{4}x^2 & \text{if } 0 \leq x < 1 \\ -\frac{1}{4}x + 1 + \frac{1}{x} & \text{if } 1 \leq x < 2 \\ 0 & \text{otherwise} \end{cases}$$

Note that some constants are moved out of the kernel in order to reduce the number of operations performed, and that in the following equations, the terms at the right side of the equation are computed by the accelerator. The density function from Eq. 1.15 is therefore computed in step 1 as:

$$\pi\rho_i = \sum_j \frac{m_j}{h_{ij}^3} W' \left(\frac{|\mathbf{r}_{ij}|}{h_{ij}} \right) \tag{6.1}$$

together with the curl and the divergence of the velocity:

$$-\frac{\pi}{3}\rho_i \left(\nabla \cdot \mathbf{v}\right)_i = \sum_j \frac{m_j}{h_{ij}^5} \left(\mathbf{v} \cdot \mathbf{r}\right) \Omega \left(\frac{|\mathbf{r}_{ij}|}{h_{ij}} \right) \tag{6.2}$$

$$-\frac{\pi}{3}\rho_i \left(\nabla \times \mathbf{v}\right)_i = \sum_j \frac{m_j}{h_{ij}^5} \left(\mathbf{v} \times \mathbf{r}\right) \Omega \left(\frac{|\mathbf{r}_{ij}|}{h_{ij}} \right) \tag{6.3}$$

Since the density is needed to compute the acceleration, this is done in step 2 as

$$\frac{\pi}{3}\frac{d\mathbf{v}_i}{dt} = -\sum_j \frac{m_j}{h_{ij}^2} \left(\frac{P_j}{\rho_j^2} + \frac{P_i}{\rho_i^2} + \Pi_{ij} \right) \Omega(\frac{|\mathbf{r}_{ij}|}{h_{ij}}) \tag{6.4}$$

together with the entropy and the $max(\mu)$ from the artificial viscosity, with the entropy (more properly, the irreversible thermal energy introduced by the artificial viscosity[96]) computed as

$$\frac{\pi}{3}\rho_i \frac{du_i}{dt} = \sum_j \frac{m_j}{h_{ij}^5} \Pi_{ij} \left(\mathbf{v} \cdot \mathbf{r}\right) \Omega \left(\frac{|\mathbf{r}_{ij}|}{h_{ij}} \right) \tag{6.5}$$

6.4 Architectural Overview

When we consider the racesph library at system level, it is the uppermost component of our software stack, and the sole element that interacts with the astrophysical simulation. It is therefore the interface that the user sees, hiding the complexity of the different elements and libraries below, namely the PCI driver, the mprace library, the buffer management library as well as the accelerator hardware itself.

In Fig. 6.2 we can see an overview of the classes of the library, with a focus in the `SPHCore` hierarchy. The subclasses are grouped according to the accelerator used: Software Cores, FPGA Cores and GPU Cores. In addition, we group together a set of *Profiling Cores*, which are used for debugging, logging and profiling purposes. Also shown are the two interface classes to use the library with C code, as well as the Fortran library for specific use with the VINE application. Data types and Translator classes are represented here only as a package. Each accelerator core will be covered in detail in their own section in this chapter, while the data structures and supporting features will be described later in this section.

The interface of the base class, `SPHCore`, defines the interface that has to be implemented by all subclasses, which is also the interface used by the applications. It is more clearly summarized in Table 6.1, where functions are grouped by task, so we have functions for Step 1, Step 2, as well as control functions and Translator setter methods. This interface is inspired primarily by the FPGA accelerator interface, because it is the more restrictive of them, as well as one of the first implemented. After several

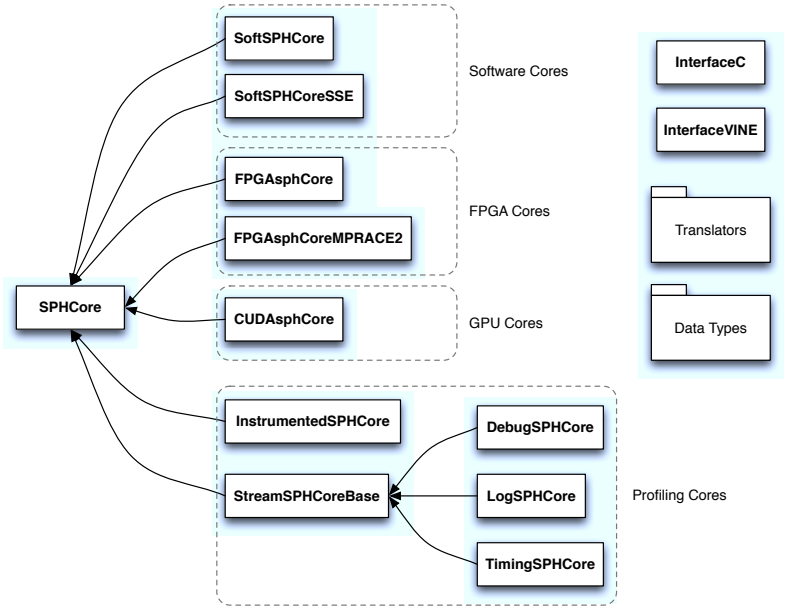


Figure 6.2: RACESPH library class diagram

iterations, we found this set of functions to be flexible enough to cover the use cases presented by the applications under study without sacrificing performance, and to be adaptable to a large set of platforms. We can now focus our attention in two main aspects: the data formats and the use case scenario.

The data formats are summarized in Fig. 6.3. While the particle data is unique and can be visualized as a simple array of particles, three structures (`s1_particle`, `s2_particle` and `mhc_particle`) are used to represent the particle data uploaded to the accelerator at different stages. The particle data is organized in three arrays, were each array if of one data type. The arrays are synchronized, so any index i refers to the data of particle i in all three arrays. Each structure does not represent data from a different particle, but data for the same particle to be uploaded at a different stage, so the data of particle i is uploaded using 3 different data structures. Consequently, Step 1 requires the data from `s1_particle` only, while Step 2 requires the data from both `s1_particle` and `s2_particle`. However, the smoothing length h and the speed of sound c change between Step 1 and Step 2 because of the equation of state. The structure `mhc_particle` is used to update this data without the need to send the full set of data of `s1_particle`, saving communication time. Finally, from the computations of each step, one array of type `s1_result` (from Step 1) or `s2_result` (from Step 2) is returned.

The final data structures are related to the communication of the neighbour lists to the accelerator. The neighbour lists are passed as a very long unidimensional array (`NL array`), with one neighbour list after the other. The `cutpoints` array contains the offset were each neighbour list starts, serving as a pointer or *shortcut* to a certain list. Every neighbour list is a list of indexes in the particle array. The indexes of

Step 1	
`s1Prepare`	Prepare Accelerator to compute Step 1
`s1SetParticles`	Write Step 1 Particles into the accelerator
`s1Calculate`	Calculate Step 1 and Read Results
Step 2	
`s2Prepare`	Prepare Accelerator to compute Step 2
`s2SetParameters`	Set Parameters required for Step 2
`s2UpdateMHC`	Update m, h, c in particles
`s2SetParticles`	Write Step 2 Particles into the accelerator
`s2Calculate`	Calculate Step 2 and Read Results
Control Functions	
`getStep`	Get the current processing step
`setFlag`	Set a Flag in the Accelerator
`getFlag`	Get a Flag value from the Accelerator
`setRegister`	Set a Register in the Accelerator
`getRegister`	Get a Register value from the Accelerator
Translators	
`setParticle1Translator`	Set the Particle 1 Translator
`setParticle2Translator`	Set the Particle 2 Translator
`setMHCTranslator`	Set the MHC Translator
`setResult1Translator`	Set the Result from Step 1 Translator
`setResult2Translator`	Set the Result from Step 2 Translator

Table 6.1: SPHCore class interface

K i-particles are listed first, were K is the number of concurrent i-particles that can be computed with the same neighbour list, and a fixed number that depends on the accelerator. Typically $K = 1$, but it can be higher if more than one i-particle share the same neighbour list, as in a shared-list scheme. Next in the neighbour list comes the number of neighbours N, followed by N indexes for the corresponding j-particles. N can be fixed or variable, depending on the method used to construct the neighbour lists: fixed neighbours N but variable smoothing length h, fixed smoothing length h and variable number of neighbours, or both parameters varying.

To better understand the relationship between the interface and the data structures, it is best to explore the use cases scenarios for it. The first scenario to consider is the processing of a single time-step with all particles active (see 1.4 for a discussion of active/inactive particles). This process involves a sequence of operations started by the application and directed to a *SPH Core* object, shown in Fig. 6.4 and whose can be described as follows:

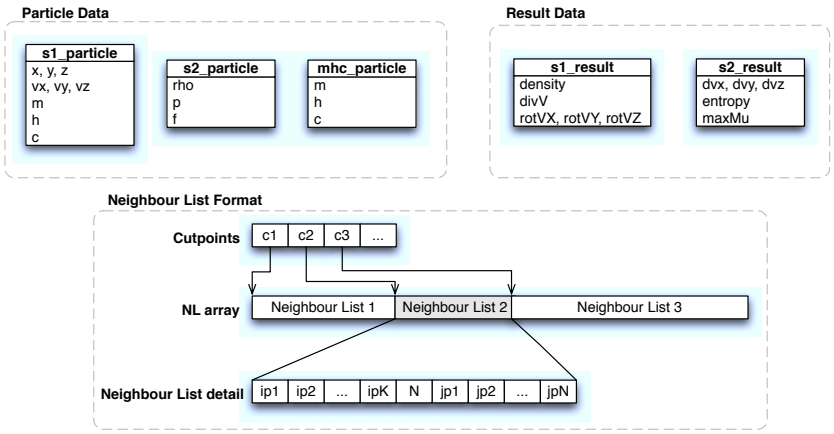


Figure 6.3: RACESPH data structures

1. **Prepare Step 1**. This must be done before doing any other operation which involves Step 1. It is used to prepare the accelerator for this task, like setting flags changing the mode of operation, cleaning temporary data, or loading a new design into an FPGA.

2. **Load Step 1 Particles**. This step receives from the application the data needed to compute Step 1 and loads it into the accelerator. This operation is also responsible of any data transformation of particle data. By means of an optional translator object and the `bufmgr` library, the `racesph` library is capable of including a conversion step efficiently into this transfer. In this way, the data passed to this operation can be in the native format of the application, but it is loaded into the accelerator by whatever data structure it requires natively (typically an array of `s1_particle` elements).

3. **Calculate Step 1**. This is the operation that computes Step 1 and returns the results to the application. It receives the neighbour lists structures and passes it to the accelerator core for processing. The accelerator returns an array of `s1_result` elements, which is again optionally transformed by a translator object into the desired data format of the application.

4. **Compute Equation of State**. This is done entirely by the application, and it is a required operation between Step 1 and Step 2 in order to update the sound of speed and possibly the smoothing length of the particles being computed.

5. **Prepare Step 2**. Similar to Prepare Step 1, this operation prepares the accelerator to perform operations related to Step 2.

6. **Set Parameters**. This operation loads dataset-wide parameters into the accelerator. Currently, it sets the constants *alpha*, *beta* and *eta* required for the computation of artificial viscosity.

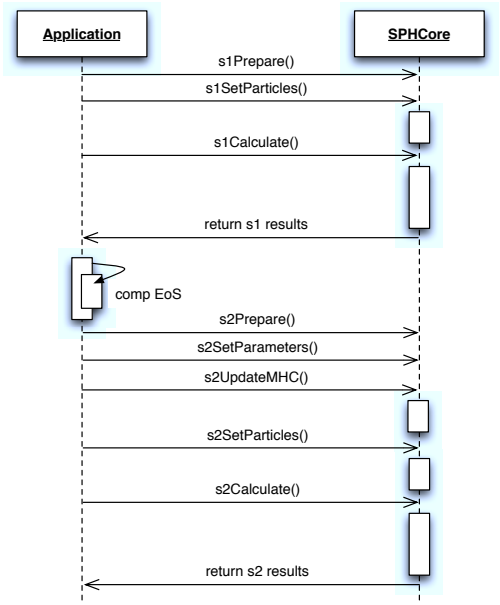


Figure 6.4: SPH usage

7. **Update MHC Data**. This operation uploads to the accelerator updated values for mass, smoothing length and sound of speed. New values of the smoothing length and sound of speed were computed by the EoS and need to be uploaded. Similarly to other data uploads, this can use an optional data translator to match the need of the accelerator. In this data set, only the sound of speed has a significant change on every step as a consequence of the EoS. The smoothing length can in theory change, but in practice is much more dependent on the neighbour configuration and therefore slow changing, as is sometimes reused by as much as 10 time-steps. The mass doesn't need to be updated, but is present here because a data dependency on the FPGA design: these three values are in the same data line in the FPGA board memory, so writing the data line requires to update all 3 values. Reading the data back from the FPGA would cut memory performance in half and affect the overall performance significantly, so we chose to read it back from the host as a better alternative.

8. **Load Step 2 Particles**. As with Load Step 1 Particles, this operation loads the additional particle data needed for Step 2 into the accelerator. It does not reloads the Step 1 data (that is done by Load Step 1 Particles and Update MHC data), but it uploads the data corresponding to `s2_particle`s. Similarly, a translator can be used for data format conversion.

9. **Calculate Step 2**. Computes the Step 2 and returns the results to the application. It receives the neighbour lists structures again, and passes them to the accelerator. Since the neighbour lists is the same for both steps, the *SPH Core* in use has the option to cache the neighbour lists and ignore the input, saving

communication time. The accelerator returns an array of `s2_result` elements which can be transformed by a translator into the data format of the application.

This sequence describes the sequence that all applications execute to use a SPH Core of the `racesph` library, and all use cases are variations over this main sequence. The first variation to consider is when not all particles are active in the current timestep. From the neighbour list point of view, it means that only the neighbour lists where these active particles are i-particles need to be computed. In terms of the particle data needed to compute these neighbour lists, it is composed of the active particles (the i-particles) plus all neighbour particles in the lists (the combined j-particles). While the list of i-particles is very easy to obtain, the list of j-particles is a more complicated subject, as it involves walking all active neighbour lists and selecting the j-particles. Then, the combined set of i- and j-particles is loaded into the accelerator.

However, the upload of a subset of the particles involves a gather operation, as the particles are distributed across the memory and must be collected for transfer. In addition, since the particle index inside the accelerator is important (the neighbour lists use particle indexes), the gather operation will change the indexes, so the neighbour lists will in turn need to be rewritten with these new indexes. Furthermore, the results that come back from the accelerator need also to be scattered in memory following the same mapping in the opposite direction.

The process just described consumes too much time, relative to the actual computational time. As some time-steps have very few active particles, the problem represent an even bigger overhead in these cases. Therefore, we opt for a different approach: we use the accelerator when only a minimum number of particles is present, and we then load the whole particle data set to the accelerator, which is faster than loading several small data sets of gathered data, and send only the neighbour lists for the active particles for computation, effectively ignoring the non-used particles.

Handling boundary particles presents a special case. While astrophysical simulations do not usually have boundaries, for the applications that do require them like a fluid in a box for a dam break simulation, the use of the accelerator depends on the boundary model used. There are many ways to handle them, but models like repulsive force or negative mass can be used and handled by the accelerator as any other particle. Other models that involve computing special interactions with regular particles must be handled by the CPU.

6.5 CPU and SSE implementations

The CPU and Streaming SIMD Extensions (SSE) classes cover the implementations to be run exclusively in the host. The CPU implementation serves several purposes: it provides a reference implementation, that users wanting to integrate with an accelerator can use to test the functionality of the library and its interaction with the application without the need to access any special hardware. Once the integration with the application is done, the CPU core also provides the capability to test the effects of using single precision floating point operations for the SPH computations, instead of the double precision floating point more commonly used. Finally, being very straightforward

code (i.e. without optimizations), it allows developers to test the effect of changes in the computations before implementing them in an accelerator.

The regular CPU version consist of a loop that processes sequentially every neighbour list passed to the core, and for every neighbour list, it computes sequentially each interaction. Every interaction accumulates the result for that particular neighbour list and returns it as the result for that i-particle when the neighbour list has been processed fully. This loop is repeated for both Step 1 and Step 2.

The SSE version uses the streaming SIMD instructions available in most modern processors to compute the SPH interactions. These instructions operate over 128-bit registers and allow a single CPU core to execute up to 4 single precision operations concurrently. The functional units share registers with the regular floating-point pipeline, and depending on the architecture, it has available 8 FP registers and 8 128-bit (XMM) registers in 32-bits architectures, or 16 FP and 16 XMM registers in 64-bits architectures[48]. Under proper conditions, SSE instructions enable the application to use the peak performance of the CPU, being the two more important constraints that enough data is available, and that all functional units are used during a SIMD instruction.

Because of the low number of registers available and the SIMD nature of the instructions, the performance is very sensitive to data dependencies and starvation. Data dependencies come as operations that require a previous pending operation to complete before it can be executed, while starvation (which can be considered a consequence of a not fulfilled data dependency) involves data that is not accessible when needed or accessing memory that is not present in the cache, requiring a delay until it becomes accessible. For a more in depth analysis of data dependencies on performance, see [41].

These constraints require careful crafting of an SSE program to keep the functional units as busy as possible. From a high level language like C++, there are several ways to use SSE instructions in a program, which lead to several combinations of optimizations being available. One approach is to embed assembly code directly into the program; to use intrinsic functions to guide the compiler very closely in the SSE operations to perform; or to use vectorizing compilers to convert the code automatically from a serial description into SSE implementations.

The first, and most straightforward, is to assembly code directly into the program. This can be done by some compilers like `gcc` with special directives, or by linking an external function which is been assembled independently. The biggest advantage of this approach is that the programmer has complete control of the code: the instructions, registers and algorithms being used all need to be specified by the programmer. Because of the same reason, this solution becomes quite inflexible: any change requires reworking the optimizations. Techniques like register reuse, instruction reordering, function inlining and loop unrolling have to be implemented manually by the programmer, and as the number of registers changes between 32- and 64-bit platforms, separate versions of the code are required to cope with the optimizations for each case.

The second method is to use SSE intrinsic functions. SSE intrinsic functions are function wrappers for SSE instructions, where each instruction is paired with a function, so each instruction is seems as a regular C function by the language. Specific data types are used to ensure the correct data alignment and register association of

the instruction and the memory. Intrinsic functions rely on the compiler for register allocation, instruction reordering, inlining and loop unrolling, while leaving to the programmer the vectorization of the algorithm and the choice of actual instructions to execute. Because register allocation is done by the compiler, the same program can be compiled in 32- and 64-bits platforms and the best register allocation will be used, up to the capabilities of the compiler and the platform. Another advantage is that intrinsic functions are fully supported by the two main compilers for the x86 platform: GNU `gcc` and Intel C Compiler `icc`. The main disadvantage is that the compiler might not be as effective in optimizing the code as a skilled programmer, but that is true for any compiler.

The last method we mentioned earlier, using a vectorizing compiler, gives full control of the process to the compiler. By crafting code in predefined patterns (typically as loops), one can guide a capable compiler into converting code into SSE equivalent instructions. In this case, the successful conversion is very limited by the capability of the compiler to recognize useful fragments and parallelize the code. Both the `gcc` and `icc` have parallelizing capabilities, and be will discuss them in the next paragraphs.

On a first instance, we tested the vectorization capabilities of the Intel (`icc` version 8) and GNU compilers (`gcc` versions 3.3 and 4.2), and evaluated the generated code using Intel VTune Performance Analyzer 8, a profiling tool with code inspection capabilities. Our first conclusion was that `gcc` 3.3 was able to parallelize only the simplest loops, equivalent to reductions, vector additions, and such. We also found that it could not perform any significant optimization when dealing with intrinsic functions, performing only straightforward unwrapping and no significant register reuse. Since the SPH computing loops contain long computing sequences, we concluded it was not suitable for our complex pipeline, and we concentrated in `gcc` 4.2 and `icc` 8.

However, both `gcc` 4.2 and `icc` 8 had limitations processing the SPH pipeline. The first issue is that the double loop (all neighbour lists, all interactions per list) is divided into 3 function calls, and the innermost call, a loop to compute the interaction of one i-particle with its neighbour list, consists of a very long inner iteration. This means the compiler cannot identify the loop as a candidate for automatic SSE optimization, because of its length and complexity. Dividing them in smaller tasks provide little help, as a lot of code is repeated on each task, while several divergent branches are still present. These divergent branches are the consequence of the SPH kernel being a piecewise function (see Sec. 6.3). Because of these drawbacks, it was finally decided to not try to use the vectorization capabilities of the compilers, and we focused in using intrinsic functions.

The SSE intrinsic functions provided the best balance between flexibility and performance. The data model to follow is that of each vector unit in a SSE instruction acts as a single thread, so 4 interactions can be computed in parallel, provided they share the same i-particle, which means they are part of the same neighbour list. Individual contributions are then sequentially added to the final result for the active particle. The first task we did was to create a couple of custom data types, shown in 6.1, to pack and unpack float and integer values easily. These structures provide data alignment and easy access to data in XMM format, as well as easy access for the rest of the code.

Afterwards, the original loop functions were modified in order to perform particle

Listing 6.1: XMM union data types

```
union v4 {
                __m128 xmm;
                float v[4];
        } __attribute__ ((aligned(16)));

union i4 {
                __m128i xmm;
                int i[4];
        } __attribute__ ((aligned(16)));
```

data fetching and packing at the start of each iteration, to comply with XMM data formats. Then, the actual interaction was ported to be computed with SSE instructions, 4 interactions in parallel. In order to reduce the expensive cost of branching for special cases when less than 4 interactions are left in the list, we use dummy particles that provide no contribution to the final result. This makes the code more readable and only requires a single branch at the end of the neighbour list to perform a partial data fetch.

As each vector unit is computing an independent interaction in parallel, it is possible for the SPH kernel function to diverge, that is, to take different values on each interaction. In order to handle this efficiently, we always compute all 3 options, and use an integer mask to select which contribution to use. This is best shown with an example. Let's assume a function $y(x)$ with 3 possible cases, as shown in Eq. 6.6.

$$y(x) = \begin{cases} A & \text{for } x \leq k_1 \\ B & \text{for } k_1 < x \leq k_2 \\ C & \text{for } x > k_2 \end{cases} \tag{6.6}$$

We can present a solution as in Fig. 6.5. Each case can be computed independently, and we can call the partial results `y1`, `y2`, `y3`. Then, each condition of `x` is evaluated, and a mask produced for each case. The mask is used to select and combine the partial results with the final result `y`, as shown in the figure.

