# How to obtain efficient GPU kernels: an illustration using FMM & FGT algorithms

Felipe A. Cruz[a], Simon K. Layton[b], L. A. Barba[b,*]

*[a]Department of Mathematics, University of Bristol*
*[b]Mechanical Engineering Department, Boston University*

## Abstract

Computing on graphics processors is maybe one of the most important developments in computational science to happen in decades. Not since the arrival of the Beowulf cluster, which combined open source software with commodity hardware to truly democratize high-performance computing, has the community been so electrified. Like then, the opportunity comes with challenges. The formulation of scientific algorithms to take advantage of the performance offered by the new architecture requires rethinking core methods. Here, we have tackled fast summation algorithms (fast multipole method and fast Gauss transform), and applied algorithmic redesign for attaining performance on GPUs. The progression of performance improvements attained illustrates the exercise of formulating algorithms for the massively parallel architecture of the GPU. The end result has been GPU kernels that run at over 500 Gop/s on one NVIDIA TESLA C1060 card, thereby reaching close to practical peak.

*Keywords:* fast summation methods, fast multipole method, fast Gauss transform, heterogeneous computing

## 1. Introduction

The reality of the past few years is that the computing industry is betting everything on parallel computing, with first the multi-core era, and now a many-core trend [1]. The implication of this trend is that it will not be enough to develop applications for quad- or eight-core systems, but rather for tens and hundreds of processor systems. The unwelcome news that industry players are delivering is that many-core architectures require "going back to the algorithmic drawing board"[2]. Incrementally thinking about "parallelizing" an existing serial formulation of an algorithm will not work for the many-core architecture. We have to *rethink* the logic of the formulation from the bottom up, and recast the algorithm starting from the mathematics.

Concurrently to the changing situation for CPU technology in recent years, new prominent hardware architectures have come into play. Attracting a tremendous amount of the attention is the programmable GPU (graphics processing unit), with its dramatically faster progress compared to CPUs. Figure 1 shows the computing capacity, measured in Gigaflop/s, of subsequent generations of Intel CPUs and NVIDIA GPUs, as shown in [3] and oft-times cited since. The peak performance of the latest GPU is several times greater than the latest CPU, but more important is the trend. In addition to the raw performance, GPUs have excellent performance-per-watt. As high-performance computing centers are more and more burdened by high power costs, the energy efficiency of the GPU makes it an irresistible choice. The price/performance ratio, finally, makes the GPU architecture a winner in various settings. Given that the market for GPUs was driven by the games industry, it is commodity hardware and
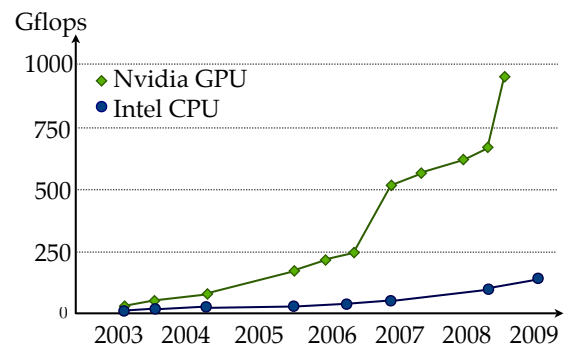


Figure 1: Evolution of the computing capacity of several generations of CPUs and GPUs. Adapted from [3].

thus cheaper than alternatives for high-performance computing based on 'mainframe' servers.

The real opportunity for using GPUs in scientific computing came about with the release of CUDA, a C-language extension that gave programmers relatively easy access to the GPU for coding general algorithms (first publicly released by NVIDIA in February 2007). However, there are significant challenges as the specialized hardware architecture again forces us to go "back to the algorithmic drawing board" [2]. Indeed, there is an immediate need for research into algorithms so that we can exploit the performance offerings of the GPU architecture. This is the motivation of the present work.

In scientific computing, there are multiple algorithms that pervade through a variety of scientific or engineering applications. Here, we focus on two fast summation algorithms —fast multipole method, FMM, and fast Gauss transform, FGT— and present examples of the workflow to produce a formulation of these algorithms that will maximize the performance on the GPU

---
*Corresponding author. Address: 110 Cummington St, Boston MA 02215
*Email addresses:* f.cruz@bristol.ac.uk (Felipe A. Cruz), slayton@bu.edu (Simon K. Layton), labarba@bu.edu (L. A. Barba)

hardware architecture. The algorithms selected in this work program constitute enabling mathematical technology used in many numerical methods. These, in turn, are key tools for enabling specific application areas in science and engineering.

The focus of our research in regards to the applications is in large-scale computational fluid dynamics (CFD) using particle methods, and radial basis function (RBF) methods. For instance, consider the vortex particle method in CFD, where one has a large number of particle-like objects that are used for the mesh-free discretization of the Navier-Stokes equation in vorticity formulation; see [4] for details of the method. The evaluation of the velocity field in this formulation results in an $N$-body problem. In general, the $N$-body problem consists of evaluating all the pair-wise interactions of a set of $N$ bodies due to a force potential, given by:

$$f(x_j) = \sum_{i=1}^{N} c_i \, \mathbb{K}(x_j, x_i). \qquad (1)$$

Here, $\{(c_i, x_i)\}$ represents the set of $N$ source particles with each particle $i$ having a coordinate $x_i$ and a weight $c_i$; and $\mathbb{K}(x_j, x_i)$ represents the kernel function that defines the interactions between two particles. The direct evaluation of Equation 1 at all $N$ particle locations has a computational complexity of $O(N^2)$. It is often the case in problems of scientific interest that the set of particles used in simulations is in the order of thousands to millions. For such large simulations, the direct evaluation of the particle interactions would take an unreasonable amount of computational time. This is one of the main motivations for the use of fast summation methods that are able to accelerate the evaluation of the particle interactions. Such methods reduce the computational complexity of the $N$-body problem to $O(N \log N)$ or $O(N)$, by rapidly computing and aggregating approximated pairwise particle interactions.

Perhaps the most renowned of the fast $N$-body methods is the fast multipole method, FMM, discovered by Greengard and Rokhlin [5]. Its applications have included long-range electrostatics in DNA simulations [6], calculations of vortex sheet evolution [7], room acoustics and scattering from multiple bodies by means of the Helmholtz equation [8], and many others. We will describe the basic aspects of the FMM algorithm in the next section, but in a necessarily brief fashion. Another fast algorithm, which specializes the FMM to the case when the kernel function in (1) is a Gaussian function, is the fast Gauss transform, FGT [9]. This paper covers both the FMM and the FGT, and their implementation on the GPU architecture for achieving maximum performance. After giving a background on the two algorithms that we treat, a motivation for GPU computing follows in §3. The optimization of the algorithmic kernels is described in detail in §4, followed by details of the implementation and discussion of the performance obtained (§5) and final concluding remarks.

## 2. Background on fast summation methods

Of central importance to fast summation methods is the idea that when evaluating pairwise interactions, the solution is only required up to a given accuracy. This idea opens the door to methods that use efficient schemes to rapidly compute an approximation to the interactions, as an alternative to directly evaluating them. It can be said that fast summation methods trade numerical accuracy for computational speed. In this work, we focus on two methods that are classified as analytic-based fast algorithms, as they use analytic approximations to obtain speed and arbitrary accuracy: the Fast Multipole Method, FMM, and the closely-linked Fast Gauss Transform, FGT.

### 2.1. Fast multipole method

The FMM was developed to calculate the pairwise interactions of large sets of particles. Whenever the physical interactions of interest have long-range effects (*e.g.*, gravitational and electrostatic problems), evaluating the interaction among $N$ particles in principle requires $O(N^2)$ operations. The FMM of Greengard and Rokhlin [5] reduces the computational complexity of this problem to $O(N)$ operations by approximating and aggregating the pairwise interactions. A detailed description of the FMM algorithm is outside the scope of this paper. We will only give a brief description of the method and outline the most important steps of the algorithm. For a formal and more detailed description of the FMM algorithm, the reader can consult the original reference [5], or other surveys such as [10].

In a nutshell, the FMM algorithm accelerates the evaluation of all the pairwise particle interactions by approximately evaluating them on clusters of particles. At the core of the algorithm, a hierarchical decomposition of space is used to group particles into clusters at different scales, as shown in Figure 2. In the FMM, the influence of a cluster of particles is approximately represented by a Multipole Expansion (ME), which is an infinite series expansion. The ME is truncated after a given number of terms, enabling the control of the accuracy of approximation. By using the ME, it is possible to evaluate a distant cluster of sources at once and as a whole. The use of the hierarchical decomposition of space and the MEs reduce the computational complexity of the algorithm to $O(N \log N)$. To further reduce the computational complexity to $O(N)$, the FMM introduces the concept of Local Expansions (LES). An LE is a series expansion that converges in a local domain. It is used to evaluate the *aggregated effect of the MEs of a group of clusters within the local domain*. By converting all the MEs that represent the far-field into a single LE, the effects due to interactions with particles in the far-field can be obtained in a single evaluation. Finally, to fully evaluate the domain, the contribution of the interactions with the particles in the near-field are computed directly.

The FMM can be organized into 5 stages:

1. *Initialization*— hierarchical decomposition of the space and generation of clusters of particles.
2. *Upward sweep*— creation of MEs for all clusters.
3. *Downward sweep*— transformation of MEs into LES, and aggregation of LES.
4. *Calculation of the far-field*— LE evaluation.
5. *Calculation of the near-field*— direct evaluation with particles in the near-field.
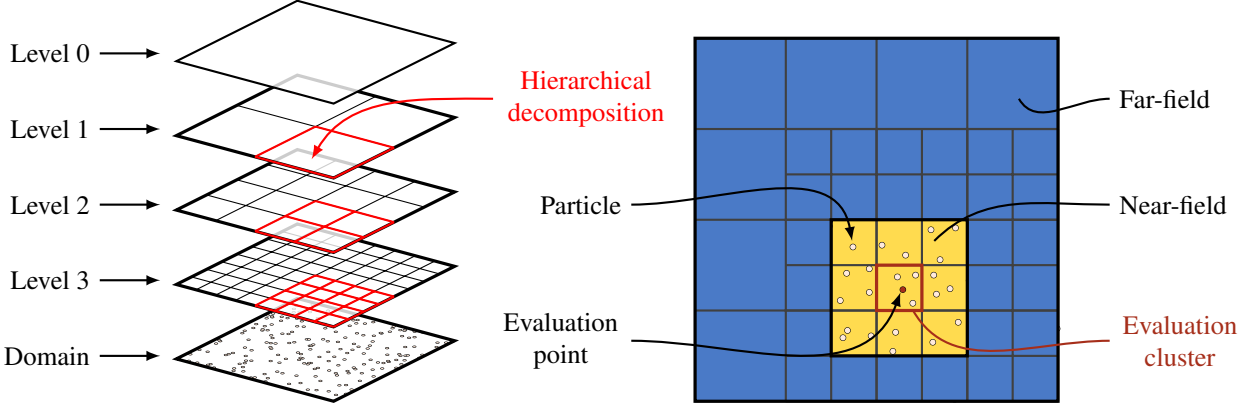
Figure 2: The computational domain is hierarchically decomposed into areas that in turn are used to cluster the particles. Using the hierarchical decomposition, the near-field and far-field for a given target domain are identified. The hierarchy is used to find spatial relations between clusters at different levels. The combination of clusters at different levels are used to approximate the far-field interactions.