The actual code for the SPH kernel is shown in Listing 6.2. The main difference with the example is that both the kernel and its derivative are selected and combined using the same mask. At the end of the interaction loop, the results are accumulated, one partial result per vector unit. When the loop is finished, the partials are combined into a single result. This is done in two possible ways: with a small loop with normal FP operations, or using an horizontal add, an instruction available in the SSSE3 instruction set only. The horizontal add looks more elegant, but in practice both approaches give very similar results, with less than 3% difference in out tests.

The computation times and speed-up are summarized in Fig. 6.6. Results shown are for a synthetic benchmark executed in two 64-bits machines: a 3.2 GHz Xeon (mp-pc109) and a 2.5 GHz Xeon E5420 (autana). Important to note in the computation time plot is the nearly linear scaling, from 100 up to 200,000 particles. This linearity tell us that there are no extraneous effects like cache sizes or memory barriers affecting

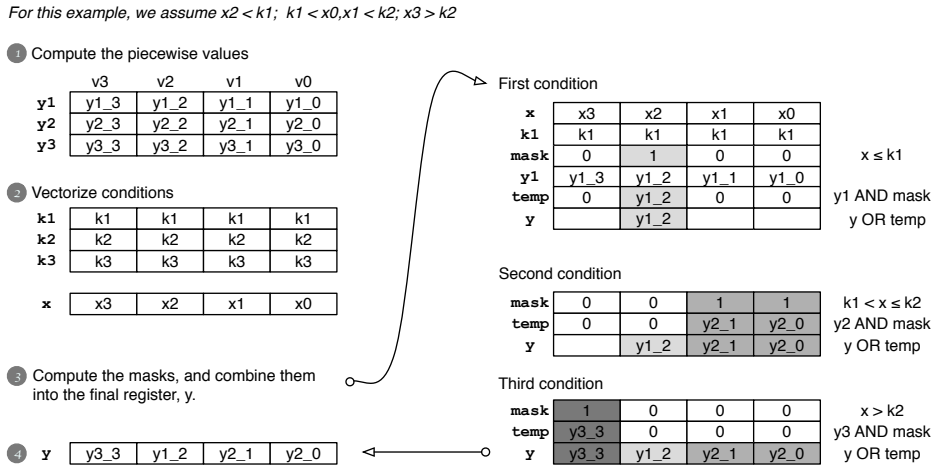


Figure 6.5: Example for the computation of a piecewise function with SSE instructions

the performance. From the speed-up plot, we only see a small effect for very small data sets (100 particles), and smaller variations as the number of particles increases. Both machines show similar speed-ups averaging 2.1x-2.3x, which is somehow deviated from the expected maximum of 4x. The main difference can be attributed to the newer architecture and bigger caches of autana, which makes the machine faster even when the clock speed is lower. Additional code is also needed to prefetch data and convert data formats, which represent a linear overhead that scales with the number of particles.

6.6 FPGA implementation

As mention earlier, to provide the FPGA implementation with a usable framework was the initial and main goal of the raceSPH library. While CPU or GPU implementations can be more or less directly integrated with actual code, the additional requirements for an FPGA board would make it very difficult, in particular the interactions with the PCI driver, the `uelib` or `mprace` libraries, and the `bufmgr` library, in addition to the actual code required for data processing.

In order to understand the design choices for the FPGA implementation, an overview of the FPGA coprocessor is needed. First of all, the FPGA coprocessor is an FPGA board that communicates with the host system via a high speed bus. The board itself contains a bridge which is in charge of the communications with the host, and a main FPGA dedicated to the computations. Additional memory connected to the main FPGA is available for the coprocessor to use. The main characteristics of the coprocessor boards used are summarized in Tab. 6.2.

The main FPGA is, for this application, loaded with a design that computes SPH interactions and in order to better understand the work done by the `FPGACore` class, we will do a short overview of its functionality. The design follows the internal structure as shown in Fig. 6.7. The modules shown represent actual blocks in the design with very defined tasks, as follows: The I/O block controls the data flow to and from the

Listing 6.2: Computing the SPH kernel with SSE instructions

```
// case 1: x <= 1.0
//smooting kernel
x2.xmm = _mm_mul_ps( x.xmm, x.xmm );
W_c1.xmm = _mm_mul_ps( threeFourth.xmm, x.xmm );
W_c1.xmm = _mm_sub_ps( threeHalf.xmm, W_c1.xmm );
W_c1.xmm = _mm_mul_ps( x2.xmm, W_c1.xmm );
W_c1.xmm = _mm_sub_ps( one.xmm, W_c1.xmm );

//derivative of smoothing kernel
O_c1.xmm = _mm_mul_ps( threeFourth.xmm, x.xmm );
O_c1.xmm = _mm_add_ps( minusOne.xmm, O_c1.xmm );

// case 2: 1 < x <= 2.0
//smooting kernel
y.xmm = _mm_sub_ps( two.xmm, x.xmm );
W_c2.xmm = _mm_mul_ps( y.xmm, y.xmm );
W_c2.xmm = _mm_mul_ps( W_c2.xmm, y.xmm );
W_c2.xmm = _mm_mul_ps( W_c2.xmm, oneFourth.xmm );

//derivative of smoothing kernel
temp.xmm = _mm_div_ps( one.xmm, x.xmm );
O_c2.xmm = _mm_mul_ps( minusOneFourth.xmm, x.xmm );
O_c2.xmm = _mm_add_ps( O_c2.xmm, one.xmm );
O_c2.xmm = _mm_sub_ps( O_c2.xmm, temp.xmm );

// case 3: x > 2.0
//smooting kernel
W_c3.xmm = zero.xmm;
//derivative of smoothing kernel
O_c3.xmm = zero.xmm;

// Select proper case for each particle
// x <= 1.0
mask.xmm = _mm_cmple_ps( x.xmm, one.xmm );
W.xmm = _mm_and_ps( W_c1.xmm, mask.xmm );
O.xmm = _mm_and_ps( O_c1.xmm, mask.xmm );

// 1 < x <= 2.0
temp.xmm = _mm_cmple_ps( x.xmm, two.xmm );
mask.xmm = _mm_xor_ps( mask.xmm, temp.xmm );
temp.xmm = _mm_and_ps( W_c2.xmm, mask.xmm );
W.xmm = _mm_or_ps( W.xmm, temp.xmm );
temp.xmm = _mm_and_ps( O_c2.xmm, mask.xmm );
O.xmm = _mm_or_ps( O.xmm, temp.xmm );

// x > 2.0
mask.xmm = _mm_cmpgt_ps( x.xmm, two.xmm );
temp.xmm = _mm_and_ps( W_c3.xmm, mask.xmm );
W.xmm = _mm_or_ps( W.xmm, temp.xmm );
temp.xmm = _mm_and_ps( O_c3.xmm, mask.xmm );
O.xmm = _mm_or_ps( O.xmm, temp.xmm );
```

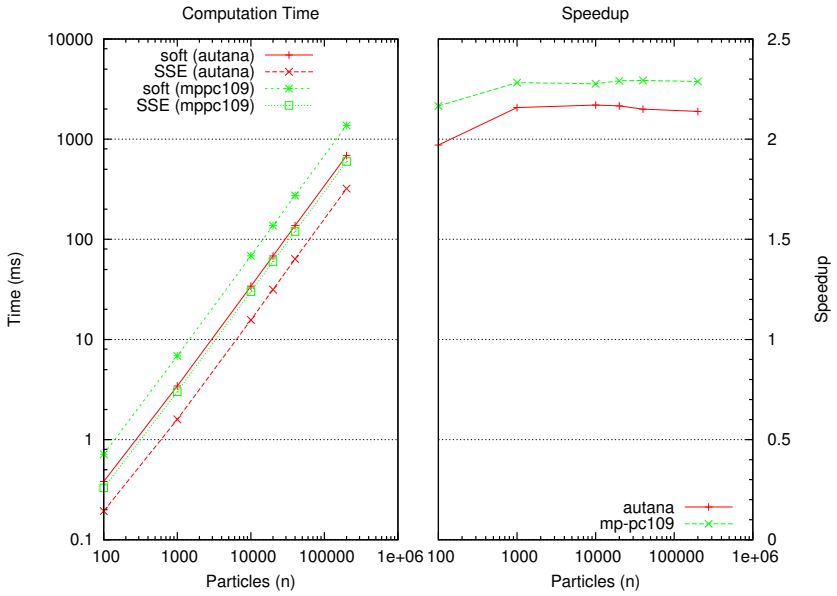


Figure 6.6: Computation Time and Speed-up for Software and SSE cores

bridge chip. The Memory Controller handles the signal protocol and data organization in the external memory. The SPH pipeline computes the interactions, and the SPH control handles the control signals. The Result FIFO hold the results produced by the SPH pipeline until the host is ready to fetch them.

The design handles streams of data, both as inputs and outputs, in different steps detailed in Section 6.4. Every stream has it own address in the Address Map of the design, so the decoding is very simple. When data is being uploaded, data is written to the `PARTICLEDATA` address, and the data stream is routed to the memory controller for upload. Similarly, when computing interactions, a stream of neighbour lists is received from IO at `CALCULATEDATA`. This stream is routed to the control unit and the pipeline, and particle data is fetched from memory according to the interaction list for computation by the pipeline, and results are pushed into the Result FIFO when ready. Afterwards, these results are fetched by the host when reading from the `RESULTDATA` address.

Additionally, the memory controller performs data width conversion. This is needed as data is received as single precision floating point, but stored in memory (and used by the pipeline) as limited precision. The actual precision used for each value was determined by Lienhart and Wetzstein, and documented at Lienhart's PhD dissertation[58] for the first implementation of the SPH pipeline, in order to not affect the result of the SPH computation significantly.

Another reason for this conversion is to be able to store more particles in the limited memory of the FPGA. As the memory banks are organized as a very wide array of 144 bits (18 bytes), the reduction in precision corresponds to less bits used, which in turn allows to store the data for each particle in only two memory lines (36 bytes). This has

	MPRACE-1	MPRACE-2
Bridge		
Part	PLX9656	Virtex4-FX20
Manufacturer	PLX Technology	Xilinx
Host Link	PCI-X	PCI Express x4
Max Bandwidth	264 MB/s	1 GB/s
Bridge-Main Link		
Link Type	Local Bus	Serial Links
Link Size	32-bits/64 MHz	4 links @ 1.25 GHz
Link Bandwidth	256 MB/s	1 GB/s
Main FPGA		
Part	Virtex2-3000	Virtex4-FX60
Max Clock Source	125 MHz	250 MHz
Memory Banks		
Banks	4	2
Type	ZBT-SRAM	DDR2-SRAM
Arrangement	512k*36 bits	1M*72 bits
Total	9 MB	18 MB
SO-DIMM Module	SDRAM	DDR2
SO-DIMM Max Size	256 MB	1 GB

Table 6.2: FPGA board specifications

two main consequences: the computation between particle index and memory address is very simple; and the maximum data rate for input to the pipeline is limited to half of the data rate of the memory.

For the MPRACE-1, this is 125 MHz for the memory and 62,5 MHz for the pipeline input. This means that, because the pipeline can compute one interaction per cycle, it needs a constant input of one particle per cycle. The i-particle data is fixed in registers and do not consume extra bandwidth during the computation of an interaction. Then, the particle data stream effectively limits the performance of the design on the MPRACE-1. However, this is not a huge loss, as the neighbour list comes as a stream from the *Local Bus* at a rate of 64 MHz, and the *SPH control* sends the particle index request to the *Memory Controller* at the same rate. It follows that additional improvements in particle fetch will give only a minimal improvement on the performance of the design *for a single pipeline*. It could lead to an improvement with more pipelines available, but there is not enough space on the current device for it. The design can therefore be considered balanced on its performance constraints.

It can also be established that the performance of the SPH design on the FPGA is determined by the speed at which the particle data is loaded, and more importantly for big data sets, the speed at which the neighbour list is transferred to the board. The computation time by the FPGA is then proportional to $T_{load}(n) + T_{nl}(n_a m)$, where n is the number of particles in the system, n_a the number of active particles, and m the

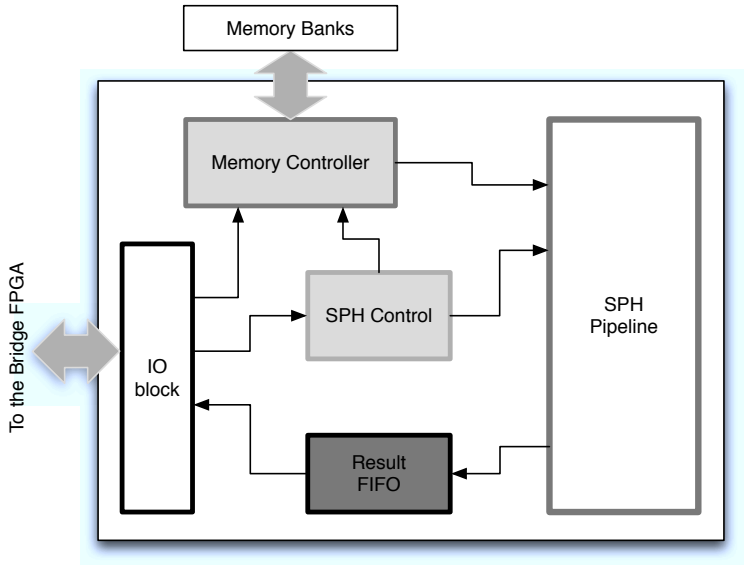


Figure 6.7: FPGA SPH design block diagram

average size of the neighbour lists. The fetching of results should also be a factor, but is proportional to $T_{res}(n_a)$ and therefore not significant for most cases.

With a clearer view of the requirements of the FPGA design, we can now set goals for the `FPGACore` class implementation. We can highlight three key operations that need to done as fast as possible: load particle data, send neighbour lists and fetch results. Loading particle data occurs only once per computed timestep of the simulation (three times if we consider step 1, step 2 and updateMHC separate instances), and it is related linearly with the number of particles. To send the neighbour lists is to take the data structure received by the function and send it to the FPGA, as the data structure provide can be interpreted by the FPGA design directly. However, it scales with the number of active particles and the number of average neighbours as $O(mn)$. To fetch results involves reading the data from the Result FIFO and transferring it to the appropriate structures in the application (it scales directly with the number of active particles).

It is important to note that both operations, load particles and fetch results, may require data transformations between the FPGA data format and the application data format. It is with this purpose that the `bufmgr` library was developed, as it provides a very flexible interface to add a high performance transfer between the host and the FPGA board. As detailed in Chapter 5, the library handles the data conversion into the same copy loop necessary for the DMA transfer. In addition, it provides the racesph with a simplified mechanism to deal with the change of FPGA library, being both the `MPRACE` and the `uelib` libraries handled by the `bufmgr`.

The sending of the neighbour lists has a couple of special considerations. First, because the data format recognized by the FPGA is the same as the input format, no additional data transformation is needed. Second, the Result FIFO in the FPGA can

hold only a very limited amount of data (in the order of 4 KB for the MPRACE-1). Therefore, in order to process a full set of lists we need to regularly empty the FIFO to avoid it to overflow. The overflow can be limited by sending only the appropriate number of lists to the FPGA, therefore producing a known number of results.

Then, a loop is implemented to send a subset of the neighbour lists and fetch the corresponding results on each iteration. This division on segments adds another restriction, as we cannot send partial lists (that is, a list is sent with all its neighbours or not sent at all), because it is not supported by the design. For this to be done efficiently, the library uses the `cutpoints` array (see Fig. 6.3). This array allows the library quick access to lists in the `nl` array, so we can select the subsection of the `nl` array easily. Afterwards we use a separate buffer manager to send the selected section to the FPGA, wait for results to be available, and fetch them with another buffer manager, in order to handle the data transformation. This is repeated until all lists in the array are processed.

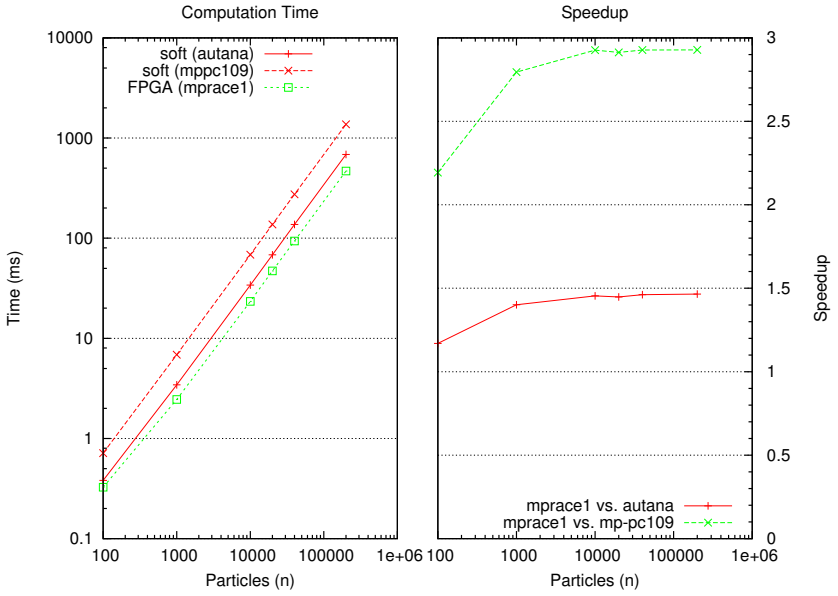


Figure 6.8: Computation Time and Speedup for FPGA core on MPRACE-1

Performance figures for the MPRACE-1 are shown in Fig. 6.8. Execution times for the FPGA include both calculation and particle load time, in order to have a fair comparison with the CPU reference code. The synthetic benchmark used for the tests simulates the computation of an all active particles timestep, with the ratio between load and computation time being fairly regular with a ratio of 0.1 for step 1 (load time 10% of the computation time) and a ratio of 0.05 for step 2. From the plots, we can see that the performance of the FPGA design is about 3x faster than the reference in mp-pc109, and about 1.5x faster than autana. Some overhead is visible for the range 10-1000 particles, being compensated by the coprocessor working at full performance for the range 10k-200k particles. The effects on actual applications will be documented

in following sections (6.8 and 6.9), after the GPU cores are explained.

6.7 GPU implementation

The `GPU Core` implementation is intended to profit from the highly parallel architecture of modern graphics coprocessors. As described in Ch. 2.2, newer architectures are fully programmable and have hundreds of floating point units, paired with very fast-access memory. This implementation targets NVIDIA GPUs, as it uses the CUDA programming language. This restricts it to a single provider, as CUDA is only available for NVIDIA-powered cards, but other free, multi-platform options were not available at the time of initial development. In order to bridge the part of the `GPU Core` written on C++ to the CUDA specific code, a separate `cudasph` library is created, containing all CUDA specific code and exporting regular C functions. This library is then linked to the `GPU Core` class in the `racesph` as regular functions. This division allows, among other things, to create a clear cut defining where CUDA specifics are allowed and where they are not, as well as operations related to the library architecture (like data transformation) or the GPU architecture (like CUDA contexts to hold device specific pointers).

A GPU card has several similarities with a FPGA board, among the obvious differences. Like the FPGA accelerators, a GPU is an add-on card that interfaces with the host over a bus (typically a 16-lane PCIe bus). This means that data must be copied to and from the card, similar to an FPGA board, but with the important difference that we do not control the data-copying loop. Therefore, we cannot extend the `bufmgr` library to handle the required data transformations as efficiently as it could be, so we need to implement the data transformation separately in this case. An additional complication is that data alignment in the GPU can (and will) be different that in the host. As a consequence, the same data structure might consume a different amount of memory in the host or the device, making a direct copy of an array of these structures impossible. Dealing with this limitations will lead to higher memory consumption on the host and a performance penalty.

The performance gain in an FPGA comes from the ability to pipeline long computational chains, producing one result per clock cycle. In the case of the GPU, the gain comes from the massively parallel architecture and its capacity to distribute work among many units, mostly independent threads, while properly accessing the required data in memory. The FPGA is capable of executing all computations at the same time, so it can be pipelined for optimal performance for an specific algorithm, but the GPU can execute only one given instruction per SM at the same time. Therefore, the parallelization of the algorithm for use on a GPU is closer to a many-core SIMD architecture than to a dataflow architecture, and must be addressed differently.

One good way to visualize the computing model of NVIDIA GPUs is to consider them lightweight threads. In fact, CUDA threads *are* lightweight threads, with some additional platform restrictions based on thread affinity and locality that apply to threads in the same CUDA block. The SPH algorithm can be parallelized in many ways on this architecture, and we will present three of them, summarized in Fig. 6.9.

On version 1 (Fig. 6.9.a), SPH computations are parallelized by computing one full

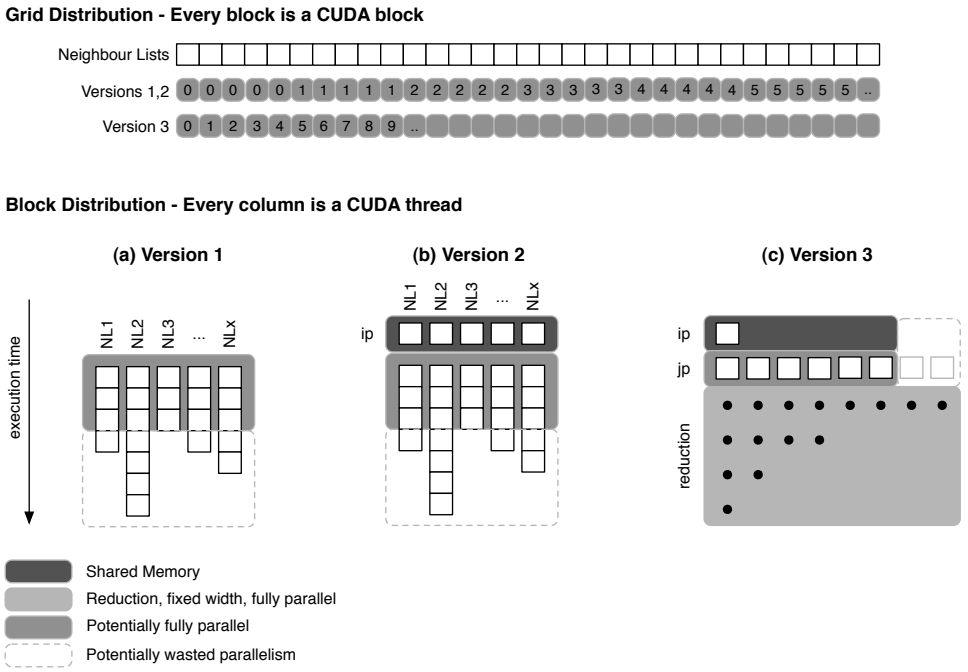


Figure 6.9: Parallelization strategies for the different CUDA versions.

neighbour list per thread. With this approach, every i-particle is assigned to a single thread, while j-particles are gathered from global memory on demand. Because the length of every neighbour list is not fixed, some threads in the same CUDA block will have to wait until the longest list of the block has been processed before completing the execution of the block, wasting some computational resources. Also in this version, the data is organized as an array of structures (AOS), so the data organization is very similar to the arrays on the host memory. This version represents a valid but straightforward porting of the algorithm to GPUs, as a reference point to measure other GPU implementations against.