Of the stages enumerated above, the most computationally intensive stages are the *downward sweep* and the *calculation of the near-field*. In many experiments with a serial implementation of the FMM, we observed that these two stages can take around 99% of the overall computing time for $N$ in the order of 10 million, and around 95% for $N$ in the order of 4 million. In a parallel implementation, some overheads noticeably appear which reduce the fraction of these stages with respect to the total runtime, although they still substantially dominate. Complete timing breakdowns of a parallel FMM algorithm developed in our group are given in [11]. In the present paper we will address the GPU implementation of one of the two dominating stages of the overall algorithm: the downward sweep. The second one, the calculation of the near-field, is a pleasantly parallel problem and it has already been addressed in [12, 13]. We would like to note here also that a first publication of work on implementing the full FMM algorithm for GPUs was presented in [14].

### 2.2. Fast Gauss transform

The fast Gauss transform (FGT) was originally proposed by Greengard and Strain [9], as a specialization of the FMM, for the case when the interaction function $\mathbb{K}$ from Equation (1) takes the form of a Gaussian kernel, as follows:

$$G(y) = \sum_{i=1}^{N} q_i \exp\left(\frac{-\|x_i - y\|^2}{2\sigma^2}\right) \qquad (2)$$

Similarly to the FMM, the FGT makes use of analytical approximations to reduce the computational complexity from $O(N^2)$ to $O(N)$. The FGT relies on two simple but powerful ideas: first, that Gaussians at a large distance from an evaluation point can safely be ignored with no detriment to accuracy; and second, that multiple individual Gaussians can be combined and approximated using a single series expansion. As with many other fast summation methods, such as the FMM, the computational domain is decomposed in order to leverage these two ideas. In the particular case of the FGT, a decomposition based

on uniform box clusters is used, but the algorithm is agnostic towards the clustering scheme, a useful property in higher numbers of dimensions where simple boxes or cubes may be inefficient. Once the computational domain has been divided, it is necessary to choose what kind of approximation will be used to calculate the potential between any given pair of source and target clusters. There are four distinct options, chosen by work estimates and outlined below; they are illustrated graphically in Figure 2.2.

1. *Direct evaluation*— Use (2) for direct evaluations between individual source and target particles.
2. *Hermite series evaluation*— Generate Hermite expansions from sources around *source* centers and evaluate against individual target particles.
3. *Taylor series evaluation*— Generate Taylor expansions from sources around *target* centers and evaluate against target particles.
4. *Hermite to Taylor series translation*— Generate Hermite series around *source* centers, then translate these series to Taylor series around *target* centers. Finally evaluate these Taylor series at target particles.

We will briefly describe each of these methods in turn, and the situations in which they are used. For a more detailed description of the algorithm, we refer the reader to the original reference [9]. The first form of interaction, the *direct evaluation*, corresponds to applying the original sum of Equation (2). However, rather than evaluating all source points in the computational domain against a given target sub-domain, only the sources within a local sub-domain are considered — by doing so, the entire sum does not devolve into $O(N^2)$ complexity. For this reason, direct evaluation is only used where the number of both sources and targets is small.

In cases where there are a large number of source and target particles, one uses a more efficient way of calculating the potential than direct evaluation, based on series expansions. There exist three different interaction approximations that can be used for this purpose, these are: Hermite series (3), Taylor series
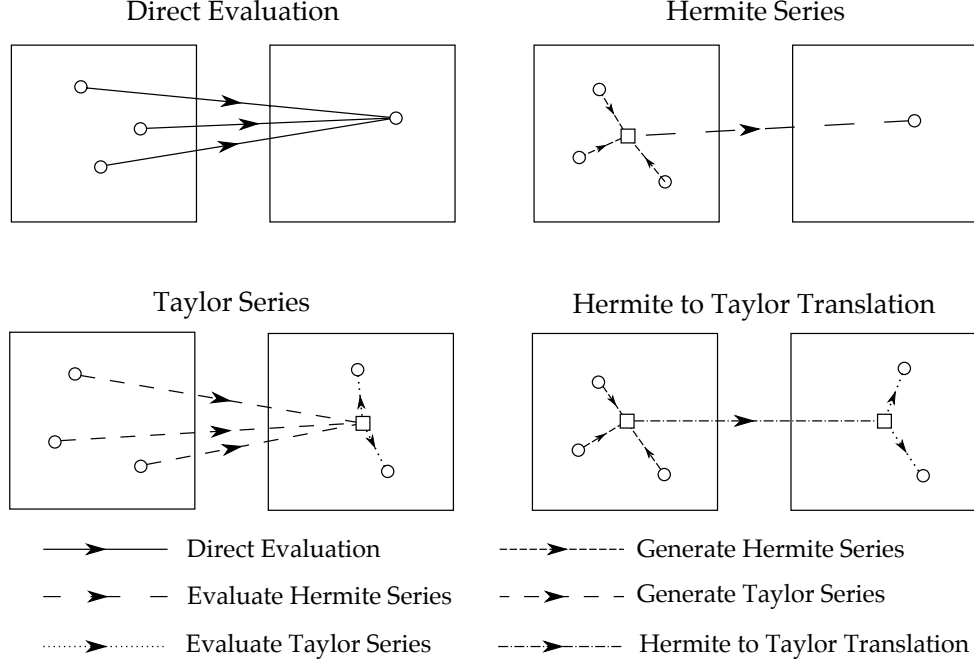
Figure 3: Illustration of the 4 main interactions in the FGT as previously enumerated. Source clusters are on the left, target clusters on the right. Arrows indicate direction of interaction.

(4), or an approach that combines the use of both by translating a pre-existing Hermite series into a Taylor series around a different point (5). In practice, the approximation to be used depends on the computational efficiency of a given situation. For all series, the accuracy can be controlled by choosing a pair of multi-index variables, $\alpha$ and $\beta$, with $h_n(t) = e^{-t^2} H_n(t)$ and $H_n$ denoting the $n$th Hermite polynomial. Furthermore, we use the relation $H_\alpha(t) = H_{\alpha_1}(t) \cdots H_{\alpha_d}(t)$ with the $d$-dimensional multivariate indices $\alpha$ and $\beta$. Other relevant properties of multivariate indices used in the FGT are as follows: $\alpha! = \alpha_1! \cdots \alpha_d!$, $|\alpha| = \alpha_1 + \cdots + \alpha_d$ and $t^\alpha = t^{\alpha_1} \cdots t^{\alpha_d}$.

$$
A_\alpha = \frac{1}{\alpha!} \sum_{j=1}^{N_B} q_j \left( \frac{x_j - s_B}{\sqrt{2}\sigma} \right)^\alpha \tag{3}
$$

$$
B_\beta = \sum_{j=i}^{N_B} q_j \frac{(-1)^{|\beta|}}{\beta!} h_\beta \left( \frac{x_j - t_C}{\sqrt{2}\sigma} \right) \tag{4}
$$

$$
C_\beta = \sum_{\alpha \geq 0} A_\alpha h_{\alpha+\beta} \left( \frac{t_c - s_B}{\sqrt{2}\sigma} \right) \tag{5}
$$

The first series (3) is used in the case where there are many source points but comparatively few targets. In this case, a Hermite series is created about a cluster center, $s_B$, with coefficients $A_\alpha$. The second alternative is a Taylor series about the center of the *target* cluster, $t_C$. This is used when there are a large number of target points, but comparatively few sources. This ensures that time is not wasted generating expensive Hermite expansions when there are not enough sources to justify the cost. Finally, when there is a large number of both sources and targets, one uses a combination of both types of series. First, a Hermite series is created about the center of the *source* cluster

as before, then that series is translated into a Taylor series about the center of the *target* cluster, which can then be evaluated.

Once all the necessary series and translations have been created, they must be evaluated to give the final potential using either (6) for Hermite series, or (7) for Taylor expansions.

$$
G(y) = \sum_{\alpha \geq 0} A_\alpha h_\alpha \left( \frac{y - s_B}{\sqrt{2}\sigma} \right) \tag{6}
$$

$$
G(y) = \sum_{\beta \geq 0} B_\beta \left( \frac{y - t_C}{\sqrt{2}\sigma} \right)^\beta \tag{7}
$$

Using combinations of the different evaluation strategies one can approximately represent the potential field at the evaluation locations. All of the options for evaluating potentials that have been described above are combined to form a fast, efficient algorithm that runs with $O(N)$ complexity. It is notable that all the formulae used are agnostic towards the number of dimensions of the Gaussian basis. Thus, the FGT is particularly useful in fields where problems of this kind occur in high dimensions, as in options pricing [15], information theory [16], image processing [17], and others.

## 3. Motivation: the opportunities of the new hardware architectures

We focus on the CUDA architecture and programming framework from NVIDIA. That said, similar hardware produced by other vendors (such as AMD and INTEL) as well as different programming frameworks (such as OpenCL), rely on the same technology paradigm of massively-parallel and throughput-optimized processors. Therefore, the principles that we discuss

here can be applied generally to current GPU technologies, independent of vendors.

### 3.1. The CUDA architecture and programming model

Developed by NVIDIA, CUDA is a parallel programming model, built for getting the most out of their GPU hardware architecture. This model can involve hundreds of processing cores, and many thousands of individual threads running at any given time. This massive amount of parallelism forces one to focus on developing inherently parallel algorithms in order to get performance. On the other hand, the NVIDIA GPU architecture and CUDA programming model can mitigate some of the complications of parallel programming, many being handled by the hardware and transparent to the programmer.

In this programming model, a parallel program is executed as a *kernel*. These kernels are simply programs written in the C language that will be executed by a single thread. It is generally the case that a kernel will be executed by thousands of simultaneous threads. Each of these threads is extremely lightweight, with no appreciable creation overhead. It also has a unique ID number, with independent control flow and memory space. A further advantage is that active threads can be "hot-swapped" by the GPU. Thus, when a thread is stalled due to a memory access, it can be swapped to another thread with no extra cost. This, combined with the fact that optimal efficiency is gained with tens of thousands of threads, allows us to hide the significant memory latency inherent in the architecture. When a kernel is executed, it is run on a number of threads that is explicitly defined by the user. On the GPU, the threads are handled as groups of *warps* (a 32-thread unit) and all the threads in a warp execute using a single-instruction/multiple-thread model (SIMT; see Table 3.1 for acronyms). Under the SIMT model, all threads will perform the same instruction and follow the same code execution path. Additionally, the SIMT model also allows the threads to branch in their execution but with a reduced execution performance.

To avoid unnecessary computations, the CUDA model allows both shared results and shared memory accesses. These combine to reduce both redundant computations and redundant memory accesses. This kind of cooperation, however, does not scale well to the thousands of concurrent threads that we have already established as the key CUDA methodology. The solution given by CUDA is to limit co-operation to small groups of threads, called *thread blocks*. These blocks are scalable, and allow synchronization between their component threads. A thread block is assigned and executed by a single streaming-multiprocessor (SM). One important characteristic of this model, is that it does not allow for communication between different blocks. This grouping is the model used by CUDA to scale on the hardware with different numbers of available SMs. The more powerful GPUS will have more SMs available and can execute more thread blocks simultaneously.

### 3.2. Key features of the GPU computing technology

The essential hardware features of GPUs are very different from CPUs: they have many times more processor cores, and a

totally different memory system. For example, the main characteristics of the NVIDIA GPU chip used in this study —the GT200, which is used in both the commodity GeForce GTX 2*xx* video cards and the Tesla C1060 computing processor— are the following[1]:

 ▷ 240 'streaming processor' cores per GPU chip, clocked at 1.296 GHz
 ▷ cores grouped into 'streaming multi-processors', with 8 cores each
 ▷ each multi-processor has a *shared memory* of 16 kB

Note the large number of processor cores, and the small amount of *shared memory* of the GPU. The GPU is specialized, highly-parallel hardware, appropriate for computationally-intensive tasks. To be specific, the GPU is a *streaming, pipelined architecture*. It is 'pipelined' because it is built for processing each of many objects in the same way (which is the situation in graphics rendering). The architecture is optimized for *homogeneous units of work*. It is 'streaming' because it allows a large number of simultaneous computational units, with synchronization and communication among these units being provided by the hardware. These fundamental architectural attributes make the GPU exceptionally suited to perform computations that have a high level of data parallelism.

Now, let us consider the performance capability of the GPU described above. To satisfy the reader of the correctness of the marketed performance numbers for this chip, we have to go into a little more detail about the architecture than we would like to. But, if the reader will bear with us, this will prove revealing.

The GT200 chip really has 10 processor clusters, or TPCs (see Figure 4 for a sketch of the GPU hardware architecture). Each of the TPCs has 3 SPMT computing cores (SM), consisting of streaming processors (SP) and special-function units (for transcendentals, *etc*.). An SM can issue, in each cycle, instructions to *either*:

 ▷ 8 single-precision FPUS (*stream processors*)
 ▷ 1 double-precision FPU
 ▷ a branch unit that manages SIMT execution

A *streaming processor* can perform a single precision multiplication and an addition (MAD) in each cycle (2 flop/cycle), thus contributing the following to the performance:

$$1.296 \text{ GHz} \times 10 \text{ TPC} \times 3 \text{ SM} \times 8 \text{ SP} \times 2 \text{ flop/cycle}$$
$$= 622.08 \text{ Gflop/s}.$$

The special function units, meanwhile, can compute 4 floating point operations per cycle in a vector instruction, and thus contribute the following to the performance:

$$1.296 \text{ GHz} \times 10 \text{ TPC} \times 3 \text{ SM} \times 2 \text{ SFU} \times 1 \times 4 \text{ flop/cycle}$$
$$= 311.04 \text{ Gflop/s}.$$

---

[1] Most of the results in this paper, and all of the development, were made on Tesla-series GT200 GPUs. During revision, we gained access to the newest generation GPU (Fermi architecture) and thus we were able to add results obtained on this chip. The discussion in the text, however, does not emphasize Fermi.

| Acronym | Meaning |
|---------|---------|
| SPMD | Single Program, Multiple Data |
| SPMT | Single Program, Multiple Thread |
| TPC | Texture Processor Clusters |
| SM | Streaming Multi-Processors |
| SP | Streaming Processors |
| SIMT | Single-Instruction, Multiple Thread |
| SFU | Special Function Units |
| FPU | Floating-Point Units |

Table 1: Acronyms for GPU architecture.

The sum of the throughput that can be obtained from the streaming processors and the special function units is therefore 933 Gflop/s, the advertised single-precision peak performance of the GPU. It must be noted that this peak assumes that instructions are constantly being co-issued to the SPS and SFUS. When no special functions are required in an algorithm, it is possible to use the special-function unit to execute instead single-precision multiplies, thus in theory the peak throughput is available. However, arguments still arise when various researchers meet to discuss their results as percentage of "peak"; some consider the practical peak to be 622 Gflop/s, for example. In double precision, on the other hand, there is only one FPU available per SM and thus the peak performance of the GPU is: 1.296 MHz × 30 SM × 1 FPU = 78 Gflop/s (the Fermi architecture improved on this considerably).

While this analysis focuses on floating point performance, we are just as interested in overall performance of the GPU, also taking into account integer operations. Thus, we wish to take our analysis based of Gflop/s and try to draw some conclusions about the general performance in Operations/s (Gop/s). From [18], we can obtain the throughput of different arithmetic functions for different datatypes for compute capabilities 1.3 and 2.0. For 1.3-capable cards (such as the Tesla C1060), we see that integer multiply and mutiply-add are listed as 'multiple instructions', while for 2.0 compute capability (such as the Fermi C2050), the throughput is identical for that of equivalent floating point operations. From this, we conclude that for the C1060 peak performance in Gop/s is slightly lower than the value determined above in Gflop/s, but for the C2050 cards, performance in floating point and integers should be equivalent.

Clearly, there will be many challenges in a particular application for extracting the promised performance out of the GPU. The throughput capacity calculated above refers to the *maximum* number of instructions that can be issued on the chip. Moreover, the *actual* throughput will also depend on issues related to the access to memory, in particular the high-latency accessess to global memory from the GPU chip (where memory transactions range between 400 and 600 clock cycles), and the use of shared memory in the streaming multi-processors to mitigate the high-latency effects.

For the algorithms of interest in this work, we have developed formulations that demonstrate the power of *heterogeneous computing* using GPUs. The subsequent section will discuss the work flow that was followed for the formulation of the algorithmic kernels in such a way as to maximize performance in the architecture.

## 4. Formulation of the algorithmic kernels for the GPU and optimization

When formulating algorithms for the GPU, one wants to have the means to analyze their performance and efficiency in terms of resource utilization for the target architecture. By performing such analysis, we can predict the maximum theoretical performance of the algorithm on the given hardware, and use this information as a guide during the design and optimization of the algorithm.

Before any analysis can take place, we need to build a simplified model of the GPU that characterizes all the important features of the architecture. From our introduction in section §3.2, a current-generation GPU can be described as a *massively parallel architecture* that has been *optimized for high throughput*. In our simplified model, we consider each GPU to have hundreds of very simple processors (processing elements) that can execute concurrently. We will assume that each processing elements is capable of computing the following operations, in single precision and at a constant cost: arithmetic operations ($+, -, \times, \div$, transcendental functions), and conditional statements. Regarding the data movement operations (load, store, and copy), we will not take into account all the details of the GPU memory hierarchy but use a simplified model where we distinguish between off-chip memory (global memory) and on-chip memory (registers and shared memory)—the main considerations are the limited bandwidth available and limited capacity, respectively.

Using the simplified GPU model, we can estimate the operation complexity and data accesses of an algorithm. We use this information to evaluate the algorithmic performance against the hardware theoretical *throughput* and *bandwidth*.

▷ *Throughput (measured in op/s):* The average number of operations per second that can be executed by the GPU. A large number of op/s implies a high level of theoretical peak performance. Traditionally, this number is presented as Giga-op/s, abbreviated as Gop/s. Giga-flop/s or Gflop/s are also commonly presented, but using op/s gives a better idea of how close to the peak performance of the GPU an algorithm obtains.
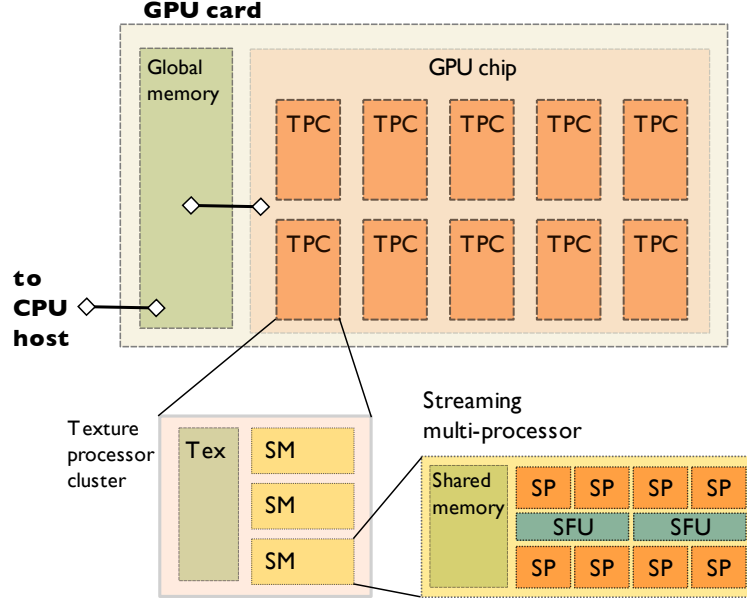
Figure 4: Details of GT200 GPU architecture.