Version 2 (Fig. 6.9.b) is a variant of the previous one, that uses a different data arrangement in global memory. In this case, data is stored as an structure of arrays (SOA), so each value (or vector) is stored in a separate array. This allows the compiler to better optimize the memory accesses and to avoid fetching unneeded data. Also, the i-particle data is cached in shared memory by each thread, as it will be reused for the whole neighbour list.

Version 3 (Fig. 6.9.c) is a different approach. Each neighbour list is assigned to a single block, and each interaction in the list is computed in parallel by a thread inside that block. Because the list is assigned to the whole block, only a single i-particle has to be cached in shared memory, reducing the limitation of the shared memory size. Also, as the interactions are now distributed among several threads, the accumulation of the results has to be performed as a reduction operation, following the recommended procedure from Mark Harris in documented in [39].

The performance of all three implementations is summarized in Fig. 6.10. At low particle number, version 3 is faster than versions 1 and 2, but all of them converge to approximately the same speed-up as it reaches the bigger data set. Overall, they provide a maximum speed-up of 26x compared to **mp-pc109**, and 14x compared to **autana**. As the speed-up is not constant, there is a more complex interaction operating as a limiting factor. A better understanding of these factors will provide a better path for improvement, therefore a profiling of the code is needed.

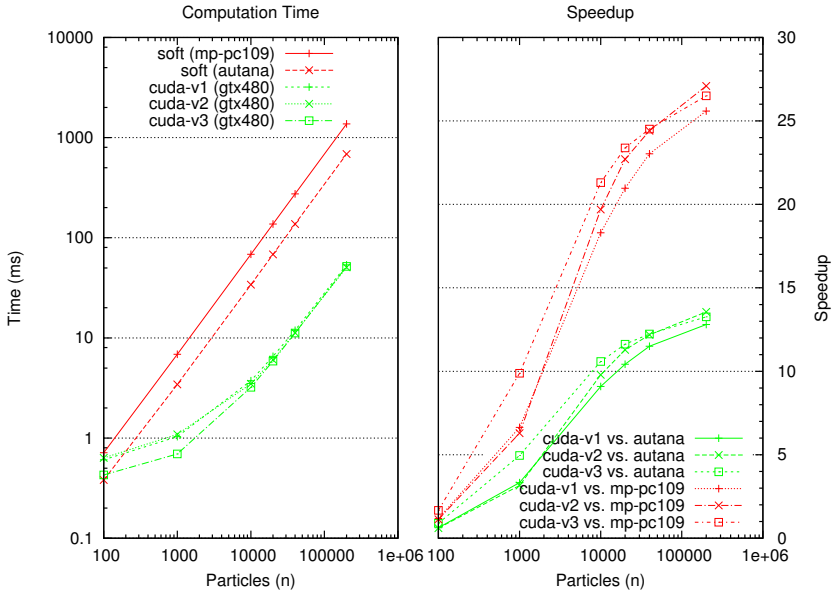


Figure 6.10: Computation Time and Speedup for CUDA core using a GTX480.

Fig. 6.11 provides a gross overview of the GPU time (in percentage of the total) for both 1k and 200k particles. For one thousand particles, both version 1 and 2 have a similar distribution, with the GPU spending approximately the same proportion in communication and kernel execution: 17% communication, 83% execution. However, version 3 spends a much higher fraction in Host-to-Device communication with 47% in communication and 53% in execution, but because the data load time is almost identical for all versions, the conclusion is that the kernel executions are significantly faster for version 3 than for the other versions, which is in agreement with the execution times in Fig. 6.10.

However, this advantage practically disappears for the case of 200k particles, where all three versions show a very similar distribution of around 45% communication and 55% execution. It can be explained as follows: For a small number of particles, the parallelization of one-list-per-thread is not able to fully occupy the resources of the GPU because not enough blocks are created, thus limiting the performance. For one thousand particles, when using 128 thread blocks only 8 blocks are active, which is not enough to use all 30 MPs in the GTX480. On the other hand, the one-list-per-block approach allows a better occupancy of the MPs, distributing them evenly as more

lists are present. This provides an increase of about 2x in speed-up. As the number of particles increases, the number of units occupied in the GPU saturates and only algorithmic differences are present, being version 3 better by a small margin.

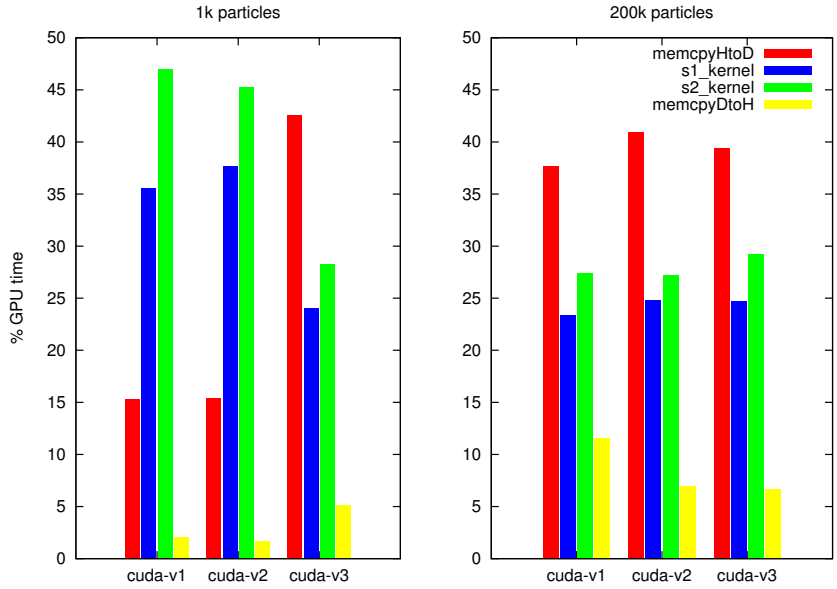


Figure 6.11: Profiling of `CUDA core` versions running in a GTX480 for a 1k and 200k particles datasets.

From the 45% - 55% distribution of communication and execution for all three versions with high particle numbers, it cannot be said that either one is dominant and in order to improve the performance, both must be reduced. The communication time is determined mainly by the bus speed (already a 16-lane PCIe v2, a very fast interconnect) and the amount of data to be transferred. The bus speed can only be changed by newer generations of GPUs, so the focus has to be in reducing the amount of data. For the 200k particles data set, 40% of the communication (that is, 18% of the total time) is consumed in the transfer of the neighbour lists. Optimally, the neighbour lists can be computed in the GPU, making the transfer unnecessary. As an alternative, the neighbour lists can be transferred asynchronously and overlapped with the computations.

As for the particle data, the data transfer can be reduced only if the active particles in the previous timestep are updated and an interpolation kernel is added. Positions and velocities of all other particles are computed internally by the GPU, bringing them to the current timestep without the need for a transfer. In this way, the full dataset needs to be transferred only a fraction of the time.

In order to analyse the execution of the kernels, in particular the parallelization strategies used by the CUDA compiler and its effect in the performance, the CUDA Occupancy Calculator is used to analyse all kernels, and the output is summarized in Tab.6.3. On a first glance, the device occupancy is below 50% and all kernels use too

	version 1		version 2		version 3	
	S1	S2	S1	S2	S1	S2
Threads / Block	128	128	128	128	64	64
Registers	28	33	23	27	23	28
Shared Memory	48	56	4688	6768	424	476
Occupancy	50%	38%	38%	25%	50%	50%
Warps	4	4	4	4	2	2
Registers	3584	4608	3072	3584	1536	2048
Shared Memory	512	512	5120	7168	512	512
Limiting	Regs.	Regs.	S.M.	S.M.	Warps	Regs.
Max Blocks / MP	4	3	3	2	8	8

Table 6.3: Characteristics of CUDA kernels versions, summarizing resource usage and GPU occupancy for a GTX480 in CC 1.3 mode. The top part are settings/compiler results, while the bottom part are analysis values produced by the *CUDA Occupancy calculator*.

many registers: above 23, when the optimal is typically less than 12 (for CC 1.3 devices). Interestingly, the limiting factor for maximum occupancy(see 2.2) varies together with the version. Some kernels are limited by the number of registers used, others by the amount of shared memory, and one by the number of warps.

The limitation of number of registers used comes primarily from the size of each kernel. All of them compute the full interaction, and many variables are passed and used during the whole execution span. Kernels from version 1 used more registers, because the i-particle data is stored in registers. This is reduced in versions 2 and 3, as it is moved to shared memory. Eliminating the iteration over the neighbour list in version 3 does not reduces the number of registers, so in order to reduce it further, the kernels will need to be split in smaller operations, which will add additional IO between the MP and the global memory.

The advantage of caching i-particle data in shared memory is evident for version 2. Instead of using registers, the use of shared memory allows a reduction in IO from global memory, as the i-particle has to be fetched only once and all subsequent accesses are from faster, shared memory. This advantage is compromised by the lower occupancy of the kernels, when compared to version 1. The lower usage is shown as the actual number of blocks allocated to a MP is reduced by one, in practice computing a lower number of neighbour lists in approximately the same time.

The number of allocated blocks is at maximum in version 3, with 8 concurrent blocks per MP. In this case, the occupation is also higher than in previous versions, but still not using the full capability of the GPU with a top occupancy of 50%. Nonetheless, it provides a measurable difference for smaller datasets.

However, this advantage is reduced as the number of particles increases, being version 2 slightly faster for our 200k particles dataset. From this behaviour we can

infer that as particle number increases, the amount of IO between the MP and the global memory saturates the available bandwidth and becomes dominant of the kernel execution time. This is reasonable, as particles are randomly distributed in memory and requiring a gather operation to fetch the particle data needed, which prevents coalesced accesses. In order to improve the memory accesses, data or neighbour lists indexes must be reordered in memory to regularize them.

Reordering the NLs is a very expensive task with limited benefit, as it is required to analyse all lists and reorder the interactions for locality. This will break the caching of the i-particle data and partition the reduction algorithm at the end of version 3. Reordering the particle data is more feasible, as the smoothing length defines the range around the locality needs to apply. Combining this previous analysis with an efficient tree algorithm in the GPU device will provide advantages both in the memory transfer between the host and the GPU and the computing time for the SPH kernels.

6.8 Application performance

The library code was integrated with a testing application developed by Peter Berczik. Working together with him, the following results were produced that exemplify the successful integration and use of the library with actual astrophysical code. Tests were performed in a single node in the Titan cluster, were every node is technically equivalent to **mp-pc109**. The GPU code used is a previous version of the library, equivalent to the code described as *CUDA version 1* in Section 6.7, running in a GeForce 8800GTX (CC 1.0).

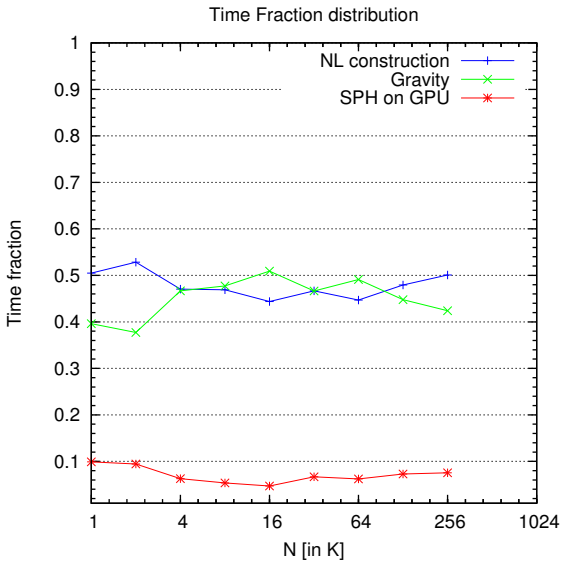


Figure 6.12: Computing Gravity with an Accelerator and SPH with a GPU. Plot by Peter Berczik, reproduced with permission.

Fig. 6.12 shows the effect of using the accelerator in the time fraction distribution.

Compared to Fig. 6.1, the SPH computational time is reduced from more than 50% to less than 10%, balancing the time distribution almost equally between the neighbour list construction and the accelerated gravity force computation. A detailed profile of the performance gains in the SPH fraction is shown in Fig. 6.13, along with a comparison of the FPGA and GPU cores. Performance gains are shown to be in the order of 5x-12x for the FPGA core with an MPRACE-1 and 13x-19x for the GPU core.

It is also important to see the global execution time for the application and the effect on its performance. This is summarized in Fig. 6.14, with both cores providing a speed-up in the range of 1.6x to 2.4x. From the Amdahl Law estimation of 2x-2.5x shown in Section 6.1, we can consider it a good result. Improvements can be done for medium number of particles, but since the maximum bound is 2.5x the possible overall gain will be low, so further improvements for the application should focus in other more time consuming sections like the neighbour list construction.

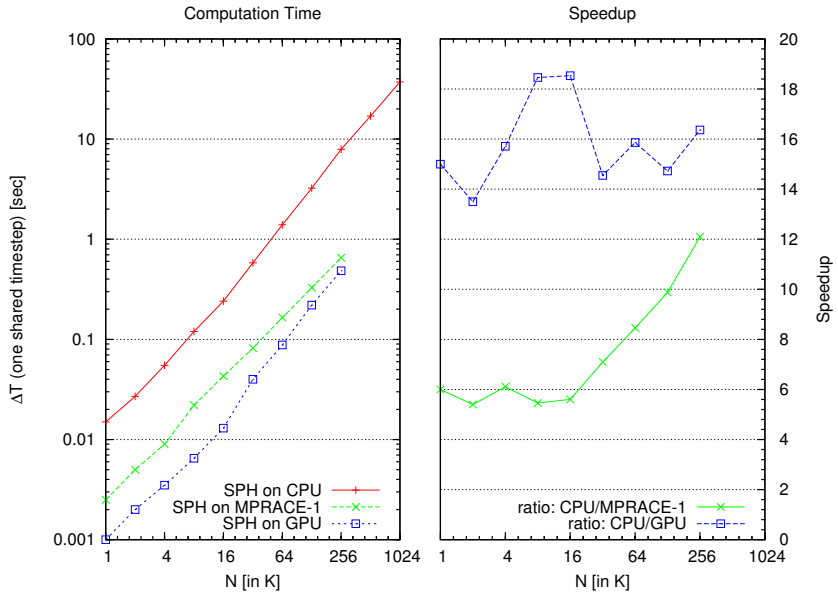


Figure 6.13: Computation Time and Speedup with reference code for the SPH fraction.

6.8.1 Comparison with previous work

When comparing the FPGA core with the results from Nakasato, several advantages can be pointed out: They are faster, a modest improvement providing a 6-12x speed-up against their result of 5-11x, but using 1/4th of the hardware resources, because the MPRACE-1 board uses 1 Virtex-II FPGA while the PROGRAPE-3 has 4 Virtex-II Pro. In this sense, it provides 4 times better efficiency for approximately the same end performance. Building over their remarks in the importance of the FPGA-to-host communication, which we also described in the FPGA section, the improvement over previous work can be attributed to the efficiency of the `bufmgr` library when handling

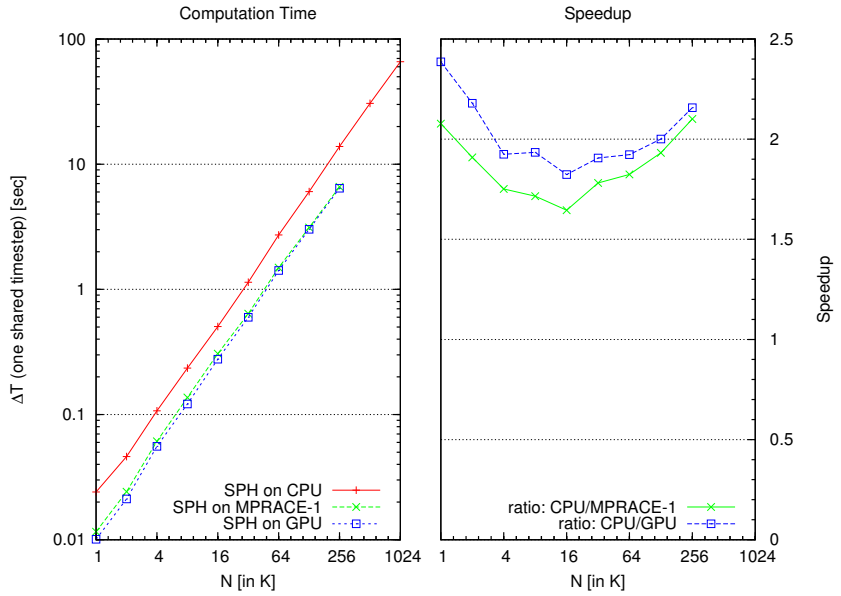


Figure 6.14: Computation Time and Speedup with reference code for the full application.

the transfers, as they are directly related. Furthermore, by recognizing the boundaries of achievable performance, we can also plan better the next developments.

Comparing with the work from Harada is possible within some limitations, as his computing time includes rendering and a clear fraction allotted to it is not given. Also, to be fair, it should be done against a similar GPU. This could be done with the aid of Fig. B.1 in Appendix B, where the performance with a GF8000GTX as the one used by Harada is shown. However, their CPU reference takes 6.725 s to do 262,144 particles, which can be normalized to 200,000 in 5.130 s, while our CPU implementation taking only 1.371 s for the same number of particles. Similarly, their GPU computing time would be 179.9 ms, very similar to our results of 142.7 ms. The computing times in GPU are therefore comparable, but the speed-up figures are not. The performance itself is dependent on the characteristics of the system simulated, so while the results are very comparable, it is important to be aware of the possible differences.

In order to compare with the work from Herault, we can extrapolate our results with a GTX280 to 600k particles. From 200k in 102.7 ms, that would be 308.1 ms for our simulations, while their code takes only 207 ms for the same task. However, their implementation does the neighbour search inside the GPU, so their force computation does not include any data transfer, which in our case accounts for 40% of the time, or 123.24 ms. We could then compare their 207 ms to our computational fraction time without neighbour list transfer of 184,86 ms, or an 11% improvement for the force computation.

6.9 The VINE implementation

VINE is an astrophysical simulation application, originally developed by Markus Wetzstein in collaboration with Thorsten Naab and Andy Nelson. The application is designed to perform best on shared memory supercomputers, but has also been ported to distributed memory systems. It is built around a particle tree and hierarchical time-steps, using aggressive caching methods to prefetch blocks (*clumps*) of data and compute them efficiently. The clumps can also be seen as a flattening of the tree leaves, used for performance optimization. An aggregation of clumps, called *groups*, allow the use of accelerator boards like the GRAPE-6 board for gravity computation. This is because in order to compensate for overhead and fully utilize the pipelining capabilities of the board, it is necessary to have a big number of particles for processing, so the best way to accomplish this while keeping the advantages of the tree algorithms is to aggregate the clumps. For a more detailed discussion of the effect of group size in performance, see Markus Wetzstein Dissertation [95].

Particles in the system are classified in at least 2 types: SPH and NBODY. SPH particles are subject to hydrodynamic and gravitational forces, while NBODY particles do not have hydrodynamic interactions. Other types are possible, but are not of interest for our tests. Particle data is organized as a collection of arrays, where data is linked by index among arrays and every index corresponds to a single particle. Tridimensional arrays are used to store velocities and other vector variables, while unidimensional arrays are used to store scalar variables like density. Other values used very often together, like position and mass, are stored in a joint, fourth-dimensional array.

For the SPH section, the code reuses the clumps to create neighbour lists for active particles, and stores them in temporary structures organized as a small matrix. The size of this matrix is chosen so it fits easily in the cache of current microprocessors. As the matrix contains only the indices for the neighbouring particles, the code must still fetch the particle data to compute the interactions, so it has to perform a gather operation among all the involved data arrays by following the indexes stored in the temporary matrix. After all the interactions for the matrix have been computed, it first checks for more neighbours for the current set of active particles, until no more neighbours are available. This is done in this way in order to benefit from both the data locality in the matrix and the tree. After the current set of active particles has been fully computed, a new set is selected from the list of active particles, until all of them are processed.

However, this algorithm brings several complications when ported to interact with an accelerator. First of all, we have to transform this temporary matrix into a linear neighbour list array, while respecting the limitation that a neighbour list cannot be partitioned. Second, we have to increase the number of lists to process in any given batch in order to reach efficiency. For this last point, similar to the GRAPE boards for gravity, we can apply the same technique and merge clumps to create bigger temporary structures for use by the `racesph` library.

For the linearisation of the array, since the tree walk with the temporary matrix gives potentially partial lists, before computing the interactions it is needed for the host to walk the full set of neighbours for any given particle. Doing this one by one would

be very time consuming, so it was chosen to extend the merged clump to not only more active particles but also more neighbours, and once it has been fetched, it will walk the merged clump to produce a set of neighbour lists that can be safely processed. This might seem like it imposes a very arbitrary limitation to the length of the neighbour lists for any given particle, but in fact this is already limited artificially during the tree walk by constants configured in the program. We can therefore guarantee that for any given active particle, we can store its neighbour list up to the limits established by the code.

In addition to these modifications, it is necessary to consider when the use of an accelerator will result in a performance gain, particularly because the tree code is very efficient. For this context, performance means less computing time, and this is directly related to the number of active particles in the system at any given time-step. Since VINE has the option to use hierarchical time-steps, it is possible that for a certain time-step only a very reduced number of particles are active, so it will not be profitable to use the accelerator as the CPU can handle this case much more efficiently. For this reason, we added threshold conditions to VINE, so it uses the accelerator only if a minimum number of particles is active. After some experimentation and evaluating the results from Fig. 6.8 and 6.10, we set up the threshold at one thousand particles for our test system.

Last but not least, it is important to note that like many other scientific applications, VINE is written in Fortran. This brings some technical challenges on how to interface a C++ library to Fortran code, which can be summarized as follows:

- **Compiler optimizations**. Accessing functions in one language from another imposes several restrictions, the most important being that the compiler cannot cross language boundaries (from Fortran to C++ or vice-versa) to perform optimizations. Compiler hints like inlining will be ignored, and loop unrolling or register reuse will not be possible. Therefore, function calls will be full-function calls, so tight loops with many calls across the boundary must be avoided and no optimizations across the border must be assumed.