▷ *Bandwidth (measured in Giga-Bytes/sec, GB/s):* The rate at which data is transferred between the memory and the processor. Bandwidth measures how much data is being read and written from global memory in a unit time. In practice, many applications are bandwidth-limited, therefore it is important to use the bandwidth as efficiently as possible.

In addition to our GPU model, we define four algorithmic properties that map to different features of the GPU paradigm. These properties are then used as guidelines when rethinking an algorithm:

▷ *Computational intensity*: ratio of operations performed to memory accesses in a unit of time. A compute-intensive task will have an operations-to-memory access ratio larger than the one given by the theoretical performance of the GPU. Highly compute-intensive tasks benefit from the high throughput of the GPU, while low compute-intensive tasks benefit from the large memory bandwidth of the GPU.

▷ *Concurrency*: parts of the algorithm that *may* be executed simultaneously. There are many levels of concurrency that range from coarse-grained concurrency to fine-grained concurrency. The latest GPUs can benefit from multiple levels of concurrency. For instance, when using CUDA-enabled devices, one is able to explicitly identify concurrency at different levels, coarse- to fine-grained: CUDA kernel, thread-block, and thread calculations.

▷ *Homogeneity*: degree at which concurrent computations are the same, independently of their input. Concurrent tasks that present a high degree of homogeneity can take full advantage of the 'single-instruction /multiple-thread' SIMT model of the GPU architecture.

▷ *Data-locality*: the way in which the physically stored data is accessed by an algorithm. The role of data locality is twofold: a high degree of spatial data locality can result in more efficient data accesses, *e.g.*, when data that is spatially adjacent can be accessed by thread collaboration, resulting in more efficient memory transactions. A high degree of temporal data locality avoids redundant data transfers of data values that are frequently accessed within small time windows.

An ideal algorithm for the GPU would be computationally intensive and have a high degree of concurrency. Concurrent computations would be highly homogeneous and show a large amount of data locality. However, in practice these properties are difficult to measure and as such do not allow one to effectively quantify the expected performance of the algorithm under study.

### 4.1. Fast multipole method for the GPU

We will now concentrate on the algorithmic redesign of the FMM for GPU computing, whereas a more in-depth discussion of the implementation details takes place in §5. As such, we start by noticing that the most time-consuming stages of the FMM algorithm are: the near-field calculations, and the downward sweep. As we observed in §2.1, these stages can take 95 – 99% of the runtime in a serial implementation of the algorithm, and continue to dominate in parallel implementations despite parallel overheads.

Let us discuss in more detail the computations performed by these two stages. In the case of the near-field calculations, the problem is to compute the mutual interactions of particles in many small- to medium-size domains, with each domain containing at most a few hundred particles. Therefore, the near-field calculation problem can be reduced to efficiently implementing the direct particle interactions for each domain, and as such, this is a special case of the *N*-body problem, where interactions are computed only with a subset of the *N* particles.

Furthermore, the near-field calculation retains the distinctive features of being computationally intensive and highly parallel. Therefore, this problem can be efficiently mapped to the GPU architecture, achieving high performance, as reported in [12, 13]. We note that, as pointed out by [12], a balanced FMM execution in the GPU will perform the near-field direct evaluation with a larger set of particles per box than in the CPU case. This difference is due to the higher efficiency achieved by the GPU in the near-field calculations, as compared to the far-field calculations. Consequently, the optimal workload distribution between these two stages is different in the GPU and CPU executions.
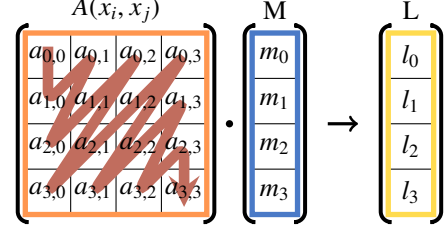
In the downward sweep, the computations are dominated by the transformation of multipole expansions into local expansions (M2L step). Tens of thousands of concurrent M2L transformations are computed in the FMM algorithm, this being the reason for such a high computational intensity. Each of these transformations can be expressed in the form of a matrix-vector multiplication, where each matrix is dense, of size ($p \times p$), where $p$ corresponds to the number of terms of a truncated ME. The algorithmic steps of the downward sweep are:

1. For every node of the hierarchical tree, we obtain the list of clusters that belong to the same level, effectively computing the interaction list of each node.

2. Then for every node (or evaluation cluster) in the tree, the multipole expansions of each cluster in its interaction list are transformed into a local expansion centered at the evaluation cluster.

3. Finally, the cluster's local expansion is obtained by aggregating the local expansions obtained from the M2L transformation of all the clusters in the interaction list, this step is referred to as the *reduction*.
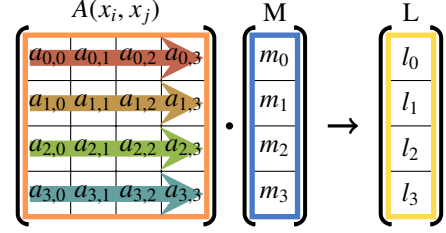
We present the following example to illustrate the parallelism and computational intensity of the downward sweep stage. Consider a two-dimensional case ($d = 2$) where the computational domain is hierarchically decomposed by a uniform tree (a quadtree) with $l = 5$ levels of spatial refinement. Such a tree will approximately have $4^l$ nodes, where each node will transform on average 27 multipole expansions (more precisely, this number corresponds to the size of the node's interaction list). Therefore, the approximate number of M2L transformations computed are $4^5 \times 27 = 27,648$. For deeper trees, more transformations would be needed.

In the original algorithm for the M2L transformation, the multipole expansion of a given cluster $j$, with center $x_j$ and multipole coefficients $m_j$, is transformed into a local expansion for cluster $i$ with center at $x_i$. The result of the transformation will give the local expansion coefficients $l_i$. In the 2-dimensional case, both the coordinates and expansion terms are usually represented by complex numbers. The M2L transformation in its matrix-vector multiplication form has a transformation matrix $A$, with $p \times p$ complex elements, $a_{nk}$. These elements can be obtained using the following expression:

$$a_{nk} = (-1)^n \binom{n+k}{k} (x_i - x_j)^{-k-n-1} \tag{8}$$



(a) Traversing the matrix diagonals.



(b) By rows.

Figure 5: Sketch of different strategies for M2L computation.

As previously noted, the M2L stage is highly parallel and computationally intensive. However, in order to obtain a high-performance implementation for the GPU, the M2L stage needs to be reformulated.

### 4.1.1. Reformulation of the M2L problem

As a first approach, one can consider distributing the mat-vec operations across threads. That is, each CUDA-thread would perform one such operation. Suppose that the evaluation requires a truncation parameter $p = 12$, as required for high-accuracy simulations; see for example [19]. In that case, each matrix has a size of $2p^2 = 288$, requiring 2304 bytes of storage. Thus, in the GPU shared memory of 16 kB one could fit a maximum of 6 such matrices, which implies that a maximum of 6 threads could be run in one multi-processor at a time. Clearly this number of threads is much too small—as discussed in §3.2, more than one hundred threads per SM are needed in order to hide memory transfer latencies. Alternatively to storing the transformation matrix in shared memory, it is feasible to use Equation (8) to compute the matrix elements as the matrix-vector multiplication takes place (also known as matrix-free format).

When performing the matrix-vector product in matrix-free format, one can "traverse" the matrix such that the matrix terms are efficiently calculated by reusing the computations performed to obtain the previous terms. Therefore, one can reuse a part of both the binomial and the power terms by traversing the diagonals of the matrix, as sketched in Figure 5(a). This represents a strategy for matrix creation and multiplication that is efficient from the point of view of operations that it performs, in both the GPU and CPU architectures. Moreover, for the GPU, by assigning the computations of one matrix-vector multiplication to only one CUDA-thread, thread synchronization is not required. Consider now that each thread block performs the transformations of only one ME for all the interaction list, accounting for an estimated 27 M2L transformations. During computation, the

results of the transformation are stored in shared memory. With a maximum of 8 concurrent thread blocks per multi-processor in the GT200 GPU, we have a maximum of $27 \times 8 = 216$ active threads running on the multiprocessor. This NVIDIA GPU allows a maximum of 512 threads in each thread block and 1024 active threads per SM [3, p. 8], and thus the approach just described could achieve less than one third of the maximum thread capacity of each SM, without considering the register usage per thread. When the translation of an ME is finished, one needs to move the result from shared memory to global memory, in the form of one LE consisting of $2p$ floats. Hence, transforming one ME for all the interaction list results in the movement of $27 \times 2p$ floats. For instance, translating one ME of $p = 12$ results in 648 floats that need to be stored in global memory, which at the hardware level results in a minimum of twenty 128-byte and one 32-byte memory transactions (see §5.3 for a more complete discussion on memory management). When transferring the results to global memory, the memory transactions are executed by the thread-block one after the other, resulting in a stage where only memory transactions take place. A considerable disadvantage of such a memory-intensive stage is that multithreading is less effective in hiding the memory latency when the number of active threads is small. Nevertheless, our implementation using this approach sustained 20 Gop/s and was able to perform 2.5 million translations per second, using a single C1060 TESLA card.

The performance of the implementation just described is very much below the theoretical peak for both throughput and bandwidth of a C1060 TESLA card. An assessment of the CUDA kernel implementation revealed that its main limitation was the inefficiency of the memory transfers to and from global memory. This was due to two problems: first, many of the memory transfers were non-coalesced (i.e. adjacent threads reading adjacent memory values, see §5.3 for a full explanation), thus the *effective bandwidth* used by the kernel was very low when compared against the theoretical performance of the card; and second, the kernel did not have enough active threads working per SM to effectively hide the latency of the memory transfers, making the efficiency of the kernel implementation suffer. These two problems can be tracked down to a flaw in the design of our algorithm: using only one CUDA thread to perform a single translation. The resulting kernel was resource-intensive since the diagonally traversing matrix-free multiplication required too many variables to store the state of the computations and to control the flow of the program, thus limiting the total number of active threads per SM. Additionally, having only one thread to perform each translation made it difficult to use multiple threads to perform collaborative memory transactions.