- **Parameter passing**. When calling a function, the way parameters are passed must follow a certain convention, and Fortran and C++ follow different ones. While Fortran passes all parameters by reference (the address of the variable is copied into the stack), C++ passes by default all scalar values by value (the value of the parameter is copied into the stack), and arrays are passed by reference. C++ supports optionally pass-by-reference for scalar parameters, but it must be selected for every parameter passed.

- **Array ordering**. When storing a multidimensional array in memory, Fortran and C++ follow also different conventions. In Fortran, arrays are stored in column-major order, while in C++ they are stored as row-major. This has the consequence that when a reference to an array is passed across the border, the order on the indexes has to be reversed: An array with indexes [i][j][k] in Fortran has to be accessed as an array with indexes [k][j][i] in C++.

- **Exception handling**. Error conditions in C++ programs are typically signalled by the use of exceptions. However, Fortran has no exception handling mechanism, so exceptions have to catched and appropriate return codes sent by the library over the return path to VINE.

- **Function name mangling**. C++ uses name mangling (also known as *name decoration*) to create unique names for functions and add additional information to the function specification that can be used by the linker. Because the name mangling is not standardized and changes with the compiler used, C++ functions calls cannot be added directly from Fortran without also fixing the C++ compiler used.

For all these reasons, the Fortran code cannot be interfaced directly to the C++ and a wrapper class was implemented. While a C wrapper was already in place and covers part of the issues mentioned, it still leaves parameter passing and array ordering as non-addressed issues. Since modifying VINE to change the array ordering would be counter productive, as the code is already optimized for this arrangement, the library gets a new wrapper interface to improve the communication with VINE specifically.

In this new interface, the translators to convert the data from VINE format to the library format for particle and result data is located inside the wrapper interface. This allows for the use of the translation capabilities of the library without interfering with the VINE data arrangement. In addition, instead of processing the extended neighbour matrix in the Fortran code, the matrix is passed to the wrapper and processed inside the library, displacing two tight loops from Fortran to the C++ side and improving the readability of code.

The results for VINE are analysed in two ways. First, there are correctness tests, where a simulation running in a single host is compared with results using the `racesph` library. The second are performance tests, where the computing time is compared for the same simulations.

The correctness can be summarized in Fig. 6.15. The plot shows the energy evolution for the collapse of an adiabatic sphere of 5K particles. Curves for each core are superimposed, providing a visual comparison. Because the system evolves, small changes accumulate and will effect the results as it develops. This is most visible for the `CUDA` core, as the system is most dynamic, it starts to deviate from the solution with other cores. Because the `software` and `SSE` cores have the same equivalent precision and use the same VINE interface, the difference can only be attributed to the CUDA core itself. This result is present with both a GTX260 and a GTX480 card, so it is not attributed to the limited accuracy of the previous generations (the GTX480 has the same accuracy as a CPU for single precision operations). Deeper investigation is pending to uncover either the exact causes of this discrepancy, possibly a programming mistake.

Performance results are detailed in Tab. 6.4. In short, the accelerator library brings little to no benefit for the overall execution of VINE, a 23.85% improvement in the computation of SPH, but no gain in the total execution time. This result is in agreement with the independent results provided by Markus Wetzstein with an earlier version of

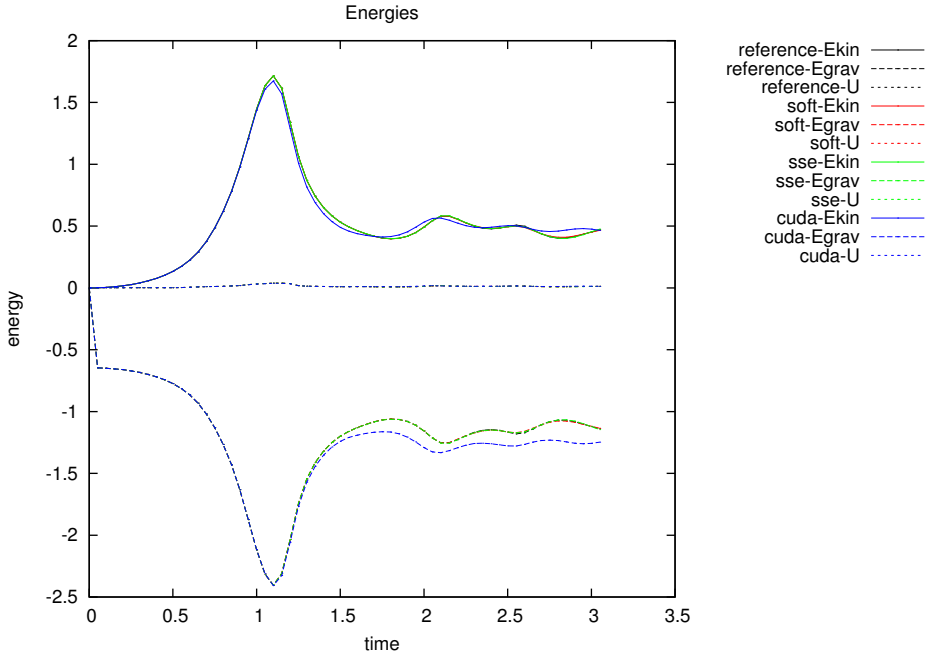


Figure 6.15: Energies from the collapse of an adiabatic sphere with five thousand particles, compared for several computing cores.

the library and the MPRACE-1 board. It is interesting to understand the reasons and implications of it. First, the problem size is five thousand particles, so for the CUDA core, the speed is limited to ∼7.5x maximum. But even if the size of the problem would be increased, the clumping of neighbour lists would limit the size of lists processed at any given time to approximately this amount. Increasing the size further would be detrimental for the tree performance, so a balance of the parameters must be reached.

Additionally, the clumping has some negative consequences on the neighbour list construction. Clumping the particles implies that only a subset of neighbour lists are sent to the GPU on any given time. Because these clumps are (hypothetically) only a fraction of the total size, the transfer time is less than optimal, as GPUs benefit from big data transfers. In consequence, to process one step, multiple clumps must be processed one after another, and the full set of neighbour lists is never constructed, so the list cannot be cached and reused by the GPU in a later step. Since the neighbour list transfer is a significant portion of the CUDA core execution time, this affects the performance accordingly. The effect would be even more dramatic when processing time steps with only a fraction of the particles active.

When analysing the execution time, it is important to remember that the 7.5x speed-up factor does not applies to the full execution time of the application but only to the SPH computations, which in this case amounts to less than 20%, and the percentage on the accelerated runs include the overhead of the VINE interface. From the information provided, we can approximate the overhead time to be 19.37 s, assuming a constant

	autana	autana SSE	GTX260	GTX480
Total time	158 s	158 s	165 s	174 s
Gravity	123.43 s	125.22 s	131.11 s	147.9 s
Percentage	78.09%	79.125%	79.144%	84.6726%
SPH	30.85 s	29.19 s	30.34 s	23.49 s
Percentage	19.51%	18.445%	18.31%	13.445%

Table 6.4: Comparison of execution times for VINE with and without acceleration of SPH

speed-up of 7.5x and not compensating for tree traversing and smaller time-steps.

All in all, while the results of integrating with VINE are not as successful as desired, they provided valuable information that will be used for the implementation of more advanced simulations codes that can use accelerators efficiently for SPH. VINE also provided an opportunity to demonstrate the extensibility of the library. While the main development for VINE was done oriented to the use of an FPGA accelerator, when the GPU cores were available they could be used with minimal changes to the VINE code, only a few flags and constants to instruct the library to use the new core and to accommodate for bigger clumps.

Chapter 7

The raceGRAV Library

Gravity forces are unlike the SPH forces computed in the previous chapter in several aspects. First, they are long range forces, so they are not localized in one region of the simulation but can interact any other particle in the system. Next, the complexity is lower, as they are computed in one single step and there are fewer FP operations needed. The formulas computed by this implementation correspond to those described in 1.1. Similarly to other accelerator implementations, the derivative of the acceleration (*jerk*) is also computed in order to support the implementation of 4th order Hermite integrator schemes[64].

Direct force summation is the computation of force interactions between every particle in the system. Because the direct force summation scales as $O(n^2)$ with the number of particles and every interaction is independent, it is an easily parallelizable task. As it also consumes the most significant portion of computation in the CPU, significant efforts have been done to speed it up. One of the most successful has been the GRAPE family of accelerators, already described in Section 2.4. Other implementations have been done for FPGAs[74, 35, 85, 29].

The goal of the raceGRAV and the implementations described in this section is not to compete with these other implementations but to complement them. Makino and Aarseth[66] describe how special hardware can support other schemes different than direct summation, in this case the Ahmad-Cohen scheme (ACS)[9]. In this scheme, the gravitational force is divided in two components, a far reaching force that interacts with distant regions of the systems and a local force that interacts only with neighbouring particles, with each component using different time-steps. While their tests with GRAPE hardware were promising, it also shown that the full performance of the board was not reached because the average number of active particles was too low. Our goal with the raceGRAV is to provide an accelerator that can work concurrently with the direct summation implementations, efficiently accelerating the computation of local forces.

In the following sections we will review the architecture of the library, the CPU and FPGA implementations, and discuss the results in terms of performance and accuracy. A GPU implementation is not done because several good ones already exist, and it would be a more sensible approach to modify them for ACS than made a completely new implementation.

7.1 Previous and Related Work

There are many developments for acceleration of the gravitational forces, but arguably the best known is the GRAPE family of accelerator by Makino et al.[27, 67, 65]. The GRAPE is an ASIC design with multiple pipelines for the parallel computation of gravitational forces by direct summation, that can be scaled up to several hundred processors by a combination of hardware and software[67]. Since its introduction in the 1990s and until recently, it provided a level of performance unmatched by any other solution, with speed-ups several orders of magnitude above the performance of CPUs. The GRAPE cards enabled scientists to perform simulations in a timely manner that where otherwise not possible or prohibitively time consuming.

Because the computing pipelines are fixed in the hardware design, it lacks the flexibility to be useful in other applications or to address some assumptions made during its design. An alternative to the GRAPE based in configurable FPGA technology is the PROGRAPE (PROgrammable GRAPE) series introduced by Hamada et al. [33], with a performance comparable to the GRAPE-3. The PROGRAPE-3[36], as mentioned previously, contains multiple FPGAs that can be used to compute different parts of the algorithm simultaneously, like some dedicated to gravity and others to SPH. The performance is higher than a GRAPE-6A for direct summation, and it is capable of supporting Barnes-Hut tree algorithms. A disadvantage is that it does not provide discrete local memory onboard (only memory inside the FPGAs), so the computation is limited to 16000 particles at a time, but it can be supported by partitioned in software. In theory, they can also be used to compute gravitational interaction lists.

Being the use of the GRAPE cards so common and with the increasing speed of CPUs, Nitadori[80] created a replacement version of the GRAPE libraries that use the SSE capabilities of the processor instead of the actual hardware, providing a fraction of the performance while maintaining code compatibility without the need for any special hardware.

With the introduction of programmable GPUs, higher bandwidth and performance became available. Because of its highly scalable and ease to program nature, gravitation was one of the proof of concept fields used to exemplify the capabilities of the new platform. One such proof of concept applications was done by Harris et al. and documented at the GPU Gems 3 book[78], providing the same performance of a GRAPE card with the advantages of programmability at only a fraction of the cost.

An subsequent implementation is the Chamomile Scheme from Hamada[35], which fully utilizes the capabilities of the GPU for the computation of direct summation forces. The GraCCA cluster[85] was one of the first parallel implementations to show the efficient usage of GPUs in a cluster install, with parallel efficiency over 90%.

An interesting development is also the work by Portegies Zwart et al., were the implement a GPU accelerator using Cg[103], and later rework it to work with CUDA[29], as itt provides a clear comparison of the advantages of CUDA against the older and more limited Cg. While the performance is very similar, the flexibility of CUDA is much higher. The performance with Cg is slower than the GRAPE cards, but the newer implementation in CUDA improved over it in a relatively short time.

Gravity Functions	
`getMaxIPcount`	Maximum Nr. of i-particles permitted
`getMaxParticles`	Maximum Nr. of Particles
`setEpsilon`	Set Epsilon
`setParticles`	Load Particles
`calcDirectSummation`	Perform direct summation over the particles
`calcInteractionList`	Compute interactions following a list

Translators	
`setParticleTranslator`	Set the Particle Translator
`setResultTranslator`	Set the Result from Translator

Debug Functions	
`log`	Output string to the log
`debug`	Output string to the debug log

Table 7.1: GravCore class interface

7.2 Architectural Overview

The architecture of the raceGRAV library is a simplified version of the raceSPH architecture, together with a few additions. It follows the same concept of computing cores, but simplifies the debug interface. Buffer Managers and Translators are supported in the same way. The main distinction comes in the functionality supported. Two computing functions are implemented in the base core class, in order to support both direct summation and neighbour list based force interactions.

Computations performed are based in Load-Compute-Read cycles, that is: Load particles, compute forces and Read results back. Computations to be performed are defined based in the particles indexes of the loaded set. Direct summation follows the same organization as the GRAPE accelerators, defining i and j particles and computing the force summation of all j-particles over every i-particle in the array, with one difference: both i and j particles are part of the same particle set uploaded to the processor. Table 7.1 summarizes them.

On the design phase of the library, we realized that several versions would be needed. In particular, separate designs might be needed to implement direct summation and interaction lists, and we could devise additional versions including, by example, predictor units for inactive particles in the accelerator memory or interaction lists with shared neighbours among i-particles. It was therefore needed a mechanism to pair the designs used to the core implementations containing the appropriate driving code.

Our solution was a `Design` class, containing both the design file together with a *Capabilities* set, specifying the operations supported by the design. Cores can therefore inspect the capabilities of the design being used and use the corresponding code. A set of access methods complete the required functions.

7.3 CPU and SSE implementations

As usual, the CPU implementation is a reference for validation of the results when integrated with application codes. The SSE implementation follows the same basic guidelines used in the raceSPH library, using XMM structures as depicted in Listing 6.1. No pairwise functions are used in the computation of gravity forces, but the accumulation is sensitive to the precision used. A simple implementation would accumulate one i-particle per SSE vector, accumulating the result using the single precision operation. On tests (see Fig. 7.2 in Sec. 7.5) this shown to be inadequate, so it was replaced by an accumulation over a regular (non-SSE) double precision register. Other alternatives tested used Kahan's summation algorithm[50], and yet another a partial accumulation on SSE that is accumulated over regular intervals on a double precision register, with very similar results both in speed and accuracy.

7.4 FPGA implementation

The FPGA pipelines were described using our in-house Pipeline Generator[60], using a frame design (the logic surrounding the pipeline) similar to the raceSPH. Designs were prepared by Gerhard Lienhart and finalized by Yang Yuning, and validated in a MPRACE-1 board. In addition to the force computation, the design implements virtual pipelines similar to those implemented in GRAPE, in order to reduce the memory transfer requirements as noted by Makino[64]

One important consideration the library needs to make is the handling of the virtual pipelines for the direct summation. The design contains 16 virtual pipelines implemented in software as separate i-particles, that communicate to one physical pipeline. However, the design requires between 14 and 16 i-particles to operate properly because of constraints in the latency needed for the accumulators, so the library must fill with dummy particles the empty slots when less than 14 particles are sent. The design presented here contain one physical pipeline, but up to four physical pipelines are supported based in the memory bandwidth, more if additional CLBs are available.

7.5 Results

Tests were run with their own test applications by two astrophysicists, Peter Berczik and Ingo Berentzen. Studied was the performance, in wall-clock time and Gflops, as well as the relative error when measured against a reference CPU implementation.

Performance tests are run in a Intel workstation with a Core2Quad Q6600 Quad-Core 2.4 GHz CPU with 4 GB of DDR2-1066 memory running Ubuntu Linux, using an MPRACE-1 board with the direct-summation design and a GRAPE-6A board. For CPU tests, only one core is used. The GPU performance shown is a GeForce 8800GTX (CC 1.0) running an early version of CUNBODY-1[35].

Fig. 7.1 shows the execution time for a particle range between 1k and 128k, within the limitations of the GRAPE memory. Here, both FPGA and SSE implementations are faster than the CPU for most relevant ranges, but still about one order of magnitude slower than the GRAPE-6A or the GPU.

However, when comparing the Gflops normalized per pipeline, the MPRACE-1 and the GRAPE-6a have similar relative performance, with the MPRACE-1 showing an advantage for small number of particles without penalty for bigger numbers, which is the desired use case when switching to interaction lists. The Gflops for the SSE shows the performance per CPU core, but it could also be considered a 4-pipeline implementation, in which case the normalization would be a curve slightly over the CPU line.

The three remaining plots in Fig. 7.1 show the relative error for the potential, the acceleration and its derivative. Here the MPRACE-1 shows a better accuracy than the GRAPE-6A for the potential and the jerk, but slightly lower accuracy for the acceleration. This can be attributed to the limited precision used in several of the internal operations in the pipeline. The increased accuracy of the jerk is the important factor to improve the results of the integrator.

Fig.7.2 shows a more detailed comparison of the accuracy of the accelerators for a simulation with 64k particles. Plots are a distribution of relative error against the radius, providing a visual distribution of the error values inside the sphere. MPRACE-low refers to implementations with single precision accumulators, while MPRACE-high use double precision accumulators for the force summation. These plots show very clearly the effect of the precision of the accumulator in the pipeline over the accuracy of the results produced, with MPRACE-low roughly two orders of magnitude below the GRAPE-6A for the potential and the acceleration. Increasing the precision sets the accuracy on the same order of magnitude, although with differences in shape. The jerk from the GRAPE-6A is known to be of limited accuracy, and the MPRACE-1 is capable of providing between a relative error that is roughly from one to three orders of magnitude better.

Comparing the performance of the SSE core with the hand written routines from Nitadori over the GRAPE interface library, his routines provide an additional 0.5x - 1.0x advantage against our cores. After inspecting the generated assembly language for our code, it can be attributed to non-optimal register placement by the compiler, which in our case is done automatically. This is in our opinion, an acceptable trade-off for more portable and more readable source code, that can improve with newer version of the compiler used.

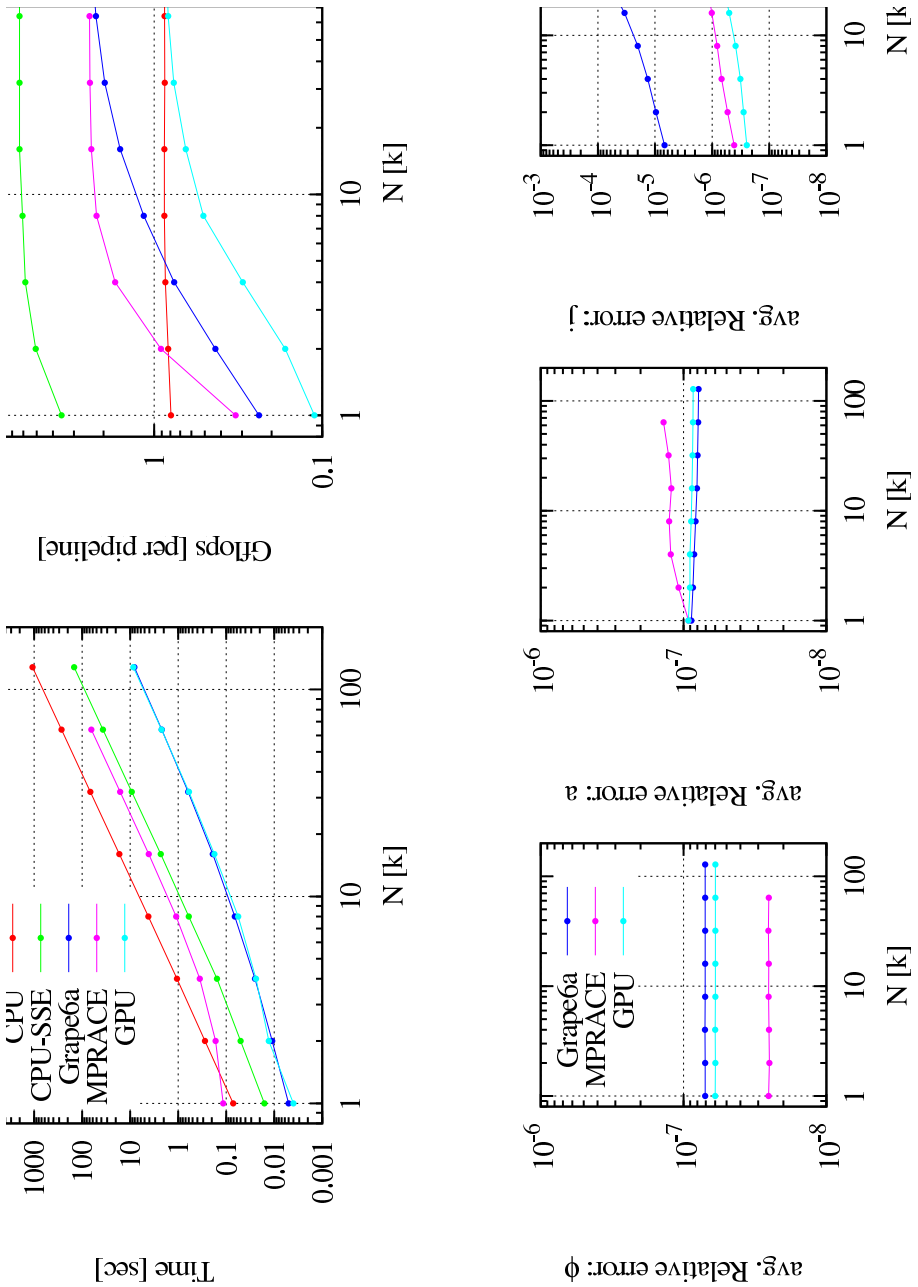


Figure 7.1: Direct Summation Performance. Plots by Peter Berczik and Ingo Berentzen, reproduced with permission

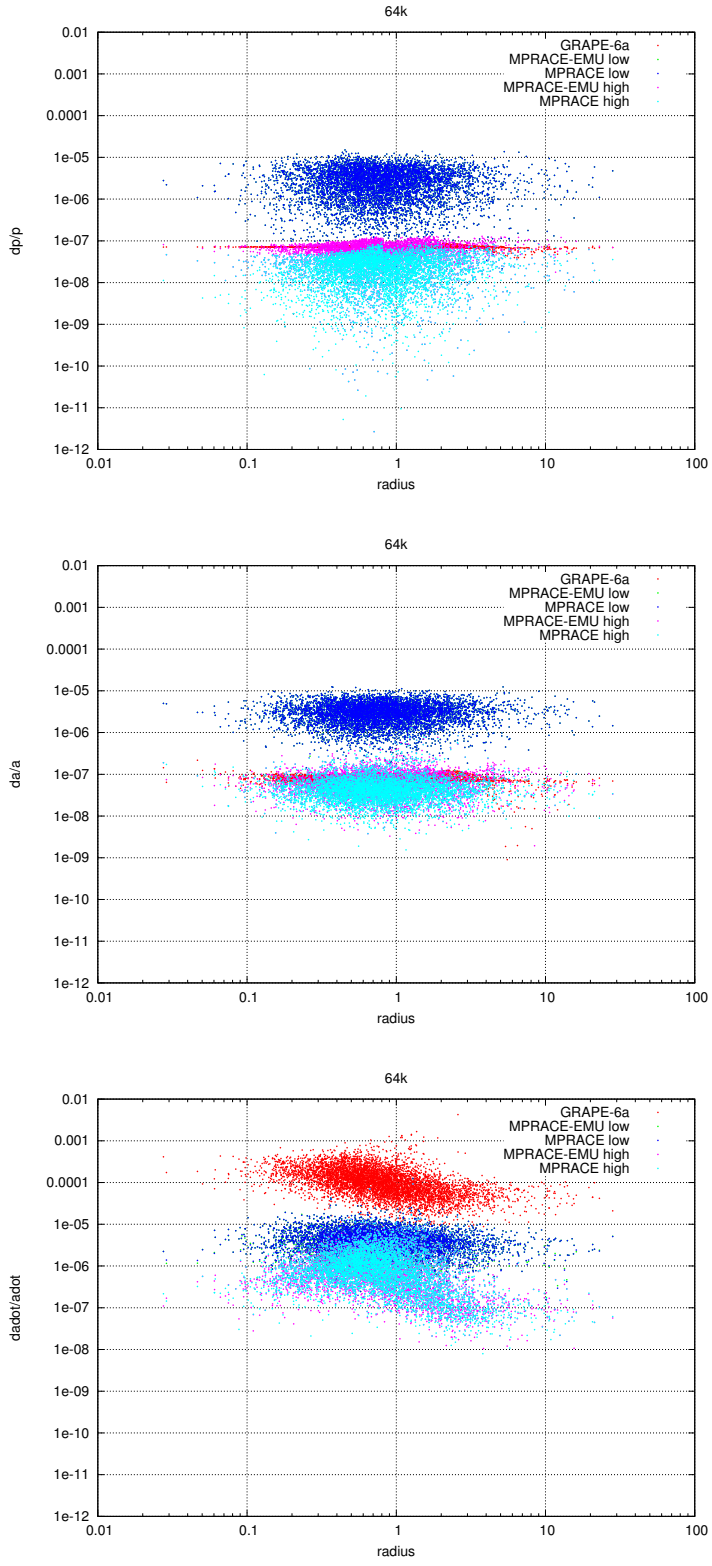


Figure 7.2: Accuracy of the cores. Plots by Ingo Berentzen, reproduced with permission

Chapter 8

Summary

The goal of this work was to use special hardware to increase the performance of actual astrophysical simulations by accelerating the computation of SPH forces as well as the gravitational forces with high accuracy. The rest of this section makes an overview of the achievements documented in this work, providing a contribution to the state of the art in SPH computations with special hardware and the modelling and refinement of the software interfaces for communication of accelerators with special hardware.