Desirable features of a more optimized kernel would be for it to be: (*i*) *simpler*, requiring less variables to store the state of its execution, thus allowing more threads to be active at a time; (*ii*) *more parallel*, allowing the use of more concurrent threads, and to use coordinated collaboration between threads for performing memory transfers.

Now, let us examine again Equation (8). A more parallel alternative to traverse the matrix is to have multiple CUDA threads assigned to traversing different rows of the matrix, as each row can be computed concurrently with each other; see Figure 5(b). As with the diagonal-traversal but to a more moderate extent, the row-traversal can also be implemented so that it reuses a part of both the binomial and the power terms.

Let us consider for a moment a straightforward implementation to obtain the powers in Equation (8) shown in the code fragment in Figure 6. What characterizes this simple approach is that the number of iterations per thread will depend on $(k+n+1)$, which takes a different value for every thread. This straightforward implementation would have two problems: first, threads within a thread-block naturally stop at different points provoking the threads within a warp to diverge; second, loop unrolling would be unavailable as the compiler can only effectively unroll loops with known trip counts.

In our implementation, we separated the computation of powers in Equation (8) into two steps in order to minimize thread divergence, and to use compiler optimizations such as loop unrolling. In the first step of the calculation, the term $(x_i - x_j)^{n+1}$ is computed by each thread; and in the second step, the next $k$ $(x_i - x_j)$ factors, common to all threads, are multiplied. When computing the first step, we use a for-loop but with a fixed number of iterations ($P - 1$, with $P$ the variable containing the truncation level of the ME) and an *if* conditional for computing only up to the power relevant to the thread; see the code fragment in Figure 7. This approach minimizes thread divergence at the cost of a small overhead of evaluating a few extra empty iterations. However, the overhead is minimal when compared to the performance that would be lost due to thread divergence. In contrast to the first step, the second step consists of homogeneous computations for all threads, therefore it can be easily implemented using a for-loop with a known trip of $k$. The second step achieves high performance as it can be loop-unrolled and further optimized by the compiler. For a discussion on loop unrolling see §5.4.

### 4.1.2. Discussion of the M2L GPU implementation and results

One advantage of having more available threads per thread-block, is that all the memory transactions that take place to and from global memory can be organized so that threads in the same warp access sequential memory locations. This allows the GPU to optimize the memory transfers by grouping together memory transactions to the same segment in global memory. In the row-traversal kernel, there are two cases when accessing global memory: the first case occurs when loading the multipole expansion coefficients from global memory to shared memory. The usage of a sequential memory layout of the ME coefficients allows efficient memory transactions by warps of threads. The second case occurs when the threads finish computing one local expansion coefficient, and here each thread directly saves the state into global memory. This global write is sequential between the threads, as at any given time a sequential number of coefficients are being computed by a sequential set of threads.

From a resource utilization point of view, each thread uses a small amount of resources: the shared memory usage is limited to storing the ME information that is read at the start of the

```
1    int k = 12;
     int n = threadIdx.x;
3
     float ac, bd;
5    tkn1_r = t_r;
     tkn1_i = t_i;
7
     for (m = 1; m < k+n+1; m++)  // tkn1 = tkn1 * t
9    {
         ac       = tkn1_r;
11       bd       = tkn1_i;
         tkn1_r = ac * t_r - bd * t_i;
13       tkn1_i = ac * t_i + bd * t_r;
     }
```

Figure 6: Code snippet for computing the complex power $t^{k+n+1}$, with $t = t_{\Re} + i * t_{\Im}$. This code shows a for-loop with a variable number of iterations.

```
1    #define P 12
3    int m;
     float ac, bd;
5    float tn1_r = t_r;
     float tn1_i = t_i;
7
     #pragma unroll
9    for (int m = 1; m < P; m++)
     {
11       if (threadIdx.x >= m) // tn1 = tn1 * t
         {
13           ac     = tn1_r;
             bd     = tn1_i;
15           tn1_r = ac * t_r - bd * t_i;
             tn1_i = ac * t_i + bd * t_r;
17       }
     }
```

Figure 7: Code snippet for computing the complex power $t^{n+1}$, with $t = t_{\Re} + i * t_{\Im}$. This code shows a for-loop where the number of iteration has been fixed at compile time to $P - 1$, and it uses a conditional for computing only the relevant power to each thread.

| # of terms | # of trans. | M2L kernel [seconds] | Reduction [seconds] | Data HtD [seconds] | Data DtH [seconds] | OPS. [Giga] | B. [GB/s] | Mill. trans. per sec |
|---|---|---|---|---|---|---|---|---|
| 8 | 2160 | 5.79e-05 | 3.48e-05 | 6.70e-05 | 3.10e-05 | 125.27 | 2.60 | 37.28 |
| 8 | 9072 | 9.30e-05 | 5.51e-05 | 1.07e-04 | 4.01e-05 | 327.82 | 6.81 | 97.57 |
| 8 | 36720 | 2.69e-04 | 1.41e-04 | 3.59e-04 | 8.99e-05 | 458.77 | 9.53 | 136.54 |
| 8 | 147312 | 9.66e-04 | 4.91e-04 | 1.27e-03 | 2.80e-04 | 512.35 | 10.65 | 152.49 |
| 8 | 589680 | 3.69e-03 | 1.91e-03 | 4.40e-03 | 8.29e-04 | 537.36 | 11.17 | 159.93 |
| 8 | 2359152 | 1.44e-02 | 7.58e-03 | 1.65e-02 | 3.10e-03 | 548.72 | 11.40 | 163.31 |
| 12 | 2160 | 7.41e-05 | 3.48e-05 | 6.91e-05 | 2.69e-05 | 185.97 | 2.93 | 29.13 |
| 12 | 9072 | 1.60e-04 | 5.79e-05 | 1.17e-04 | 4.41e-05 | 362.02 | 5.71 | 56.71 |
| 12 | 36720 | 5.11e-04 | 1.45e-04 | 4.09e-04 | 1.28e-04 | 458.60 | 7.24 | 71.84 |
| 12 | 147312 | 1.90e-03 | 5.17e-04 | 1.37e-03 | 4.02e-04 | 493.93 | 7.79 | 77.37 |
| 12 | 589680 | 7.44e-03 | 2.01e-03 | 4.87e-03 | 1.14e-03 | 506.32 | 7.99 | 79.31 |
| 12 | 2359152 | 2.95e-02 | 8.00e-03 | 1.78e-02 | 4.51e-03 | 511.06 | 8.06 | 80.05 |
| 16 | 2160 | 9.39e-05 | 3.48e-05 | 7.30e-05 | 2.91e-05 | 236.93 | 3.03 | 22.99 |
| 16 | 9072 | 2.26e-04 | 5.70e-05 | 1.31e-04 | 5.20e-05 | 413.58 | 5.28 | 40.14 |
| 16 | 36720 | 7.68e-04 | 1.52e-04 | 4.31e-04 | 1.62e-04 | 492.69 | 6.29 | 47.82 |
| 16 | 147312 | 2.94e-03 | 5.36e-04 | 1.47e-03 | 5.15e-04 | 515.93 | 6.59 | 50.07 |
| 16 | 589680 | 1.16e-02 | 2.08e-03 | 5.19e-03 | 1.55e-03 | 524.79 | 6.70 | 50.93 |
| 16 | 2359152 | 4.61e-02 | 8.30e-03 | 1.89e-02 | 5.85e-03 | 526.90 | 6.73 | 51.14 |

Table 2: Results of the multipole-to-local computation on the NVIDIA TESLA GPU. Each row entry of the table presents the results of a single test run. The description of the columns follows, from left to right: number of terms computed, number of translations performed, GPU execution time for M2L kernel, GPU execution time for reduction, time for data transfer from host to device (HtD), time for data transfer from device to host (DtH), number of giga-operations per second (OPS.) for M2L kernel, effective bandwidth (B.) utilization for M2L kernel, metric of M2L translations per second performed (in millions).

| # of terms | # of trans. | M2L kernel [seconds] | Reduction [seconds] | Data HtD [seconds] | Data DtH [seconds] | OPS. [Giga] | B. [GB/s] | Mill. trans. per sec |
|---|---|---|---|---|---|---|---|---|
| 8 | 2160 | 3.79e-05 | 2.10e-05 | 5.20e-05 | 2.19e-05 | 191.45 | 3.98 | 56.98 |
| 8 | 9072 | 7.61e-05 | 3.81e-05 | 9.92e-05 | 2.41e-05 | 400.79 | 8.33 | 119.28 |
| 8 | 36720 | 2.52e-04 | 1.25e-04 | 2.93e-04 | 4.89e-05 | 489.58 | 10.17 | 145.71 |
| 8 | 147312 | 9.41e-04 | 4.96e-04 | 8.46e-04 | 1.32e-04 | 526.11 | 10.93 | 156.58 |
| 8 | 589680 | 3.70e-03 | 2.02e-03 | 2.71e-03 | 5.79e-04 | 535.66 | 11.13 | 159.42 |
| 8 | 2359152 | 1.47e-02 | 8.19e-03 | 9.33e-03 | 2.55e-03 | 538.79 | 11.20 | 160.35 |
| 12 | 2160 | 5.10e-05 | 2.22e-05 | 4.72e-05 | 1.91e-05 | 270.27 | 4.26 | 42.34 |
| 12 | 9072 | 1.34e-04 | 4.01e-05 | 9.70e-05 | 2.79e-05 | 432.23 | 6.82 | 67.71 |
| 12 | 36720 | 4.87e-04 | 1.27e-04 | 2.65e-04 | 5.70e-05 | 481.27 | 7.59 | 75.39 |
| 12 | 147312 | 1.87e-03 | 5.03e-04 | 1.23e-03 | 2.98e-04 | 502.67 | 7.93 | 78.74 |
| 12 | 589680 | 7.42e-03 | 2.04e-03 | 2.86e-03 | 7.26e-04 | 507.56 | 8.01 | 79.50 |
| 12 | 2359152 | 2.96e-02 | 8.31e-03 | 9.34e-03 | 2.24e-03 | 509.26 | 8.03 | 79.77 |
| 16 | 2160 | 6.60e-05 | 2.22e-05 | 5.39e-05 | 2.00e-05 | 337.01 | 4.31 | 32.71 |
| 16 | 9072 | 1.88e-04 | 3.89e-05 | 1.17e-04 | 3.29e-05 | 497.56 | 6.36 | 48.29 |
| 16 | 36720 | 7.05e-04 | 1.31e-04 | 3.41e-04 | 7.70e-05 | 536.68 | 6.86 | 52.08 |
| 16 | 147312 | 2.73e-03 | 5.10e-04 | 1.27e-03 | 2.97e-04 | 556.61 | 7.11 | 54.02 |
| 16 | 589680 | 1.09e-02 | 2.08e-03 | 3.07e-03 | 9.81e-04 | 559.91 | 7.15 | 54.34 |
| 16 | 2359152 | 4.33e-02 | 8.43e-03 | 9.90e-03 | 2.94e-03 | 561.51 | 7.17 | 54.49 |

Table 3: Results of the multipole-to-local computation on the NVIDIA FERMI GPU. Each row entry of the table presents the results of a single test run. The description of the columns follows, from left to right: number of terms computed, number of translations performed, GPU execution time for M2L kernel, GPU execution time for reduction, time for data transfer from host to device (HtD), time for data transfer from device to host (DtH), number of giga-operations per second (OPS.) for M2L kernel, effective bandwidth (B.) utilization for M2L kernel, metric of M2L translations per second performed (in millions).

block execution, and all other memory is stored in the registers. Furthermore, as each thread only computes a single complex coefficient of a local expansion at a time, the overall register usage stays low.