FPGA development and software integration

When using FPGAs as coprocessors together with host computers, it is easy to focus on the FPGA design and forget on the required software and how it can affect the performance. Most devices require at least a device driver, but if high speed transfers are needed a DMA engine is the only option. This is normally complex and very dependent on the engine used.

A set of libraries were implemented in order to allow the efficient communication between FPGA based accelerators and the application code. They represent the two lowest levels of a software stack, or framework, for general coprocessor application development, as both the PCI driver and the MPRACE library are generic enough to be used by many boards and in several applications. In addition, the MPRACE handles the communication with the DMA engine developed by Gao.

Currently, the framework is a candidate to replace the current driver implementation used by the next-generation ROBIN board at CERN, is the main choice for the ABB boards to be used by the GSI in the CBM experiment and has been used by Hamada during his experiments with the late prototypes of PROGRAPE-4 boards.

Buffer management schemes

Efficient IO transfers alone is not enough to ensure good performance, as it is easy to spend the same time in the transfer as preparing the data to be transferred (or the data received). The Buffer Management library provides a novel approach to the problems of data buffering.

In the past, known buffering schemes like double buffering or pools of buffers were unaware of the characteristics of the data being transferred, as they were implemented

in a generic way. Specific implementations were needed when the characteristics of the data provided or required special treatment. This requires that as new needs arise, the buffer algorithms have to be reimplemented, and if a new scheme is to be used, it has to be ported into every single application.

By properly defining the operations performed, we are able to make the buffer scheme aware of the data structures being transferred while keeping the scheme generic. The buffer management scheme and the data handling can be specified independently of each other. This two operations are composited together into the inherent data-copy loop of the buffer manager, integrating a data copy/transformation operation and reducing the number of data operations needed.

Acceleration of SPH computations

While an SPH accelerator based on FPGAs was already developed in an earlier phase of the project, the software infrastructure presented several challenges, as the previous libraries and drivers could not operate in the intended target system. It was therefore decided that a new software stack was needed to ensure compatibility with newer systems and newer boards in development.

The integration with the application was done as an abstraction layer, where the functionality exported represents tasks to be performed as required by the target application, but taking into account its effect in the performance of the hardware. In particular, it was important that the neighbour lists be assembled easily and that the data transfer be done efficiently, as the performance of the FPGA design is a direct function of the data transfer rate of the board. The buffer management library was created as a solution to these needs, to have constant high speed transfers by means of a double buffer scheme and to transform the data between the format used in the application and the format needed by the accelerator by the use of data translators inserted directly in the buffer manager.

The raceSPH library provides a generic interface to multiple SPH engines in diverse accelerators, but it was designed to interact firstly with the FPGA accelerators. A previous FPGA based design for SPH computations as well as implementations for SSE instructions and GPUs were integrated into a single interface. In total, six implementations were developed in the frame of this work. The library was used and tested with two astrophysical applications, with mixed results. While one presented very good speed-up (5-19x for SPH) and a result in accordance with the expected performance (1.6x-2.4x total time speed-up), the other presented almost no improvement. The difference in the integration of both cases allow us to stress the importance of the hardware/software/application interface when using hardware accelerators, as a mismatched interface will nullify its benefits. This is in itself a success, as we can now identify the potential for acceleration in a code and design new algorithms that use the accelerators effectively.

Acceleration of Gravity computations

The raceGRAV library implements an interface similar to the raceSPH, oriented at the computation of gravitational interactions. While the direct summation of gravity

interactions has already being successfully accelerated with special hardware like the GRAPE-6, this particular implementation aims at working together with these accelerators by using a modified ACS, with the GRAPE-6 accelerating regular forces and the raceGRAV irregular ones. This would be of interest as the irregular forces are based in interaction with local neighbours, not unlike SPH, and because the computational pattern does not profit from the full speed of the GRAPE.

The tests of direct summation in the raceGRAV show that it provides a fraction of the performance of a GRAPE-6 with better accuracy, and comparable or better performance when normalized per pipeline. However, with the publication of several GPU implementations with single and double precision cores as well as higher performance, more study is the current scenario is required before continuing with this line of development.

Chapter 9

Conclusions and Final Remarks

The goal of this work was to use special hardware to increase the performance of actual astrophysical simulations by accelerating the computation of SPH forces as well as the gravitational forces with high accuracy. The rest of the section discusses the achievements documented in this work as well as considerations for future steps, presented as a series of topics and questions.

SPH is accelerated by hardware, is it worth to go faster?

It was shown that the SPH accelerator cores can provide an speed-up of the SPH computations between 6x and 19x for a reference astrophysical simulation, which translates to 1.6x to 2.4x speed-up of the overall wall clock time of the application running in a single node, close to the theoretical maximum set by Amdahl's law at 2.5x. From this, we could conclude that it is not profitable to accelerate the SPH further, as its percentage of utilization is already below 10%. However, additional scenarios need to be considered: What if gravity is computer faster, the number of particle increases, or the application is distributed in a cluster?

If gravity is computed faster, the SPH percentage of the total computational time will increase again, needing a corresponding increase of the SPH accelerator to maintain the same time-fraction distribution. Therefore, faster SPH accelerators are needed to maintain the computational load balanced between tasks inside the same application.

As the number of particle increases, the maximum number of particles allowed per accelerator will be reached. In this case, additional strategies are needed (in the application level) to properly partition the particle sets into computable subsets, so each neighbour for a neighbour list is present in the actual set. An alternative can be borrowed from the SPH implementation of GADGET2, were partial accumulations are done for the same i-particle and later added together. The effectiveness of this approach would need to be validated for the case of hardware accelerators, as GADGET2 uses it for MPI particles across nodes.

If the application is distributed in a cluster, then Gustafson's law let us state that the speed-up of the application will be $S \approx P$ if the serial part of the program is sufficiently small. This also means that as the number of processors increase, so does the absolute benefit provided by the SPH accelerator. It follows that having a faster

SPH accelerator will be beneficial for this case. However, Gustafson's law does not account for network data transfer times, that is a significant factor in the scaling of this simulations, so this point must be studied in further detail.

Another important point is that when integrated with VINE, the result was less than ideal. In contrast with the previous figures, the performance with VINE shows very little improvement. While this can be attributed to the particular format of the tree data in the application, it is worth to study more in detail the interaction with other tree implementations, as the interaction list format is suited for computation in conjunction with a tree walk.

Furthermore, there are other applications that do not require the computation of gravity. SPH is also used to compute fluid dynamics in several other fields, applying them e.g. in the simulation of lava flows or oil spills. In these cases, the computation of the SPH forces is the most significant part of the computing time, so they would benefit directly for a coprocessor that is as fast as possible.

Gravity is accelerated and computed with high accuracy

The main purpose of the raceGRAV library is to support the computation of irregular forces when using an AC integration scheme, because the GRACE accelerator has limited precision. In this context, it would be of benefit. However, its practical use is shadowed by other solutions using GPUs to replace GRAPE boards, as they provide an accuracy comparable to that of the reference CPU. While a core could be programmed to run in a GPU in a similar way as with the raceSPH library, it will then collide with these other solutions, requiring device sharing or a separate GPU to do this task in parallel. It seems more reasonable to extend the existing GPU libraries to include the computation of irregular forces, instead of using a separate accelerator.

The next steps in astrophysical accelerators

The next logical step in the acceleration of the simulations we have focused on is to deal with the generation of the neighbour lists and the use of tree codes. The neighbour list generation was shown in the raceSPH analysis to be the next most demanding task after the computations. In addition, the results from VINE suggests that tree walks might be another potential focus to further improve the performance.

Furthermore, as several tasks are accelerated, possibly by diverse cores, a growing concern is to minimize the communication times. Loading and retrieving data is a significant portion of the time used by accelerators, as shown in the profiling of CUDA cores and the analysis of the FPGA cores of raceSPH. With multiple cores it is necessary to avoid unnecessary transfers. While this can be done manually, a more automatic method of compositing tasks might keep track of this requirement and insert data transfers as needed, facilitating the exchange of tasks between multiple accelerators.

Is there a future for FPGAs in HPC?

From the performance figures presented, one could ask why spend so much effort with the FPGA cores when the GPUs provide such clear advantages? The point to bring

into consideration is that GPUs are relatively new to the High Performance Computing (HPC) world (CUDA was released in 2007), while FPGAs have been growing steadily on a niche segment of it. FPGAs also consume less absolute power than a GPU, which makes them more suitable for big or green installations. When comparing the GFlops per Watt, the GPUs have a clear advantage when considering both devices working at the same efficiency. However, a GPU is not able to reach its peak performance in many but the most optimized applications, and the gather operations needed by a neighbour list in SPH require random accesses to memory that prevent coalesced accesses and drives the efficiency low. FPGAs are, in the other hand, a custom design built for this purpose that does not have this limitation, therefore wasting much less energy.

It cannot be denied that GPUs will play a more central role in scientific computing in the future. FPGAs face three main obstacles to overcome to compete in this area: interconnect speed to host, component density and ease of programmability.

We have already mention the importance of the communication between the accelerator and the host. The GPUs have in this case a clear advantage, with hardware designs that can use the bandwidth of 16 lanes of PCIe v2. In contrast, current FPGAs reach at most 8 lanes at PCIe v1 or 4 lanes at PCIe v2, equivalent to one fourth of the bandwidth usable by a GPU. Using higher speed interconnects like HyperTransport has been suggested to reduce this gap, but the options of motherboards supporting this technology are rather limited.

Component density relates to how many computing units fit in a device. For GPUs, this number is fixed to the number of PEs present, while for FPGAs it is the number of FP units that can be implemented, which varies according to the size of the device, the number of FP operations to include and their chosen precision (and even the implementation used). However, it is clear that a FP operation in a FPGA will be slower and take more area than in a GPU, as those are precisely the advantages of using an ASIC. The exploration of coarse-grained architectures, with building blocks closer to the needs of floating point operators, would be an alternative to balance this obstacle.

The last point to consider is a long standing one. FPGAs are difficult to program, with many attempts been done to reduce the complexity of this task, but it is still closer to hardware design than to actual computer programming. This gap has been reduced with the development of pipeline generators, but they are still very specific solutions, adapted to certain computational patterns like summations. We saw an opportunity in this direction with the introduction of the OpenCL as a platform neutral language for accelerators that provides explicit work distribution (similar to CUDA) and in consequence, started a research line to bring the language to FPGAs.

FPGAs still have a clear advantage when dealing with IO, specially when processing input streams to be loaded into the host, or when processing outgoing streams. They are also competitive for the processing of integer-type data sets. But for the case requiring floating point operations for scientific computing, is the opinion of the author that GPUs will take a leading position in the following years.

Clusters of the future, no more hosts?

With such big portion of the algorithms moving into specialized platforms, it is only logical to ask what will be the role left for the host processors. On the extreme case where e.g. a GPU does all the computing work, the host will only serve as a bridge between the network communication, the saving of data to disk, and the accelerator. This is not an hypothetical question, as several applications using GPUs are already migrating to these arrangement. This questions the reason to have a high-end microprocessor handling this task, which can be achieved equally well by a much smaller and power efficient low-end CPU.

In order to study the advantages and challenges of this approach, a research project together with the Department for Computer Architecture of the Institute of Computer Engineering of the University of Heidelberg (ZITI) is being prepared, in order to replace the host in a first prototype with a minimal system based on a ultra-low power processor and a dedicated network, and with the farther goal of detaching the accelerator from the host altogether.

How will software look like for Exascale computing?

With the introduction of multi-core CPUs and the massively parallel architecture of GPUs, a small revolution is taking place. In the span of only a few years, their use as coprocessors has moved from research institutions to desktop applications and into the operating system itself. In clusters, the nodes can reach several thousands, with some reaching in the tenths of thousands. With the number of cores increasing and GPUs including even more PEs, it is well possible that the number of threads in a program are counted in millions. It is then necessary to ask, how are these heterogeneous machines will be programmed.

Most of the individual parts require special languages, with CUDA used by GPUs, mostly C/C++ or FORTRAN in the CPUs, and VHDL or Verilog in the FPGAs. The communication is also partitioned, with technologies like OpenMP or the Intel Threading Building Blocks being common APIs used to distribute the work among the processors, while Message Passing Interface (MPI) is used to pass data among nodes in a cluster. It is therefore necessary to consider at least four levels of communication for a program to utilize the resources of a modern cluster with GPUs, with inter-node communication handled by MPI, intra-node communication by OpenMP, the node-accelerator transfers handled by CUDA specifics and inter-thread communication inside the GPU done in the CUDA program itself.

It is very possible that we are close to a generational change in software development, where new paradigms and algorithms capable of handling millions of cores will emerge. This is in itself a very general and broad topic, but it can be explored in a more focused way if we restrict it to the realm of astrophysical applications. In order to explore this direction further, a proposal was submitted and granted for the research of domain-specific languages for astrophysical applications, that will allow the continuation of the ongoing collaboration with our colleagues in astrophysics and provide very valuable preparation to fully utilize the capabilities of these new generation of machines as they become available.

Part IV

Appendices

Appendix A

Buffer Manager Profiling Plots

Running a profile for a buffer manager sweeps over the data transfer size and the buffer size, generating one transfer function per buffer size. The following figures provide additional insight on the behaviour of the buffer managers.

Buffer Managers on MPRACE-1

Figures A.1,A.2,A.3,A.4, A.5 and A.6

Templatized Buffers on ABB (Virtex 5)