The crucial features of the row-traversal formulation of the kernel that resulted in remarkable performance gains are summarized as follows:

1. Increased the number of threads per block (in fact, we can have any number of threads).
2. Avoid thread branching for threads in the same warp.
3. Loop-unrolling (when done *manually*, performance was even greater).
4. Reduced accesses to global memory.
5. When accessing global memory, we ensure it is *coalesced*.

The final step taken in the optimization of the GPU formulation of the kernel was to overlap memory movements to and from global memory with computational work. The result is a high-performance algorithmic formulation tuned for the GPU architecture, that attains **548** Giga operations per second on TESLA and up to **561** Giga operations per second on FERMI.

In Table 2 and Table 3 we present performance results of several numerical experiments for the M2L kernel using TESLA and FERMI GPUs, respectively. The numerical tests for the M2L implementation are controlled by two parameters: *the number of terms of the multipole expansion*, and *the number of expansions to be translated*. The intensity of the calculations is given by the number of terms of the ME, and the size of the problem is given by the number of expansions to translate. Both tables report the time spent on the kernel execution and data transfer between host and GPU, and performance metrics for the M2L kernel. Performance metrics reported correspond to the number of giga-operations per second, effective bandwidth utilization, and the number of M2L translations per second performed in millions.

Looking at the results presented in Table 2 and Table 3, consider that the M2L is a compute-bound kernel. As such, it is limited by the maximum throughput of the GPU. Therefore, for the GPU to achieve its maximum throughput performance, it requires a test run large enough to completely utilize all of the streaming processors. This is reflected in the results shown, as the M2L kernel execution on the GPU reports its best throughput on the test runs that have more ME terms and larger problem sizes. We also report the number of translations per second as a "real world" performance metric that only depends on the test problem, *i.e.*, it reflects wall-clock time to solution. Note that we obtained comparable single-precision performance between the TESLA and FERMI architectures. We attribute this to the fact that our application does not benefit from the new features introduced in FERMI, such as improved double precision performance, and L1 & L2 cache.

### 4.2. Fast Gauss transform

We will propose several improvements beyond simply rewriting CPU code for CUDA, and later, in §5 we will present actual code used for these accelerations. Firstly, instead of using the entire FGT algorithm, we will focus on the evaluation of the Hermite series. According to many test runs with a full CPU implementation of the FGT, this part of the algorithms takes in the order of 90% of the total run time, so any optimizations here will have an appreciable effect on the overall speed of the algorithm. These optimizations will be placed in the context of the hardware metrics we introduced earlier in this section.

The Hermite series evaluation takes a set of series coefficients, $A_\alpha$, a target point (in $d$ dimensions), $y$, the center of the source cluster, $s_B$ and returns the influence of that source cluster at $y$. The equation form is given previously by (6), but is presented again here for ease of reading:

$$G(y) = \sum_{\alpha \geq 0} A_\alpha h_\alpha \left( \frac{y - s_B}{\sqrt{2}\sigma} \right). \tag{9}$$

#### 4.2.1. Reformulation of the FGT algorithm for GPU

If one were to make no changes in the FGT, then the GPU implementation would follow the same logic as that used for a normal CPU implementation. This means that for every source-target cluster interaction, a single CUDA kernel would be called. Each call uses the coefficients for this cluster along with its center point. Each thread would then take care of a single target point and compute the potential for that point from the Hermite coefficients. This approach, while naive, does have some advantages; it allows us to keep all variables needed by more than one thread in shared memory, while also ensuring concurrency and homogeneity are high. However, such an approach makes no deliberate attempt to leverage the advantages of CUDA. This is especially evident as the data sets being worked on are small—thus the overhead from memory accesses is going to be large compared to the run time, slowing down the implementation significantly.

Clearly, the approach described above will not give substantial acceleration on the hardware. Thus, we explore the algorithmic changes that will be necessary to improve performance. One first possibility is to work with multiple series expansions from multiple clusters at once. In this way, instead of sending the coefficients for a single source cluster, $A_\alpha$ and its associated cluster center, $s_B$, we amalgamate all necessary coefficients and cluster centers that must be evaluated at our target points into two large arrays,

$$A_\alpha = \left[ A_\alpha^{(0)}, A_\alpha^{(1)}, \cdots, A_\alpha^{(j)} \right] \quad \text{and,} \tag{10}$$

$$s_B = \left[ s_B^{(0)}, s_B^{(1)}, \cdots, s_B^{(j)} \right]. \tag{11}$$

In this way, we try to "hide" the time required to transfer all of the data to the device by increasing the computational intensity. We do this by performing a larger amount of work for each of the target points, which, once again, are assigned a single thread each, ensuring we keep the high levels of concurrency we would have enjoyed in a naive implementation.

This iteration falls short on a couple of new points: firstly, the number of values to be read into shared memory are too many. The size of the amalgamated coefficients, cluster centers

and other necessary shared values such as $\alpha$, can exceed the limited amount of shared memory (16kB) available. Secondly, to maximize concurrency and occupancy, we choose to let each thread read only a single value and perform collaborative memory operations (as discussed in Section §5.5). The number of threads per block is hardware-limited to 512, hence the number of shared values that can be read from global memory cannot be greater than this before we potentially compromise the homogeneity of the implementation. This indicates that we either need to store more values in global memory, take the performance hit and relinquish data-locality, or we must once again change our kernel.

### 4.2.2. Final algorithm

Taking into account the previously mentioned failings, our current implementation combines the best parts of both previous attempts to obtain very high performance. Instead of having one single kernel call with all the data needed, or multiple small calls as in a naive implementation, we split the evaluation into smaller calls. Each call then evaluates the influence of multiple source clusters, such that the amount of data needed is small enough to fit into shared memory, giving us spatial and temporal data-locality, and each thread need only read a single value into shared memory in order to ensure homogeneity. In this way, we ensure that all data needed across threads in a block (especially the series coefficients) is kept in shared memory, reducing the number of costly transfers between the device and host. This gives the GPU enough data to take full advantage of its features, while ensuring that all memory accesses are as fast as possible.

A further advantage of this approach is that spatial and temporal data-locality can be maximized. However, this refactoring is only one step of the optimization process. Within our evaluation of the Hermite series, we must perform other tasks, in particular evaluating Hermite polynomials for every appropriate source cluster at every target point.

### 4.2.3. Hermite polynomial evaluation

To evaluate polynomials in CUDA, we must first have some way of representing and working with them. Representing polynomials can be done with an ordered array of coefficients, but evaluation requires more work. There already exists an algorithmically optimal method for the evaluation in Horner's rule, and this method was implemented as a function in CUDA. Horner's rule allows us to evaluate a polynomial in $O(n)$ time [20], where $n$ is the degree of the polynomial. It does this by writing the evaluation in the form:

$$P(x) = \sum_{i=0}^{n-1} a_i x^i = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1})) \cdots). \quad (12)$$

Our first version of this function used a loop over the polynomial coefficients, and was implemented as a device-only inlined function. This use of loops involves the evaluation of conditionals, and consequent potential branching of code. This branching causes significant performance impact in CUDA, and so this

method needed to be re-implemented in some fashion to eliminate (if possible) all of the branching code. Our solution to this problem was to manually unroll the evaluation loop, a process outlined in more detail in §5. This technique improved the computational intensity and thus throughput of the algorithm, and as an added incentive, reduced the number of registers that were being used, allowing us to optimize the occupancy more easily.

### 4.2.4. Shared memory use

The optimal use of shared memory is vitally important to overall performance. As outlined in §4, the access time for shared memory is around 100× faster compared to global memory. Thus, every variable that needs to be used by more than one thread, or even accessed more than once, should be copied into either shared memory (in the case of variables to be accessed by multiple threads) or local registers in order to obtain both spatial and temporal data locality. In the case of our implementation, the Hermite series coefficients, $A_\alpha^{(j)}$, the cluster centers, $s_B^{(j)}$, the multi-index values themselves, $\alpha$, all need to be copied into shared memory.

### 4.2.5. Computing on-the-fly

A corollary of the access time for global memory is that one can have a situation where it is easier to calculate certain values on-the-fly, rather than read them from global memory. One such case is when calculating factorials. The standard method of calculating factorials by recursion is so computationally light that we can assign each factorial value to a single thread. This is preferential to the overhead of copying pre-computed values from global memory, and can be done while other threads are copying their values from global to shared memory. Additionally, all threads can calculate all values, but only save one. This allows us to unroll loops for greater performance and to maintain zero-branching.

### 4.2.6. Occupancy

Equally distributing work and memory accesses across threads is a necessary step for maximizing the occupancy of our implementation, as is reducing the number of registers used by each individual thread. These considerations allow us to make use of the highest possible number of threads at all times.

While the work performed by each thread is identical, ensuring homogeneity, memory transactions to global memory also need to be balanced. We achieve this balance by doing collective memory operations for retrieving data that is used by multiple threads. Each thread is then responsible for copying a single value from global memory into shared memory, and the index of the thread within its block dictates which value (polynomial coefficient, cluster center $x$ or $y$ value, etc.) the given thread will copy. For a more complete discussion, see §5.5.

### 4.2.7. Results

We present results of testing both a CUDA and a standard CPU code implemented in C++. Attempts have been made to optimize this C++ code, including parallelization for shared

memory systems using openMP, loop unrolling (using the `-funroll-loops` option from `gcc`) and evaluating hard-coded polynomial coefficients using Horner's rule. While this implementation is still not optimal, it offers a comparable code. This also explains the greater than 100× speedups observed, as these should be questionable when comparing similarly optimal CPU and GPU codes.

Timings for these tests were obtained by assuming that the kernels would be called within part of a greater algorithm. Thus, time taken to assign / populate target points and cluster co-efficients was not counted. In both the CPU and GPU cases these values would have been read from a file and computed by a separate method, respectively.

Tables 4 and 5 show comparisons between CPU and GPU codes running on identical datasets with a number of evaluation points ranging up to over 6 million. The truncation variable for the Hermite series is varied between $p = 3$ and $p = 9$. For table 4, the CPU implementation is run on between 1 and 4 cores of an Intel Penryn processor, and the GPU code on a single NVIDIA TESLA C1060 card. For table 5, the processor is an Intel Nehalem and the GPU is an NVIDIA FERMI C2050. In both cases only the particles evaluated per second are compared (PPS in the tables), with the additional reporting of the number of giga-operations (GOPS) performed on the GPU given as a measure of the total utilization of the card. Useful to note is that the CPU code scales nearly linearly with the number of cores, showing it is at least fairly efficient, and that the greatest speedup from the GPU with respect to 4 cores on the GPU is around 25 times on the Nehalem / FERMI system and 30 times for Penryn / TESLA. Given the good multi-processor scaling of the CPU code, this represents a roughly $100 - 120$x speedup over a single CPU core. The greater than 100× speedup encountered is above what is commonly quoted as possible for a compute-bound application, but can be explained due to the possibility of many more optimizations (such as SSE operations) being implemented in the CPU code. Particle evaluations per second (PPS) are also shown, to give a more "physical" representation of the speed of the code. While these results may seem high at first glance, they are within the theoretical peak of both cards (1033 Gops for the FERMI card), and we have verified by examining the intermediate .ptx files that fused multiply-add instructions have been used throughout the main inner loop that evaluates the Hermite polynomial. Given that our calculation is entirely throughput-bound, this ensures that we obtain very good performance.

The results presented show a significant disparity in the number of giga-operations performed, especially in the case of $p = 12$, where the peak for the FERMI card is above 1000 GOPS, while the TESLA card only manages 700. The reason for this is likely the large number of integer operations required in the Hermite evaluation kernel in order to resolve memory locations. For instance, the full version of the code-unrolling snippet in Figure 8 requires only 2 floating point operations $(+, \times)$, which can be done in a single fused multiply-add (FMA) instruction, while it requires an additional 2 integer operations $(+, \times)$ to obtain the correct coefficient memory location. In fact, for every floating point operation, on average, an integer operation must

also be performed. Referring to Table 5-1 in NVIDIA's programming guide [18], we see that for the TESLA GPUs (Compute Capability 1.3), 32-bit integer multiplies and multiply-adds both require multiple instructions, giving an integer throughput significantly lower than that for floating point operations. From the same table, we see that the FERMI card (Compute Capability 2.0) has the same throughput for these operations as for floating point operations. This significant increase in integer throughput removes a bottleneck in the code, and thus allows to improve our performance over the previous generation of cards.

## 5. GPU implementation discussion

In order to assist other practitioners in reaching these levels of performance, we now discuss the design strategies and in some cases demonstrate optimized fragments of code used within our kernels. These methods have been tuned for performance and are suitable for adaptation and use in any applicable CUDA implementation.

### 5.1. Thread execution branching

A code design consideration that greatly impacts performance is *branching* of threads during execution due, for example, to data-dependent conditionals. In the CUDA architecture, streaming multi-processors are able to manage a multitude of concurrent threads, but they do so in groups of 32 parallel threads, called *warps*. All threads in a warp start together, but they may branch as they execute. If they do, the warp actually executes in serial mode the different branch paths, while threads that do not take the particular path wait, effectively idling and wasting cycles. Thus, all threads in the warp converge after the branching and then continue execution together. For this reason, branching can be a serious hurdle for obtaining performance.

### 5.2. Multithreading

The CUDA architecture relies on no-cost multithreading to hide the high latency of memory accesses to global memory. A multiprocessor of a GT200 GPU chip can have up to 1024 active threads and 8 active thread blocks at any given time. However, active threads on an SM share its limited resources (registers and shared memory). In practice, for a given kernel implementation, the kernel's resource utilization defines the limit on the number of active threads per SM. The ideal for a memory-bound application is to have as many active threads as possible, to hide the latency of the memory transactions. In order to characterize the efficiency of the SM utilization, the concept of *Occupancy* [3] was introduced and defined as:

$$\text{Occupancy} = \frac{\text{Number of active threads}}{\text{Maximum threads available}} \quad (13)$$

The number of active threads will depend on a few parameters: on one hand, the kernel implementation will define the number of resources used per thread and per thread block, on the other hand, the resources per SM are limited by the hardware. The only parameter that is defined by the user at run time is the

| N | p | PPS (1 Core) | PPS (2 Cores) | PPS (4 Cores) | GOPS (GPU) | PPS (GPU) |
|---|---|---|---|---|---|---|
| 25600 | 5 | 1.68e+05 | 3.36e+05 | 6.59e+05 | 461.3 | 1.55e+07 |
| 102400 | 5 | 1.68e+05 | 3.35e+05 | 6.60e+05 | 539.1 | 1.81e+07 |
| 409600 | 5 | 1.69e+05 | 3.34e+05 | 6.68e+05 | 563.2 | 1.89e+07 |
| 1638400 | 5 | 1.69e+05 | 3.37e+05 | 6.72e+05 | 570.7 | 1.91e+07 |
| 6553600 | 5 | 1.68e+05 | 3.33e+05 | 6.72e+05 | 572.9 | 1.92e+07 |
| 25600 | 9 | 3.99e+04 | 8.02e+04 | 1.59e+05 | 549.9 | 3.93e+06 |
| 102400 | 9 | 4.03e+04 | 7.93e+04 | 1.58e+05 | 637.1 | 4.55e+06 |
| 409600 | 9 | 4.03e+04 | 8.01e+04 | 1.59e+05 | 663.9 | 4.74e+06 |
| 1638400 | 9 | 4.03e+04 | 8.03e+04 | 1.61e+05 | 671.6 | 4.80e+06 |
| 6553600 | 9 | 4.02e+04 | 8.03e+04 | 1.60e+05 | 673.7 | 4.81e+06 |
| 25600 | 12 | 2.05e+04 | 4.07e+04 | 8.08e+04 | 257.1 | 8.30e+05 |
| 102400 | 12 | 2.06e+04 | 4.08e+04 | 8.16e+04 | 490.8 | 1.59e+06 |
| 409600 | 12 | 2.04e+04 | 4.07e+04 | 8.18e+04 | 649.8 | 2.10e+06 |
| 1638400 | 12 | 2.05e+04 | 4.09e+04 | 8.17e+04 | 688.8 | 2.23e+06 |
| 6553600 | 12 | 2.05e+04 | 4.10e+04 | 8.15e+04 | 703.4 | 2.27e+06 |

Table 4: Results on both a multi-core processor and GPU for the Hermite evaluation kernel. Presented are Particles per second (PPS) for all cases, and on the GPU the total number of giga-operations (GOPS) obtained are also shown. Runs were made on a Penryn series Intel CPU and NVIDIA TESLA C1060 GPU

| N | p | PPS (1 Core) | PPS (2 Cores) | PPS (4 Cores) | GOPS (GPU) | PPS (GPU) |
|---|---|---|---|---|---|---|
| 25600 | 5 | 2.31e+05 | 4.41e+05 | 7.11e+05 | 699.1 | 2.34e+07 |
| 102400 | 5 | 2.50e+05 | 4.93e+05 | 9.39e+05 | 780.0 | 2.62e+07 |
| 409600 | 5 | 2.54e+05 | 4.92e+05 | 9.31e+05 | 815.3 | 2.73e+07 |
| 1638400 | 5 | 2.54e+05 | 4.98e+05 | 9.35e+05 | 828.7 | 2.78e+07 |
| 6553600 | 5 | 2.58e+05 | 5.00e+05 | 9.76e+05 | 830.3 | 2.78e+07 |
| 25600 | 9 | 7.12e+04 | 1.38e+05 | 2.64e+05 | 786.2 | 5.62e+06 |
| 102400 | 9 | 7.24e+04 | 1.42e+05 | 2.65e+05 | 858.7 | 6.13e+06 |
| 409600 | 9 | 7.19e+04 | 1.37e+05 | 2.78e+05 | 893.0 | 6.38e+06 |
| 1638400 | 9 | 7.21e+04 | 1.39e+05 | 2.78e+05 | 905.9 | 6.47e+06 |
| 6553600 | 9 | 7.29e+04 | 1.39e+05 | 2.80e+05 | 907.3 | 6.48e+06 |
| 25600 | 12 | 3.33e+04 | 6.62e+04 | 1.20e+05 | 874.5 | 2.83e+06 |
| 102400 | 12 | 3.33e+04 | 6.65e+04 | 1.27e+05 | 957.1 | 3.09e+06 |
| 409600 | 12 | 3.32e+04 | 6.64e+04 | 1.29e+05 | 996.2 | 3.22e+06 |
| 1638400 | 12 | 3.38e+04 | 6.52e+04 | 1.29e+05 | 1010.8 | 3.27e+06 |
| 6553600 | 12 | 3.34e+04 | 6.64e+04 | 1.29e+05 | 1012.1 | 3.27e+06 |

Table 5: Results from the Hermite evaluation kernel on multi-core processors and GPU. Particles per second (PPS) are presented for all cases, and the total number of giga-operations (GOPS) obtained on the GPU are also given. Runs were made on a Nehalem series Intel CPU and NVIDIA FERMI C2050 GPU

number of threads used per thread block. It is generally the case for memory-bound applications that by maximizing the occupancy of the multiprocessor we achieve better performance. In our application, we maximized the GPU occupancy in two ways: the most straightforward approach was to select a thread-block size that maximizes occupancy, one tool to do this is provided by NVIDIA and is named "occupancy calculator" [2]; the second approach was to hand-tune the implementation so that it reuses the maximum number of variables as possible. The last strategy has the most potential to improve occupancy by reducing the number of registers used, however it is a trial and error procedure, as the actual resource utilization ultimately depends on the compiler and sometimes less variables do not translate in less registers being used.