Figures A.7 and A.8

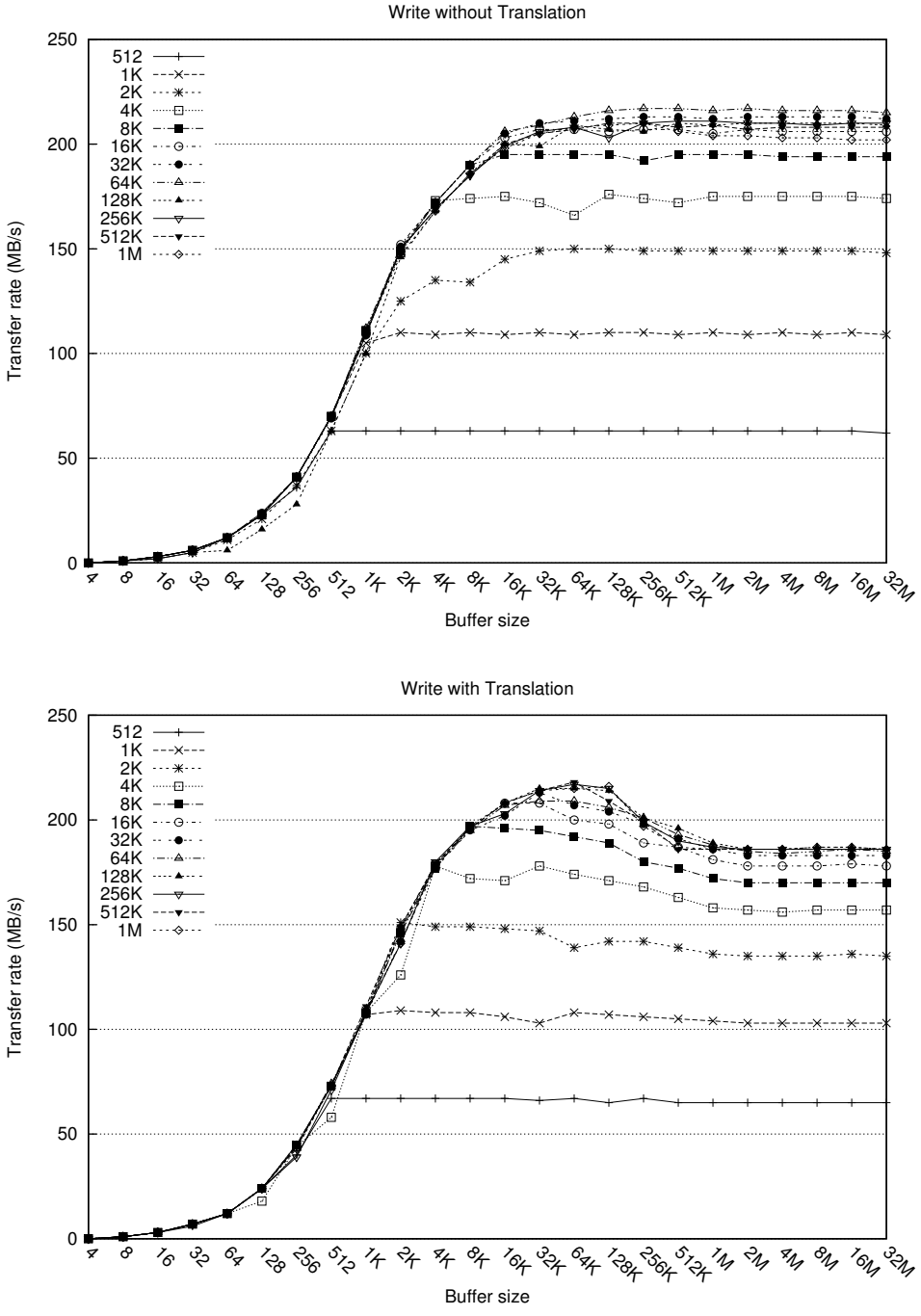


Figure A.1: Chunk Buffer Manager Write, v1

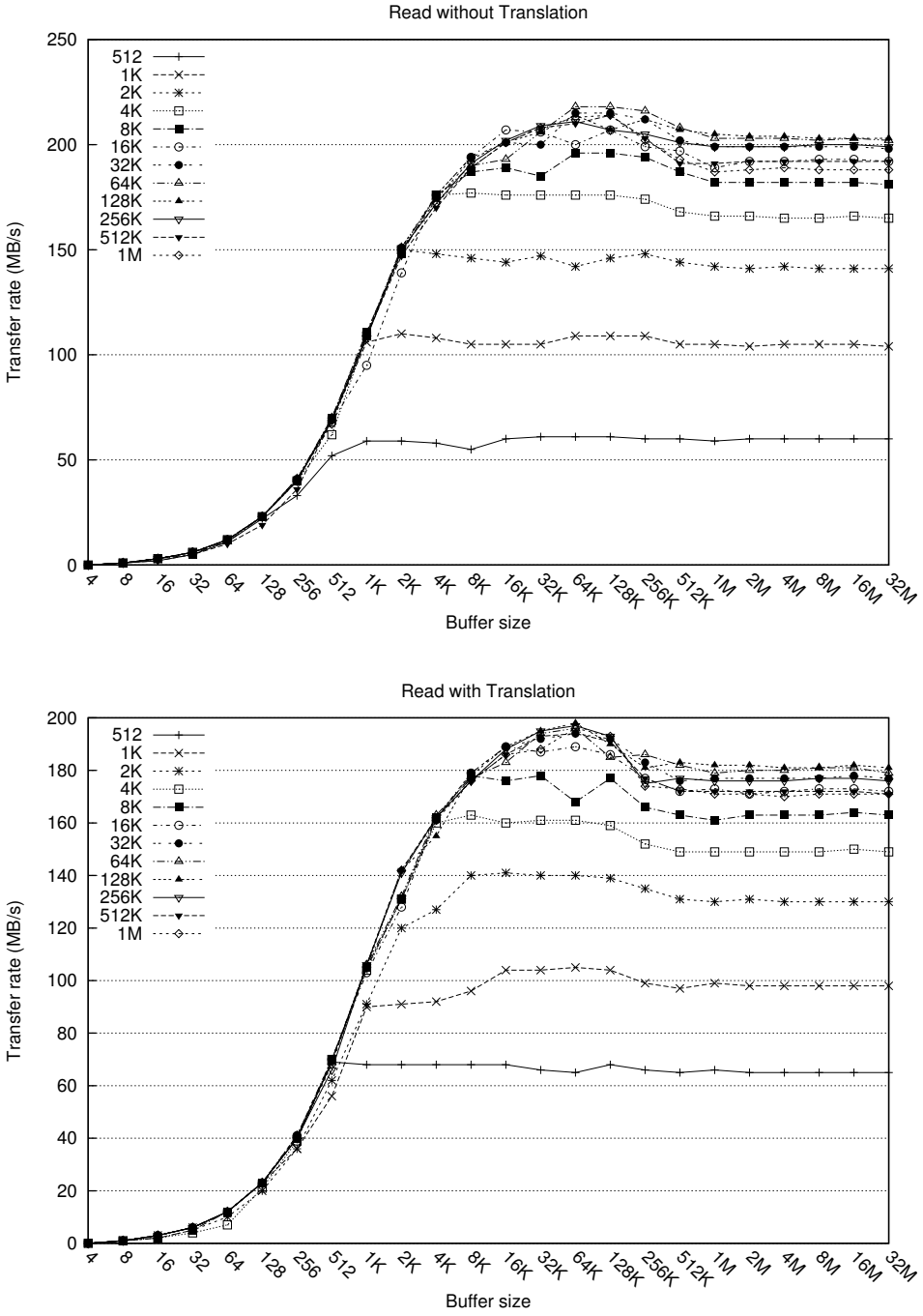


Figure A.2: Chunk Buffer Manager Read, v1

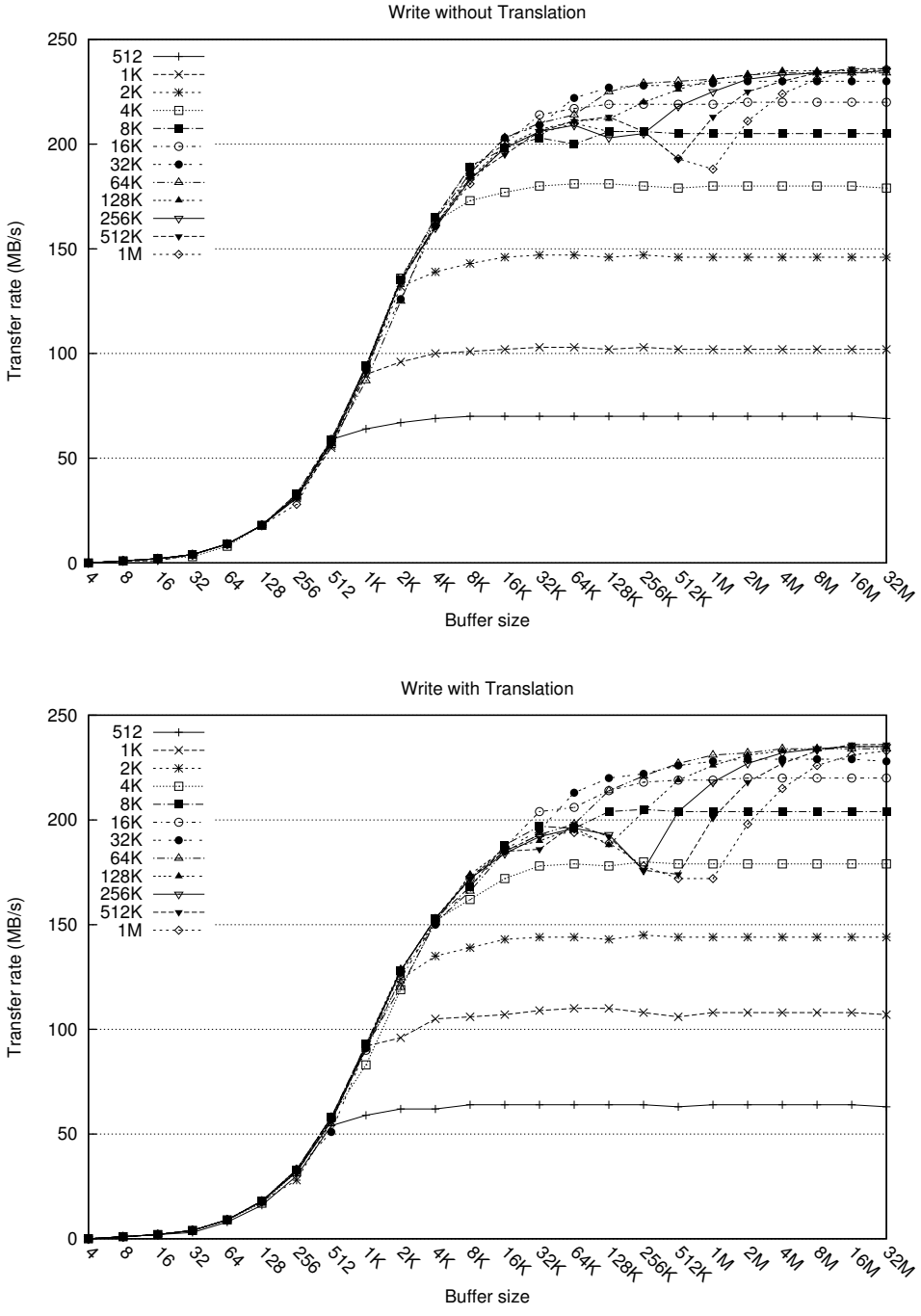


Figure A.3: Double Buffer Manager Write, v1

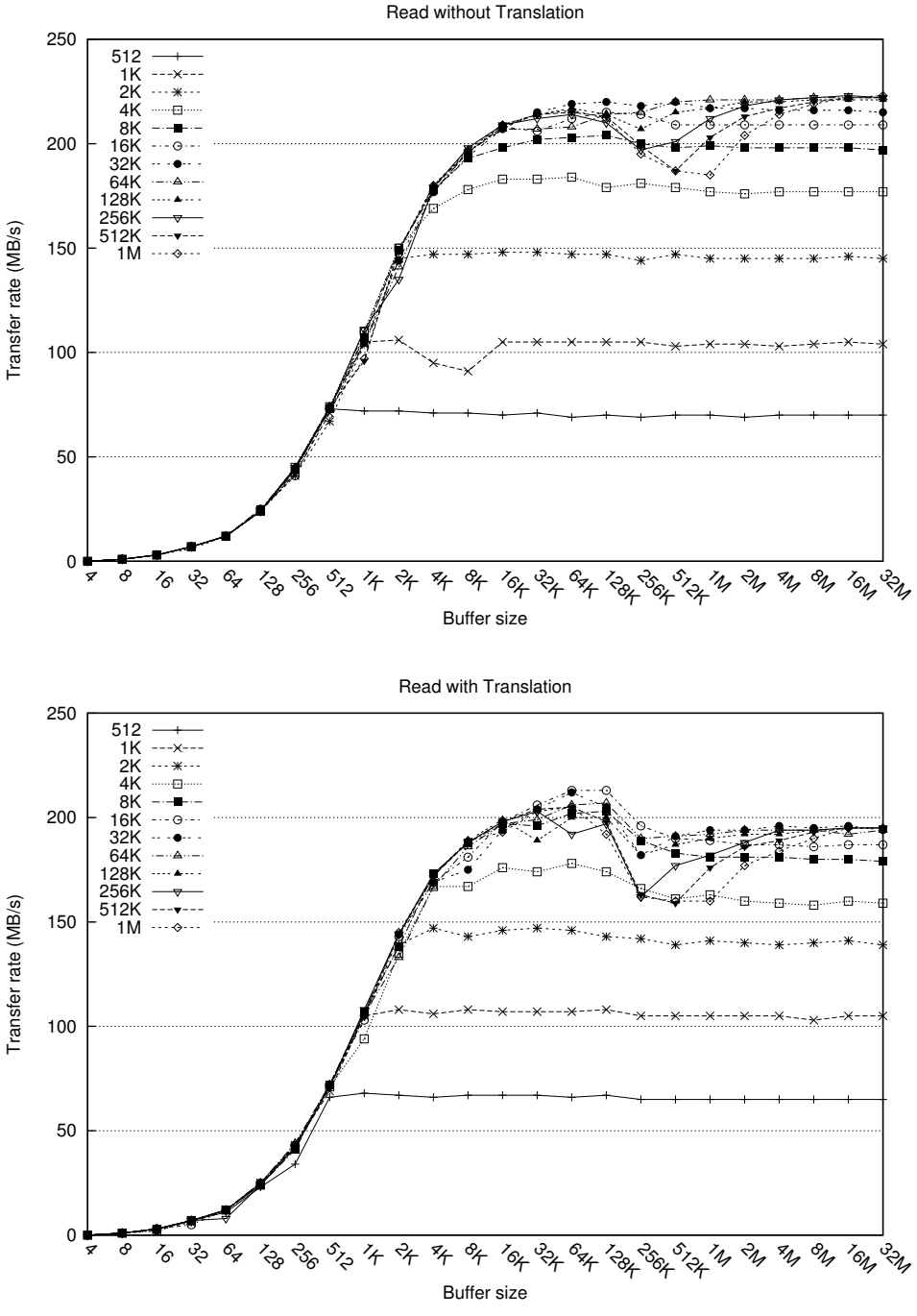


Figure A.4: Double Buffer Manager Read, v1

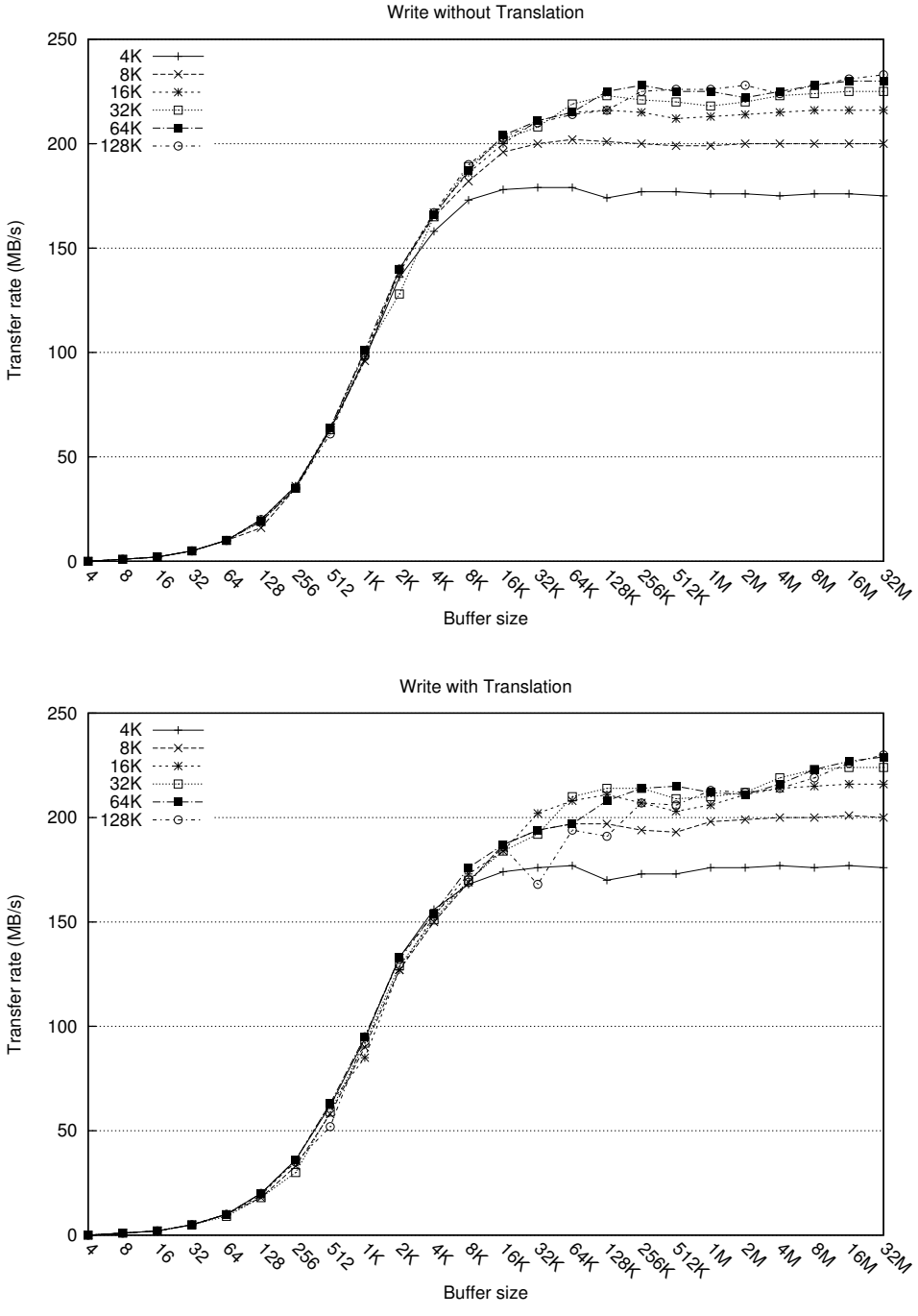


Figure A.5: Double Buffer Manager Write,x4 v1

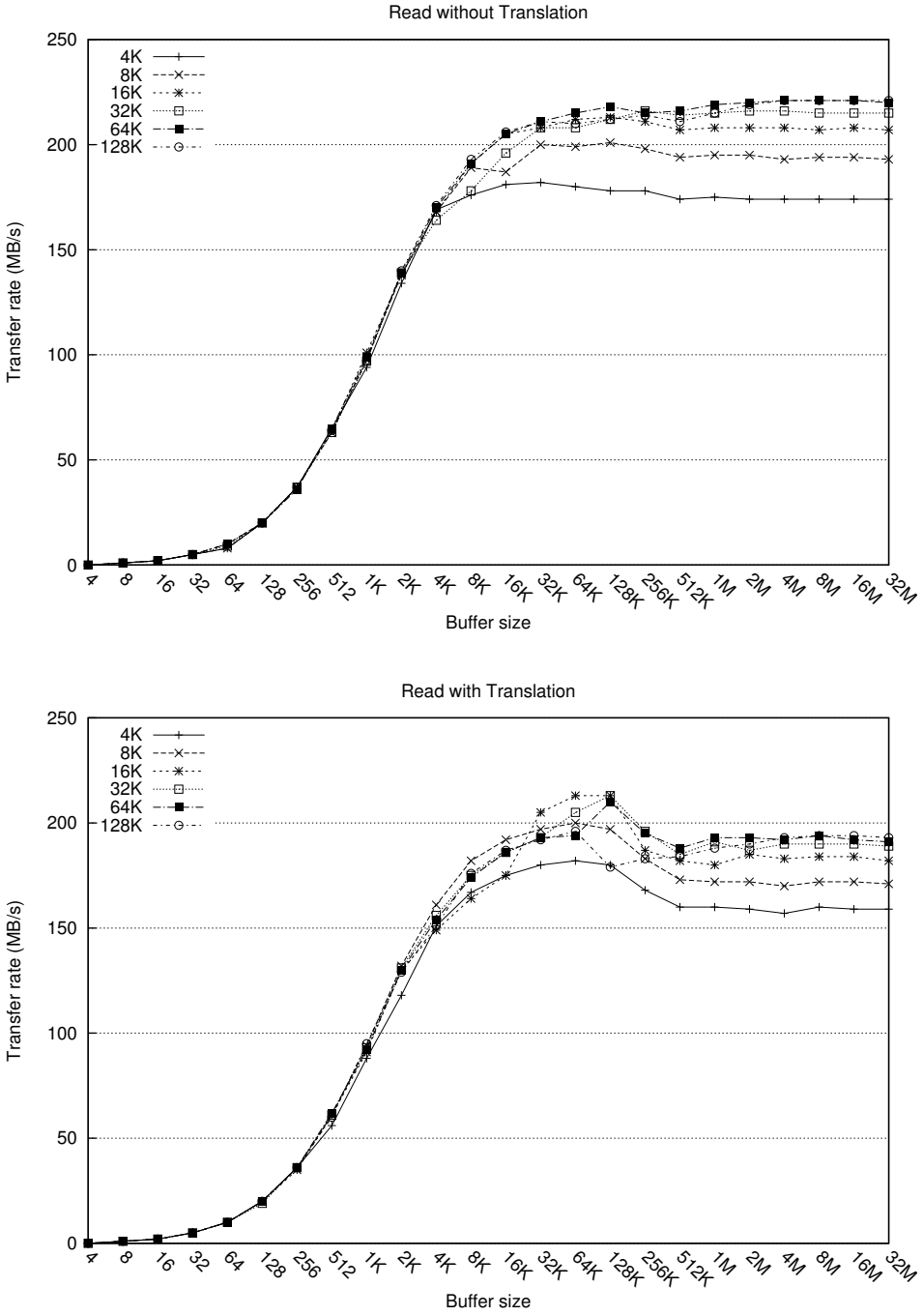


Figure A.6: Pooled Buffer Manager Read,x4 v1

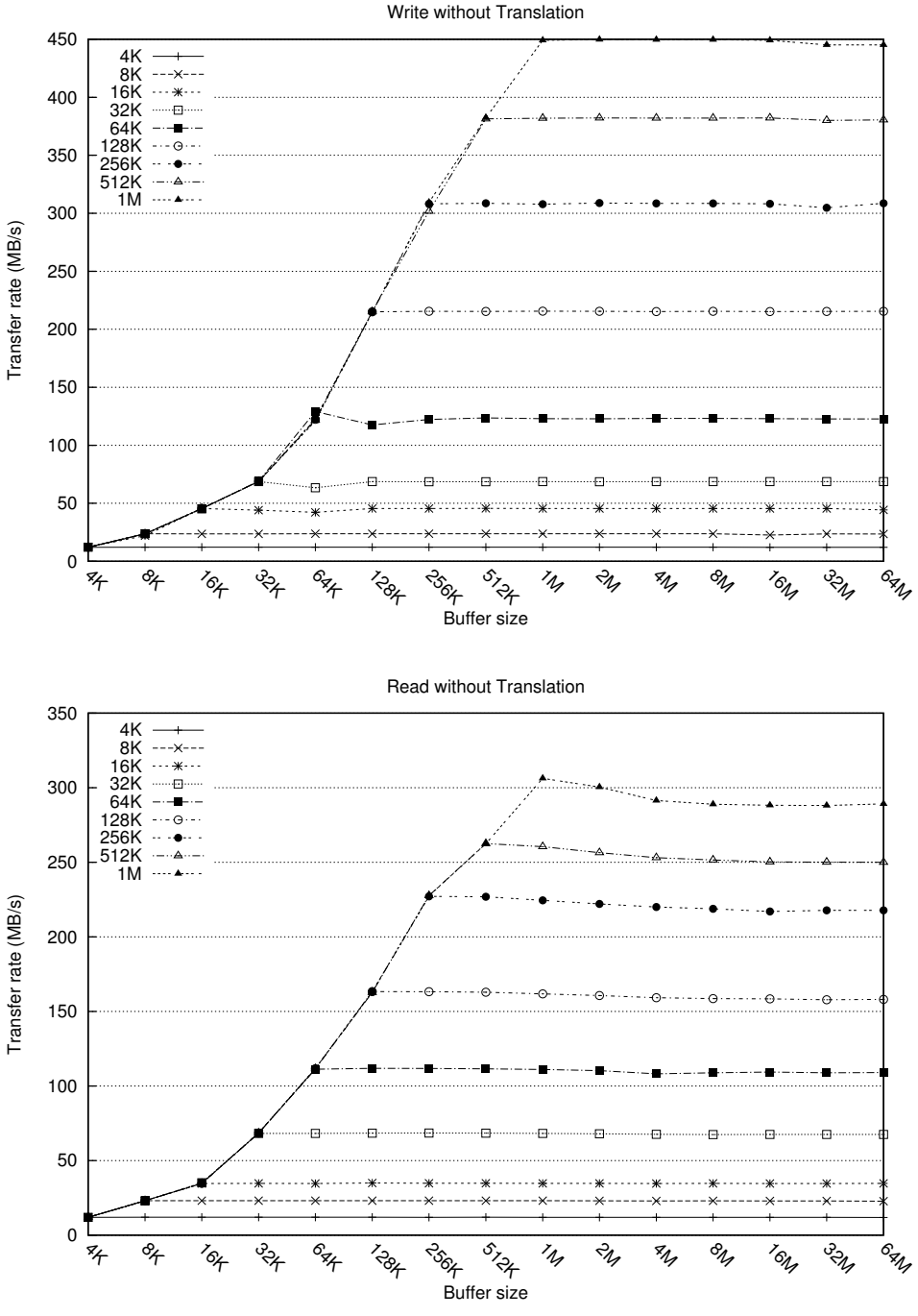


Figure A.7: Chunk Buffer Manager, v2

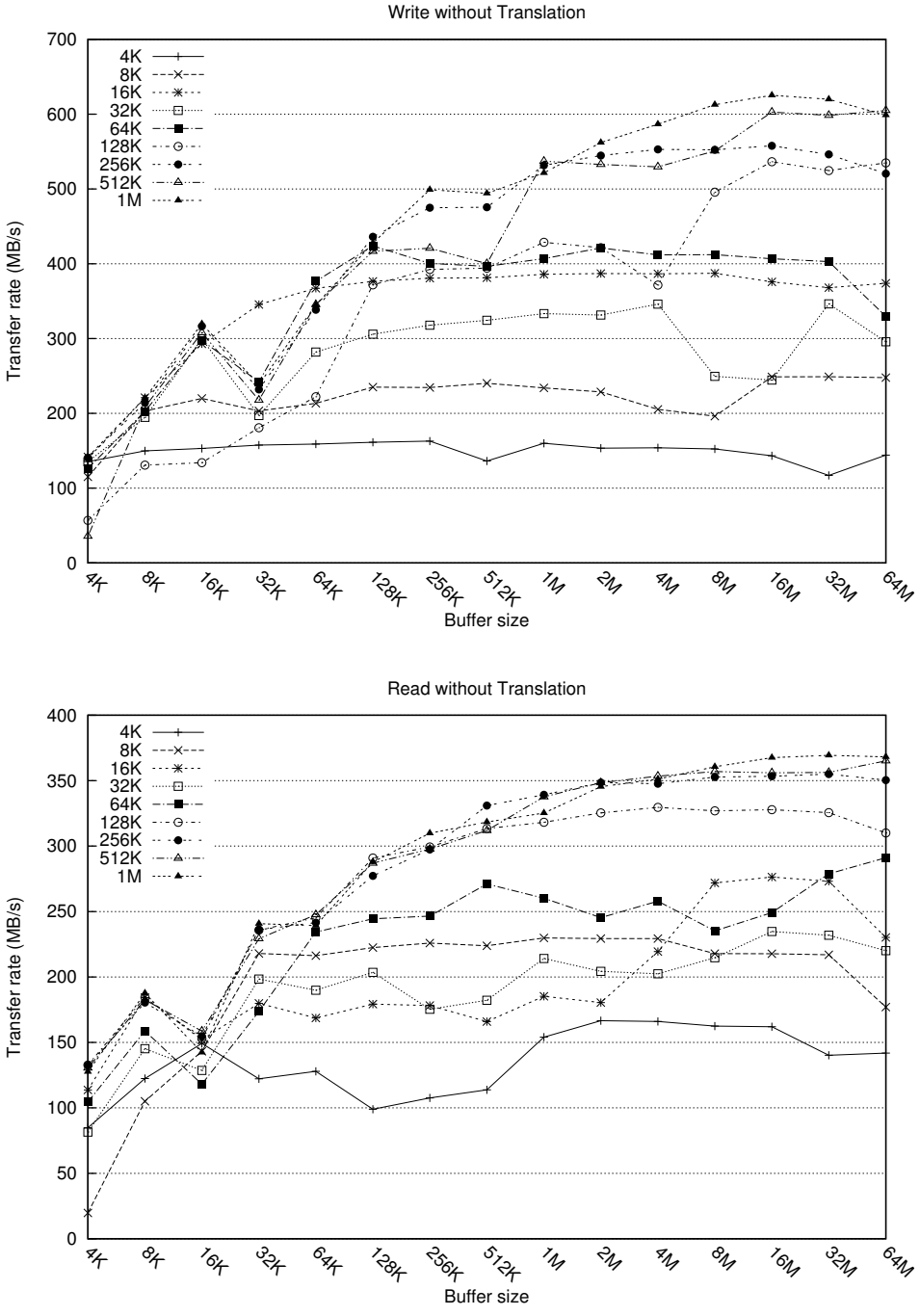


Figure A.8: Double Buffer Manager, v2

Appendix B

RaceSPH Profiling Plots

The following plots cover the three versions of the CUDA core and its performance across several GPUs. Included are versions with CUDA Compute Capability 1.0, 1.1, 1.3 and 2.0

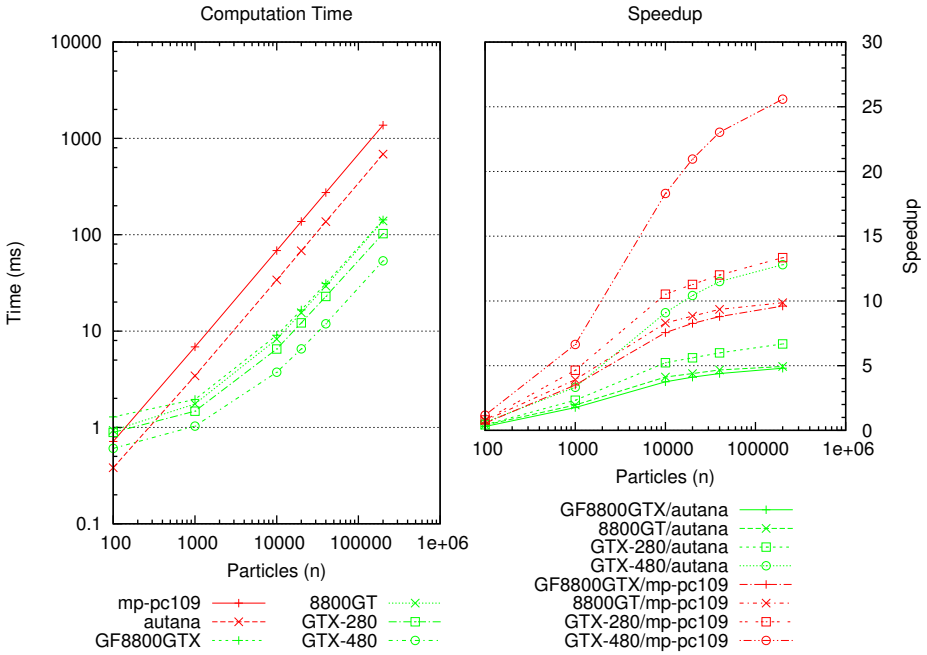


Figure B.1: Performance of CUDA core version 1, compared per GPU

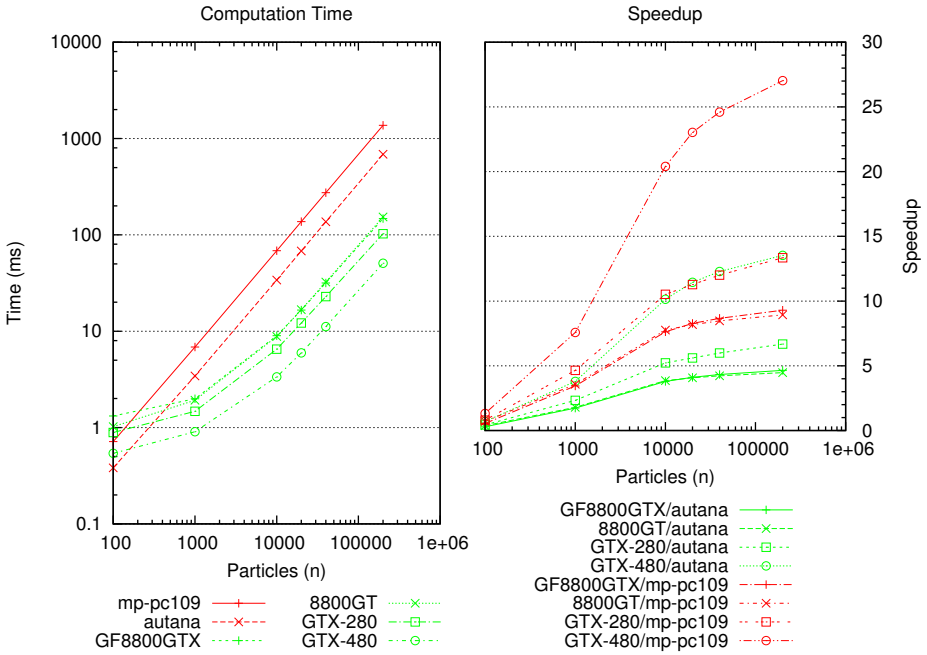


Figure B.2: Performance of CUDA core version 2, compared per GPU

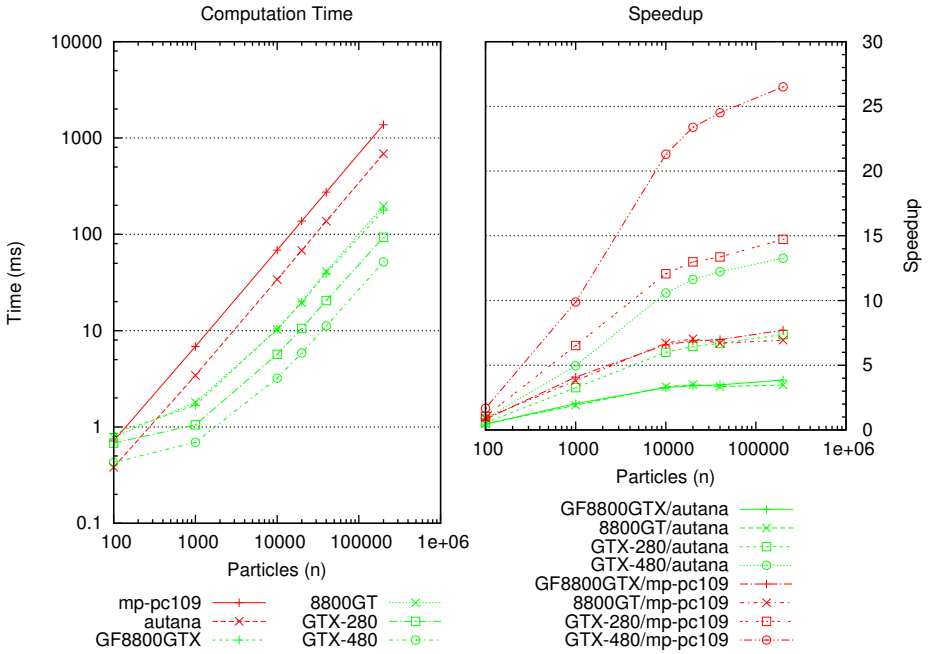


Figure B.3: Performance of CUDA core version 3, compared per GPU

Appendix C

RaceGRAV additional plots

The following plots (Figs. C.1 and C.2) provide additional views of the results presented for the raceGRAV library.

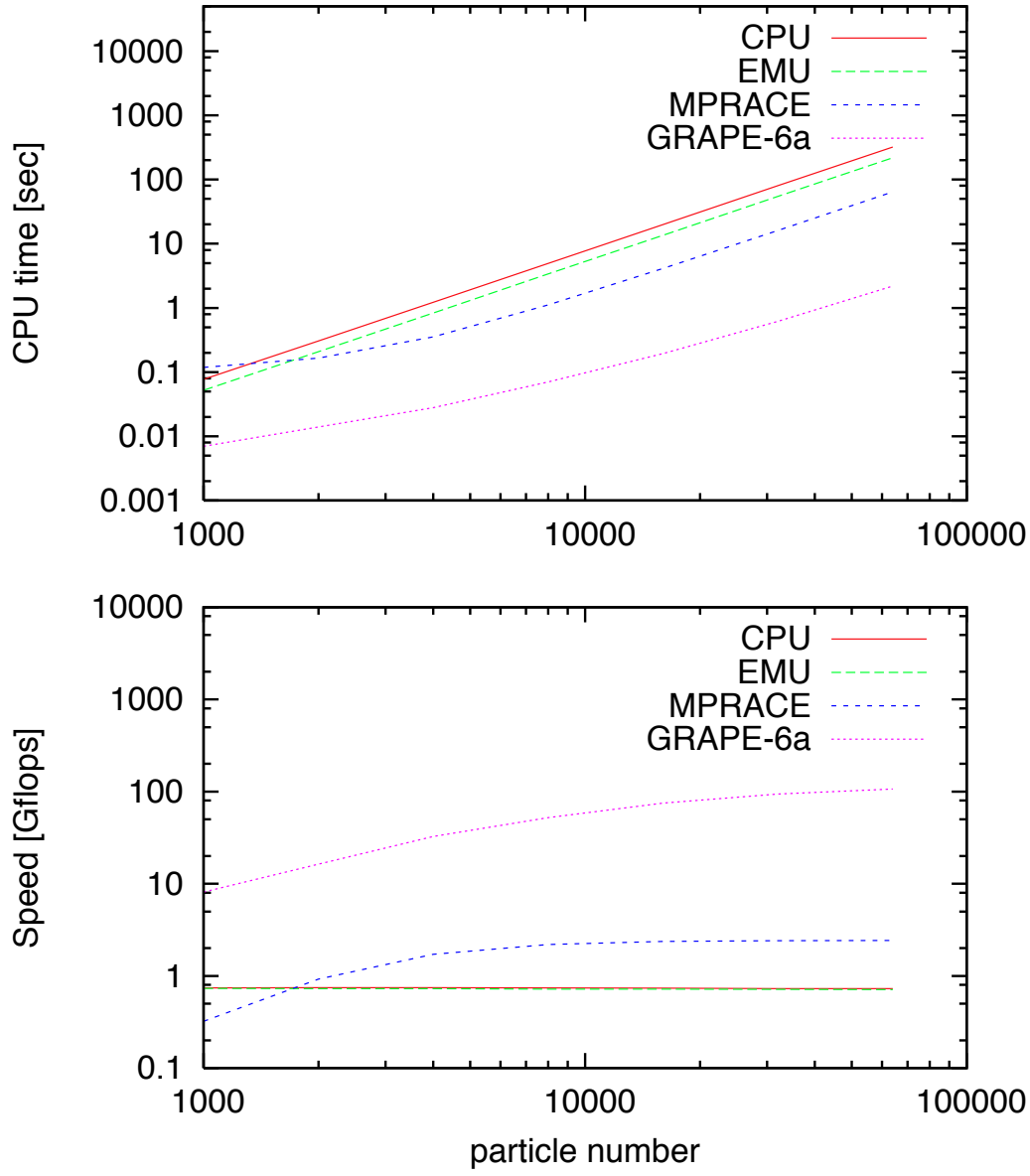


Figure C.1: Performance. Plots by Ingo Berentzen, reproduced with permission

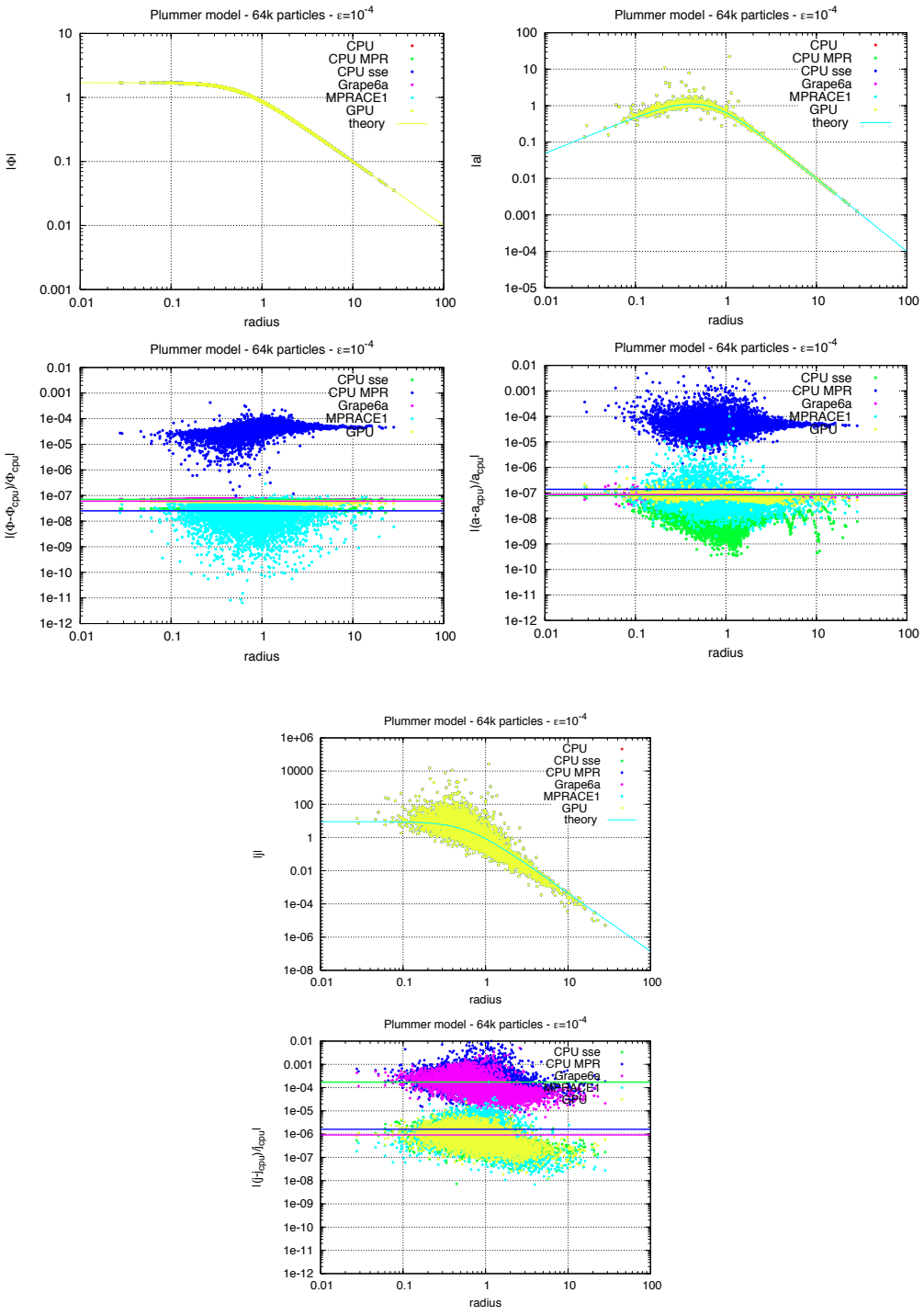


Figure C.2: Results. Plots by Ingo Berentzen, reproduced with permission

List of Figures

List of Tables

Acronyms

ACS Ahmad-Cohen scheme. 119, 129

ARI Astronomisches Rechen-Institut. 17

ASIC Application Specific Integrated Circuit. 44, 45, 120, 133

CC Compute Capability. 34, 111, 122

Cg C for Graphics. 32

CLB Configurable Logic Block. 39, 41, 43, 122

CPU Central Processing Unit. 29, 30

CTM Close-to-Metal. 32, 33

CUDA Compute Unified Device Architecture. 33, 35–38, 134

DDR Double Data Rate. 44

DLL Digital Delay Line. 43

DMA Direct Memory Access. 16

DSP Digital Signal Processor. 43

EDA Electronic Design Automation. 39, 44

EPIC Explicitly Parallel Instruction Computing. 30

FBO Frame Buffer Object. 32

FF Flip-Flop. 39, 41

FP Floating Point. 43, 119

FPGA Field Programmable Gate Array. 9, 16, 38, 39, 41, 43–45, 112, 122

FPU Floating Point Unit. 43

GLSL OpenGL Shading Language. 32

GPU Graphics Processing Unit. 9, 16, 32–38, 46, 112

HDL Hardware Description Language. 39, 41

HLSL High Level Shading Language. 32

HPC High Performance Computing. 133

IOB IO Block. 39, 43

IP Intellectual Property. 44

LUT Look-Up Table. 39, 41, 43

MP Multiprocessor. 33–38

MPI Message Passing Interface. 134

PAL Programmable Array Logic. 38

PCIe PCI Express. 43, 44

PE Processing Element. 33–37, 133, 134

PLL Phase-Locked Loop. 43

s-ASIC Structured Application Specific Integrated Circuit. 44

SIMD Single Instruction, Multiple Data. 31, 97, 106

SIMT Single Instruction, Multiple Thread. 35

SPH Smoothed Particle Hydrodynamics. 9, 15, 17, 22–24, 112, 119

SSE Streaming SIMD Extensions. 96–99, 122, 123

VLIW Very Long Instruction Word. 30

VWF Volkswagen Foundation. 17

ZITI Institute of Computer Engineering of the University of Heidelberg. 134

Bibliography

[1] Electronic.
URL http://www.altera.com/products/devices/hardcopy-asics/about/hrd-index.html

[2] Electronic.
URL http://www.xilinx.com/products/easypath/index.htm

[3] *AVR Microcontroller Architecture.*
URL http://www.atmel.com/products/avr/default.asp?family_id=607&source=redirect

[4] *PIC Microcontroller Architecture.*
URL http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2551

[5] *PLX9656 Datasheet.*
URL http://www.plxtech.com/products/io/pci9656

[6] S. Aarseth. *Gravitational N-Body Simulations. Tools and Algorithms.* Cambridge University Press, 2003. ISBN 978-0-521-43272-6.

[7] D. Adams. *The Ultimate Hitchhiker's Guide.* Wings Books/Random House, 1996. ISBN 0-517-14925-7.

[8] O. Agertz, B. Moore, J. Stadel, D. Potter, F. Miniati, J. Read, L. Mayer, A. Gawryszczak, A. Kravtsov, Å. Nordlund, F. Pearce, V. Quilis, D. Rudd, V. Springel, J. Stone, E. Tasker, R. Teyssier, J. Wadsley, and R. Walder. Fundamental differences between SPH and grid methods. *Monthly Notices of the Royal Astronomical Society*, volume 380:page 963, Sep 2007. doi: 10.1111/j.1365-2966.2007.12183.x.
URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2007MNRAS.380..963A&link_type=ABSTRACT

[9] A. Ahmad and L. Cohen. A numerical integration scheme for the N-body gravitational problem. *J. Comput. Phys.*, volume 12:page 389, Jan 1973. doi:10.1016/0021-9991(73)90160-5.
URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=1973JCoPh..12..389A&link_type=EJOURNAL

[10] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS spring joint computer conference*, pages 483–485, Feb 1967.
URL http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf

[11] P. Bellows and B. Hutchings. JHDL-an HDL for reconfigurable systems. *FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on*, pages 175 – 184, 1998. doi:10.1109/FPGA.1998.707895.
URL http://ieeexplore.ieee.org/search/srchabstract.jsp?tp=&arnumber=707895&queryText%253D%2528%2528jhdl%2529%2529%2526openedRefinements%253D*%2526sortType%253Ddesc_Publication+Year%2526matchBoolean%253Dtrue%2526rowsPerPage%253D50%2526searchField%253DSearch+All

[12] P. Berczik, N. Nakasato, I. Berentzen, R. Spurzem, G. Marcus, G. Lienhart, A. Kugel, R. Männer, A. Burkert, and M. Wetzstein. Special, hardware accelerated, parallel SPH code for galaxy evolution. *Proceedings of the 2nd SPHERIC Workshop*, 2007.
URL http://webs.uvigo.es/spheric/documents/Spheric_Book.pdf

[13] G. Bilotta, A. Hérault, C. D. Negro, G. Russo, and A. Vicari. Complex fluid flow modeling with SPH on GPU. *EGU General Assembly 2010*, volume 12:page 12233, May 2010.
URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2010EGUGA..1212233B&link_type=ARTICLE

[14] I. Buck. High level languages for GPUs. *SIGGRAPH '05: SIGGRAPH 2005 Courses*, Jul 2005.
URL http://portal.acm.org/ft_gateway.cfm?id=1198772&type=pdf&coll=DL&dl=GUIDE&CFID=111712980&CFTOKEN=43270676

[15] I. Buck. GPU Computing: Programming a Massively Parallel Processor. *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, Mar 2007.
URL http://portal.acm.org/ft_gateway.cfm?id=1252526&type=pdf&coll=DL&dl=GUIDE&CFID=111712980&CFTOKEN=43270676

[16] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *SIGGRAPH '04: SIGGRAPH 2004 Papers*, Aug 2004.
URL http://portal.acm.org/ft_gateway.cfm?id=1015800&type=pdf&coll=DL&dl=GUIDE&CFID=111712980&CFTOKEN=43270676

[17] D. Buell, T. El-Ghazawi, K. Gaj, and V. Kindratenko. Guest Editors' Introduction: High-Performance Reconfigurable Computing. *Computer*, volume 40(3):pages 23–27, 2007.
URL http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=pubmed&cmd=Retrieve&dopt=AbstractPlus&list_uids=4133992@ieeejrns

[18] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, 2005. ISBN 978-0-596-00590-0.

[19] Cray. *CRAY-1 Computer System Hardware Reference Manual*.
URL `http://bitsavers.trailing-edge.com/pdf/cray/2240004C-1977-Cray1.pdf`

[20] R. Dalrymple. Particle Methods and Waves, with Emphasis on SPH. *ce.jhu.edu*, Jan 2007.
URL `http://www.ce.jhu.edu/dalrymple/classes/785/Interpolation.pdf`

[21] R. Dalrymple and A. Herault. Levee Breaching with GPU-SPHysics Code. *4th International SPHERIC Workshop*, 2009.
URL `http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?db=pubmed&cmd=Retrieve&dopt=AbstractPlus&list_uids=related:F2GmpSt5oiYJ`

[22] R. Dalrymple and B. Rogers. Numerical modeling of water waves with the SPH method. *Coastal Engineering*, Jan 2006.
URL `http://linkinghub.elsevier.com/retrieve/pii/S0378383905001304`

[23] R. A. Dalrymple and A. Herault. Modeling of Waves with Smoothed Particle Hydrodynamics on the GPU. *American Geophysical Union*, volume 12:page 01, Dec 2008.
URL `http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2008AGUFMOS12B..01D&link_type=ABSTRACT`

[24] F. D. Dinechin, W. Luk, and S. Mckeever. Towards Portable Hierarchical Placement for FPGAs. *INRIA Report*, 1999.

[25] T. El-Ghazawi, K. Gaj, N. Alexandridis, A. Michalski, D. Fidanci, M. Taher, E. El-Araby, E. Chitalwala, and P. Saha. Reconfigurable computers: an empirical analysis (abstract only). *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, Feb 2005.

[26] E. Elsen, V. Vishal, M. Houston, V. Pande, P. Hanrahan, and E. Darve. N-body simulations on GPUs. *Proceedings of ACM/IEEE onference on Supercomputing*, 2006.

[27] T. Fukushige, T. Ito, J. Makino, T. Ebisuzaki, and D. Sugimoto. GRAPE-1A: Special-Purpose Computer for N-body Simulation with a Tree Code. *Publications of the ...*, Jan 1991.
URL `http://adsabs.harvard.edu/full/1991PASJ...43..841F`

[28] T. Fukushige, J. Makino, and A. Kawai. GRAPE-6A: A Single-Card GRAPE-6 for Parallel PC-GRAPE Cluster Systems. *Publications of the Astronomical Society of Japan*, volume 57:page 1009, Dec 2005.
URL `http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2005PASJ...57.1009F&link_type=ABSTRACT`

[29] E. Gaburov, S. Harfst, and S. P. Zwart. SAPPORO: A way to turn your graphics cards into a GRAPE-6. *New Astronomy*, volume 14:page 630, Oct 2009. doi:10.1016/j.newast.2009.03.002.
URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2009NewA...14..630G&link_type=ABSTRACT

[30] W. Gao, A. Kugel, R. Männer, and G. Marcus. PCI Express DMA Engine Design. *CBM Progress Report 2006*, page 54, Sep 2007.
URL http://www.gsi.de/documents/DOC-2007-Mar-137-1.pdf

[31] R. Gingold and J. Monaghan. Smoothed particle hydrodynamics-theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, Jan 1977.
URL http://adsabs.harvard.edu/full/1977MNRAS.181..375G

[32] J. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, volume 31(5):pages 532–533, 1988. doi:10.1145/42411.42415.
URL http://portal.acm.org/citation.cfm?doid=42411.42415

[33] T. Hamada, T. Fukushige, A. Kawai, and J. Makino. PROGRAPE-1: A Programmable, Multi-Purpose Computer for Many-Body Simulations. *Publ. of the Astronomical Society of Japan*, volume 52:page 943, Oct 2000.
URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2000PASJ...52..943H&link_type=ABSTRACT

[34] T. Hamada, T. Fukushige, and J. Makino. PGPG: An Automatic Generator of Pipeline Design for Programmable GRAPE Systems. *Publications of the Astronomical Society of Japan*, volume 57:page 799, Oct 2005.
URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2005PASJ...57..799H&link_type=ABSTRACT

[35] T. Hamada and T. Iitaka. The Chamomile Scheme: An Optimized Algorithm for N-body simulations on Programmable Graphics Processing Units. *eprint arXiv*, page 3100, Mar 2007.
URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2007astro.ph..3100H&link_type=ABSTRACT

[36] T. Hamada and N. Nakasato. Massively parallel processors generator for reconfigurable system. *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*, pages 329 – 330, 2005. doi:10.1109/FCCM.2005.45.
URL http://ieeexplore.ieee.org/search/srchabstract.jsp?tp=&arnumber=1508576&queryText%253D%2528%2528Massively+Parallel+Pro-+cessors+Generator+for+Reconfigurable+System%2529%2529%2526openedRefinements%253D*%2526sortType%253Ddesc_Publication+Year%2526matchBoolean%253Dtrue%2526rowsPerPage%253D50%2526searchField%253DSearch+All

[37] T. Harada, S. Koshizuka, and Y. Kawaguchi. Smoothed particle hydrodynamics on gpus. *Proc. of Computer Graphics International*, Jan 2007.
URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.1665&rep=rep1&type=pdf`

[38] S. Harfst, A. Gualandris, D. Merritt, R. Spurzem, S. P. Zwart, and P. Berczik. Performance analysis of direct N-body algorithms on special-purpose supercomputers. *New Astronomy*, volume 12:page 357, Jul 2007. doi:10.1016/j.newast.2006.11.003.
URL `http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2007NewA...12..357H&link_type=ABSTRACT`

[39] M. Harris. Optimizing Parallel Reduction in CUDA. *NVIDIA CUDA SDK Documentation*, pages 1–38, Nov 2007.

[40] J. Held, J. Bautista, and S. Koehl. From a few cores to many: A tera-scale computing research overview. *Intel Corporation*, 2006.

[41] J. Hennessy and D. Patterson. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann, 1996. ISBN 1-55860-372-7.

[42] A. Herault, G. Bilotta, and R. Dalrymple. SPH on GPU with CUDA. *Journal of Hydraulic Research*, volume 48:pages 74–79, Jan 2010.
URL `http://cat.inist.fr/?aModele=afficheN&cpsidt=22746053`

[43] M. Hill and M. Marty. Amdahl's law in the multicore era. *Computer*, volume 41(7):pages 33–38, 2008.

[44] C. Hinkelbein. *Control Software for Reconfigurable Processors.* Ph.D. thesis, University of Mannnheim, 2005.

[45] F.-H. Hsu. *Behind Deep Blue.* Princeton University Press, 2002. ISBN 0-691-11818-3.

[46] P. Hut and J. Makino. Astrophysics on the GRAPE Family of Special-Purpose Computers. *Science*, volume 283:page 501, Jan 1999. doi: 10.1126/science.283.5401.501.
URL `http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=1999Sci...283..501H&link_type=ABSTRACT`

[47] W. Hwu, S. Ryoo, S. Ueng, J. Kelm, I. Gelado, S. Stone, R. Kidd, S. Baghsorkhi, A. Mahesri, and S. Tsao. Implicitly parallel programming models for thousand-core microprocessors. *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*, pages 754–759, 2007.

[48] Intel Corporation. *Intel C++ Compiler Intrinsics Reference.*
URL `http://cache-www.intel.com/cd/00/00/34/76/347603_347603.pdf`

[49] J. Jones and M. Stettler. Dynamic Reconfiguration and Incremental Firmware Development in the Xilinx Virtex 5. *CERN.*

[50] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Communications of the ACM*, volume 8(1):page 40, Jan 1965. doi:10.1145/363707.363723. URL http://portal.acm.org/citation.cfm?id=363707.363723

[51] G. Kasparov. Garry Kasparov On 'Chess Metaphors': The Chess Master and the Computer. Electronic, Mar 2010. URL http://www.huffingtonpost.com/2010/01/22/gary-kasparov-on-chess-me_n_432043.html

[52] A. Kawai, T. Fukushige, M. Taiji, J. Makino, and D. Sugimoto. The PCI Interface for GRAPE Systems: PCI-HIB. *Publ. of the Astronomical Society of Japan*, volume 49:page 607, Oct 1997. URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=1997PASJ...49..607K&link_type=ABSTRACT

[53] Khronos Group. *OpenCL 1.1 Specification.* URL http://www.khronos.org/opencl/

[54] D. Kirk and W.-m. Hwu. *Programming Massively Parallel Processors.* Morgan Kaufmann, 2010. ISBN 978-0-12-381472-2.

[55] R. Klessen. GRAPESPH with fully periodic boundary conditions - Fragmentation of molecular clouds. *Royal Astronomical Society*, volume 292:page 11, Nov 1997. URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=1997MNRAS.292...11K&link_type=ABSTRACT

[56] Kornmesser. The FPGA Development System CHDL. *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, pages 271 – 272, 2001. doi:10.1109/FPGM.2001.184278. URL http://ieeexplore.ieee.org/search/srchabstract.jsp?tp=&arnumber=1420931&queryText%253D%2528%2528chdl+kornmesser%2529%2529%2526openedRefinements%253D*%2526sortType%253Ddesc_Publication+Year%2526matchBoolean%253Dtrue%2526rowsPerPage%253D50%2526searchField%253DSearch+All

[57] T. Kuberka, A. Kugel, R. Männer, H. Singpiel, R. Spurzem, and R. Klessen. AHA-GRAPE: Adaptive Hydrodynamic Architecture-GRAvity PipE. *Field Programmable Logic and Applications*, pages 417–424, 2004.

[58] G. Lienhart. *Beschleunigung Hydrodynamischer Astrophysikalischer Simulation mit FPGA-Basierten Rekonfigurierbaren Koprozessoren.* Ph.D. thesis, University of Heidelberg, 2004.

[59] G. Lienhart, A. Kugel, and R. Männer. Rapid development of high performance floating-point pipelines for scientific simulation. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006. doi:10.1109/IPDPS.2006.1639439.

URL http://ieeexplore.ieee.org/search/srchabstract.jsp?tp=&arnumber=1639439&queryText%253D%2528%2528Authors%253Alienhart%2529%2529%2526openedRefinements%253D*%2526sortType%253Ddesc_Publication+Year%2526matchBoolean%253Dtrue%2526rowsPerPage%253D50%2526searchField%253DSearch+All

[60] G. Lienhart, G. Marcus, A. Kugel, and R. Männer. Rapid Design of Special-Purpose Pipeline Processors with FPGAs and its Application to Computational Fluid Dynamics. *Field-Programmable Custom Computing Machines, 2006. FCCM '06. 14th Annual IEEE Symposium on*, pages 301 – 302, Apr 2006. doi: 10.1109/FCCM.2006.60.

[61] S. Loo, B. Wells, N. Freije, and J. Kulick. Handel-C for rapid prototyping of VLSI coprocessors for real time systems. *System Theory, 2002. Proceedings of the Thirty-Fourth Southeastern Symposium on DOI - 10.1109/SSST.2002.1026994*, pages 6– 10, 2002.

[62] L. Lucy. A numerical approach to the testing of the fission hypothesis. *The Astronomical Journal*, Jan 1977.
URL http://adsabs.harvard.edu/full/1977AJ.....82.1013L

[63] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn. GPGPU: general purpose computation on graphics hardware. *SIGGRAPH '04: SIGGRAPH 2004 Course Notes*, Aug 2004.
URL http://portal.acm.org/ft_gateway.cfm?id=1103933&type=pdf&coll=DL&dl=GUIDE&CFID=111712980&CFTOKEN=43270676

[64] J. Makino. Optimal order and time-step criterion for aarseth-type n-body integrators. *The Astrophysical Journal*, Jan 1991.
URL http://adsabs.harvard.edu/full/1991ApJ...369..200M

[65] J. Makino. Modified SIMD architecture suitable for single-chip implementation. *eprint arXiv*, page 9278, Sep 2005.
URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2005astro.ph..9278M&link_type=ABSTRACT

[66] J. Makino and S. J. Aarseth. On a Hermite integrator with Ahmad-Cohen scheme for gravitational many-body problems. *PASJ: Publications of the Astronomical Society of Japan (ISSN 0004-6264)*, volume 44:page 141, Apr 1992.
URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=1992PASJ...44..141M&link_type=ABSTRACT

[67] J. Makino, T. Fukushige, M. Koga, and K. Namura. GRAPE-6: Massively-Parallel Special-Purpose Computer for Astrophysical Particle Simulations. *Publications of the Astronomical Society of Japan*, volume 55:page 1163, Dec 2003.
URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2003PASJ...55.1163M&link_type=ABSTRACT

[68] G. Marcus, P. Hinojosa, A. Avila, and J. Nolazco-Flores. A fully synthesizable single-precision, floating-point adder/substractor and multiplier in VHDL for General and Educational Use. *Proceedings of the Fifth IEEE International Caracas Conference on Devices, Circuits and Systems*, Nov 2004. doi: 10.1109/ICCDCS.2004.1393405.

[69] G. Marcus, A. Kugel, R. Männer, P. Berczik, I. Berentzen, R. Spurzem, T. Naab, M. Hilz, and A. Burkert. Accelerating Smoothed Particle Hydrodynamics for Astrophysical Simulations: A comparison of FPGAs and GPUs. *Proceedings of 3rd SPHERIC Workshop*, pages 1–6, Apr 2008.

[70] G. Marcus, G. Lienhart, A. Kugel, R. Männer, P. Berczik, R. Spurzem, M. Wetzstein, T. Naab, and A. Burkert. An FPGA-based hardware coprocessor for SPH computations. *Proceedings of the 2nd SPHERIC Workshop*, pages 63–66, 2007. URL `http://webs.uvigo.es/spheric/documents/Spheric_Book.pdf`

[71] Microsoft. *Programming Guide for Direct3D 10.* URL `http://msdn.microsoft.com/en-us/library/bb205123(v=VS.85).aspx`

[72] J. Monaghan. An introduction to SPH. *Computer Physics Communications*, volume 48(1):pages 89–96, 1988.

[73] J. Monaghan. Smoothed particle hydrodynamics. *Reports on Progress in Physics*, Jan 2005. URL `http://iopscience.iop.org/0034-4885/68/8/R01`

[74] N. Nakasato, T. Hamada, and T. Fukushige. Galaxy Evolution with Reconfigurable Hardware Accelerator. *EAS Publications Series*, volume 24:page 291, Jan 2007. doi:10.1051/eas:2007043. URL `http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2007EAS....24..291N&link_type=ABSTRACT`

[75] N. Nakasato, M. Mori, and K. Nomoto. Smoothed Particle Hydrodynamics with GRAPE and Parallel Virtual Machine. *Astrophysical Journal v.484*, volume 484:page 608, Jul 1997. doi:10.1086/304352. URL `http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=1997ApJ...484..608N&link_type=ABSTRACT`

[76] A. F. Nelson, M. Wetzstein, and T. Naab. Vine—A Numerical Code for Simulating Astrophysical Systems Using Particles. II. Implementation and Performance Characteristics. *The Astrophysical Journal Supplement*, volume 184:page 326, Oct 2009. doi:10.1088/0067-0049/184/2/326. URL `http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2009ApJS..184..326N&link_type=ABSTRACT`

[77] I. Newton. *The Principa Mathematical Principles of Natural Philosophy.* Berkeley: University of California Press, 1999. ISBN 0-520-08817-4. Translation into English by I. Bernard Cohen and Anne Whitman, with help from Julia Budenz.

[78] H. Nguyen, editor. *GPU Gems 3*. Addison-Wesley, 2008. ISBN 978-0-321-51526-1.

[79] K. Nitadori and J. Makino. Sixth- and eighth-order Hermite integrator for N-body simulations. *New Astronomy*, volume 13:page 498, Oct 2008. doi: 10.1016/j.newast.2008.01.010.
URL `http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2008NewA...13..498N&link_type=ABSTRACT`

[80] K. Nitadori, J. Makino, and P. Hut. Performance tuning of N-body codes on modern microprocessors: I. Direct integration with a hermite scheme on x86_64 architecture. *New Astronomy*, volume 12:page 169, Dec 2006. doi: 10.1016/j.newast.2006.07.007.
URL `http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2006NewA...12..169N&link_type=ABSTRACT`

[81] NVIDIA. *CUDA C Best Practices Guide*.
URL `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf`

[82] NVIDIA. *CUDA C Programming Guide*.
URL `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf`

[83] NVIDIA. *CUDA Reference Manual*.
URL `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_Toolkit_Reference_Manual.pdf`

[84] Patterson. A case for intelligent RAM. *Micro, IEEE*, volume 17(2):pages 34 – 44, 1997. doi:10.1109/40.592312.
URL `http://ieeexplore.ieee.org/search/srchabstract.jsp?tp=&arnumber=592312&queryText%253D%2528%2528IRAM%2529+AND+%2528patterson%2529%2529%2526openedRefinements%253D*%2526sortType%253Ddesc_Publication+Year%2526matchBoolean%253Dtrue%2526rowsPerPage%253D50%2526searchField%253DSearch+All`

[85] H.-Y. Schive, C.-H. Chien, S.-K. Wong, Y.-C. Tsai, and T. Chiueh. Graphic-card cluster for astrophysics (GraCCA) Performance tests. *New Astronomy*, volume 13:page 418, Aug 2008. doi:10.1016/j.newast.2007.12.005.
URL `http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2008NewA...13..418S&link_type=ABSTRACT`

[86] V. Schumacher. Zuviel Respekt vor der Maschine. Der Spiegel, May 1997.
URL `http://www.spiegel.de/spiegel/print/d-8716749.html`

[87] A. Shirokov. Gravitational Softening and Adaptive Mass Resolution. *eprint arXiv*, volume 0711:page 2989, Nov 2007.
URL `http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2007arXiv0711.2989S&link_type=ABSTRACT`

[88] V. Springel. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, volume 364:page 1105, Dec 2005. doi: 10.1111/j.1365-2966.2005.09655.x.
URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2005MNRAS.364.1105S&link_type=ABSTRACT

[89] V. Springel, N. Yoshida, and S. D. M. White. GADGET: a code for collisionless and gasdynamical cosmological simulations. *New Astronomy*, volume 6:page 79, Apr 2001. doi:10.1016/S1384-1076(01)00042-2.
URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2001NewA....6...79S&link_type=ABSTRACT

[90] R. Spurzem. Astrophysical N-body simulations: algorithms and challenges. *arXiv*, volume astro-ph, Nov 1997.
URL http://arxiv.org/abs/astro-ph/9711238v1

[91] R. Spurzem, P. Berczik, G. Marcus, A. Kugel, G. Lienhart, I. Berentzen, R. Männer, R. Klessen, and R. Banerjee. Accelerating astrophysical particle simulations with programmable hardware (FPGA and .... *Computer Science-Research and Development*, volume 23(3-4):pages 231–239, May 2009. doi: 10.1007/s00450-009-0081-9.
URL http://www.springerlink.com/content/ew838w1334511061/

[92] M. Steinmetz. GRAPESPH: cosmological smoothed particle hydrodynamics simulations with the special-purpose hardware GRAPE. *Monthly Notices of the Royal Astronomical Society*, volume 278:page 1005, Feb 1996.
URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=1996MNRAS.278.1005S&link_type=ABSTRACT

[93] V.S.Bagad. VLSI Design - Page 5-41. page 416, Jan 2009.
URL http://books.google.com/books?id=g8np-4m2MN4C&printsec=frontcover

[94] W. Warner. Great moments in microprocessor history. Electronic, Dec 2004.
URL http://www.ibm.com/developerworks/library/pa-microhist.html

[95] M. Wetzstein. *WINE – A New Code for Astrophysical Particle Simulations*. Master's thesis, University of Heidelberg, 2000.

[96] M. Wetzstein, A. F. Nelson, T. Naab, and A. Burkert. Vine—A Numerical Code for Simulating Astrophysical Systems Using Particles. I. Description of the Physics and the Numerical Methods. *The Astrophysical Journal Supplement*, volume 184:page 298, Oct 2009. doi:10.1088/0067-0049/184/2/298.
URL http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2009ApJS..184..298W&link_type=ABSTRACT

[97] Xilinx. XC3000 Datasheet. pages 1–76, Feb 1999.

[98] Xilinx. Xilinx DS031 Virtex-II Platform FPGAs: Complete Data Sheet. pages 1–318, Nov 2007.
URL `http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf`

[99] Xilinx. Xilinx UG073 XtremeDSP for Virtex-4 FPGAs User Guide, User Guide. pages 1–121, May 2008.
URL `http://www.xilinx.com/support/documentation/user_guides/ug073.pdf`

[100] Xilinx. Virtex-6 FPGA Extended Overview. pages 1–96, Apr 2009.
URL `http://www.opensparc.net/pubs/preszo/09/brussels/12_MD_Virtex_6_Overview.pdf`

[101] Xilinx. Xilinx UG369 Virtex-6 FPGA DSP48E1 Slice, User Guide. pages 1–50, Sep 2009.
URL `http://www.xilinx.com/support/documentation/user_guides/ug369.pdf`

[102] K. Yoshikawa and T. Fukushige. PPPM and TreePM Methods on GRAPE Systems for Cosmological N-Body Simulations. *Publications of the Astronomical Society of Japan*, volume 57:page 849, Dec 2005.
URL `http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2005PASJ...57..849Y&link_type=ABSTRACT`

[103] S. F. P. Zwart, R. G. Belleman, and P. M. Geldof. High-performance direct gravitational N-body simulations on graphics processing units. *New Astronomy*, volume 12:page 641, Nov 2007. doi:10.1016/j.newast.2007.05.004.
URL `http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2007NewA...12..641P&link_type=ABSTRACT`

[104] J. Zygmont. *Microchip*. Perseus Publishing, 2003. ISBN 0-7382-0561-3.

This page is not intentionally left blank

Acknowledgements

.

To Reinhard Männer, for his support and guidance during the last six years.

To Rainer Spurzem, Peter Berczik and Ingo Berentzen, for they guidance in the field of astrophysics and high performance computing.

To Gerhard Lienhart and Andreas Kugel, for the interesting work in coprocessor design.

To Andrea Seeger and Christiane Glasbrenner, for their everyday support to a newcomer.

To Michael Stapelberg, for his contributions to the PCI driver and the MPRACE library.

To my friends, specially to Erika Fuentes, Carlos Morra and Jesus Zerpa, for all the good times that keep me going.

To all of them, I thank you sincerely for your support during the time that has taken me to write this thesis.