### 5.3. Memory management

Memory management needs to be explicitly provided by the programmer. This is a challenge, as the efficiency of the memory transactions directly depends on the algorithmic design and implementation. Issues that need to be considered are:

- ▷ *Small and fast shared memory*. The limited size of the shared memory, at 16 kB for the GPU we used, clearly imposes a restriction on the amount of fast storage available.

- ▷ *Large and slow global memory*. Global memory is characterized by the large memory space, with up to 4 GB of memory, but with very high latency of access from the GPU chip (each transaction taking between 400 and 600 clock cycles).

- ▷ *Shared memory conflicts*. Shared memory is physically divided into 16 blocks, and each block can do one memory transaction per clock (read, write, or broadcast). Therefore, if more than one thread of the same warp accesses data in the same memory block, the thread memory operations are queued and executed sequentially.

- ▷ *Efficient global memory accesses*. Global memory is physically accessed as 32-, 64-, or 128-byte segments by the hardware. Therefore, every memory transaction that is issued will read or write to a whole segment, regardless of whether only one element of the segment is read or written into the segment. In this case, much of the memory bandwidth would be inefficiently used. Efficient memory transactions that are aligned and efficient per thread warp are commonly referred to as *coalesced* memory transactions.

- ▷ *Memory camping*. Global memory is divided into physical blocks. The total bandwidth of global memory is calculated as the aggregated bandwidth of all the blocks. In order to effectively use the bandwidth, memory transactions need to be spread across all blocks. In our implementation, we addressed this by designing the kernels so that the thread blocks evenly access the global memory locations.

### 5.4. Loop unrolling

In several parts of our kernels, we require iterating over a loop to calculate values, for instance in evaluating multipole expansions in the FMM and Hermite polynomials in the FGT. Loop-unrolling is a well-known process whereby a loop over some pre-determined number of iterations, say *N*, is replaced with the loop's body code, repeated *N* times. This technique trades a larger compiled binary size for increased performance. This kind of optimization can be performed automatically by the CUDA compiler, but in our testing, we found that manually unrolling the code when possible resulted in both increased performance, and fewer registers being used, allowing us to more easily maximize the occupancy of our kernels.

The specific example we demonstrate is the evaluation of Hermite polynomials in the FGT. This is done by implementing Horner's method, as previously described. Consider the code example shown in Figure 8. This code fragment shows the loop-based implementation, with *H being a pointer to an array of polynomial coefficients, x the desired evaluation point, and poly_len denoting the degree of the polynomial to be evaluated. The code fragment in Figure 9 demonstrates the same loop, but now unrolled for the instance where the polynomial degree is 5.

While this particular piece of code has been generated by hand, it would be easy to have a separate piece of code to automatically generate at compile time the correct number of lines for a given number of terms. This would ensure the best possible performance, while removing the tedious and possibly error-prone manual adding of lines for more terms.

The optimization just described deals with speeding up computations, but we still need to get the data to compute efficiently. Thus, we now discuss balancing reads and writes from global memory.

### 5.5. Balancing global reads/writes across threads

To maintain an equal amount of load across all working threads, it is important to ensure that no thread reads or writes disproportionately more to the global memory than any other. The main area where this problem appears is during the transfer of commonly-used variables from global to shared memory. A naive method may result in some threads reading multiple values into shared memory, while others read none, an obviously unsatisfactory situation. Thus, we propose the following system: the maximum number of values to be read into shared memory is the same as the number of threads within a block. The position of a thread within a block dictates which value it will read. This ensures that no thread reads more than one value. The example shown in Figure 10 is taken from our FGT CUDA code.

The code snippet divides our threads so that as many values are read simultaneously as possible, with the type of variable to be read determined by the thread's index within its block. The first group of threads will calculate factorial values; the second will read polynomial coefficients and the third group will read the expansion center. In the second part of the code snippet, lines 29 to 49, the code segment diverges the reads

```
1   __device__ float poly_eval(float *H, float x, int poly_len)
    {
3       int i;
        float y = H[0];
5
        /* LOOP TO BE UNROLLED */
7       for (i=1; i < poly_len; i=i+1)
        {
9           y = H[i] + x*y;
        }
11
        return y;
13  }
    y = poly_eval(H,x,num_terms);
```

Figure 8: Code snippet showing a loop-based implementation of the polynomial evaluation.

```
    /* Manually unrolled code */
2   y = H[0];
    y = H[1] + x*y;
4   y = H[2] + x*y;
    y = H[3] + x*y;
6   y = H[4] + x*y;
```

Figure 9: Code snippet showing the unrolled loop.

```
    // shared memory
2   __shared__ int shared_fact[NUM_TERMS];
    __shared__ int shared_alpha[LEN_ALPHA];
4   __shared__ float shared_sb[2];

6   // Read vars into shared memory
    // WARNING: Each block needs more threads than (LEN_ALPHA + NUM_TERMS + 2)
8
    if (threadIdx.x < NUM_TERMS) {
10      // generate factorial case
        k = 0;
12      alpha1 = 1;
            for (i=1; i < threadIdx.x + 1; i++) {
14              alpha1 *= i;
            }
16          // store in shared memory
            shared_fact[threadIdx.x + k] = alpha1;
18  } else if (threadIdx.x < NUM_TERMS + LEN_ALPHA) {
        // read alpha into shared
20      k = -NUM_TERMS;
        shared_alpha[threadIdx.x + k] = alpha[threadIdx.x + k];
22  } else if (threadIdx.x < NUM_TERMS + LEN_ALPHA + 2) {
        // read sb into shared memory
24      k = -NUM_TERMS - LEN_ALPHA;
        shared_sb[threadIdx.x + k] = sb[threadIdx.x + k];
26  } else {
        // default case - no read
28      k = 0;
    }
30  __syncthreads();
```

Figure 10: Code snippet showing the balanced reads/writes.

across all threads and contiguous reads are achieved by having adjacent threads reading adjacent values from memory. It is important to note here that in earlier versions of the NVIDIA CUDA compiler, "if" statements forced thread branching on a whole threadblock, while a "switch" statement only branched within a thread warp. Thus, the optimal version of the code in Figure 10 for compilers previous to version 3.*x* involved splitting the code into two separate steps: an "if" statement to decide which threads read which values, and then a separate "switch" step to actually perform the reads. However, with compiler versions 3.0 and above, this is no longer appropriate, and so the code in Figure 10 is optimal.

This implementation has significant advantages: first, memory transactions can be organized such that adjacent threads within the same warp can read/write adjacent values from/to global memory, therefore ensuring coalesced memory transactions. Secondly, while this example only handles reading as many values as we have threads, it is trivially extendable to handle as many reads as a user desires. Finally, for some memory-intensive kernels, it is possible to define the size of a thread block to obtain the best kernel performance for memory transactions, even if this implies that some threads could remain idle during the compute-intensive part of the kernel.

*5.6. Other design strategies*

▷ *Keep the kernel complexity low.* It is better to have a highly specialized kernel than a general purpose implementation. We consider three reasons for this: first, highly specialized kernels keep the resource utilization low, which in turn allows having more active threads per multiprocessor; second, high specialization allows for the use of specific optimizations; third, general cases normally result in branching conditionals, and this reduces the kernel performance.

▷ *Use many threads per thread block.* Many threads can cooperate during memory transactions, resulting in more efficient use of the memory bandwidth. This strategy is more important in the negative, *i.e.*, for kernels that use *too few* threads it will be difficult to perform coalesced memory transactions.

▷ *Interleave computations and memory transactions.* In a kernel, it is a good idea to interleave memory transfers with computational work within the same kernel, in contrast with using a three-stage kernel that consists of one stage to load data, a second stage to perform computations, and a third stage for saving data. We observed that in our cases it was better to have smaller stages that interleave memory transfers (load/write) with work. In this way, the multiprocessor-limited resources can be used more efficiently and it is easier to hide latency by multithreading, as it is more difficult to overlap threads performing big stages than a series of small efficient stages.

▷ *On-the-fly calculations.* As stated several times before, there is a significant overhead related to reading and writing from global memory. This overhead can be so significant that for some purposes it may be more efficient to

re-calculate values than to store and retrieve them. A good example of this is found in a factorial cache. In a standard CPU code, it is more efficient to pre-compute the values of all the factorials you may need to use at the beginning of your code, then simply re-use these values when needed. In a GPU implementation however, these values would need to be read into shared memory for each block. Instead of doing this, we consider designating a set of threads in each block to calculate these values instead of reading from memory.

## 6. Conclusions

This work demonstrates that it is possible to attain close to the practical peak of modern GPU architectures, not just with embarrassingly parallel problems, but with real-world, elaborate algorithms. The process, however, is not straight-forward. Intimate knowledge of the architectural features is required to rethink the core methods such that performance can be maximized. This familiarity with the hardware architecture, in fact, reveals that some algorithms will be better suited to perform well on the GPU. Some others will suffer some unavoidable bottlenecks due to being bound by memory bandwidth, rather than computation. When this happens, only moderate speedups will be possible on the new hardware. When a computationally intensive algorithm is properly reformulated for the GPU, however, two-order of magnitude speedups can be obtained. This holds fantastic promise for extending the capability of computing to solve some of the most challenging scientific problems.

The results obtained here, sustaining over 500 Gigaop/s on one TESLA C1060 card for fast summation algorithms, can have significant impact for applications using these algorithms. In particular, we are currently working to develop boundary element methods which are accelerated with the fast multipole method. Considering the potential speedup in applications, we anticipate unprecedented capability for acoustics, electromagnetics, bioelectrostatics, and other applications of the BEM and FMM.

Our next challenge is to utilize the GPU kernels developed for this work in a truly heterogeneous environment. We will couple these kernels to software libraries for distributed systems, and investigate scalability with multiple CPU cores and multiple GPUs. Such work will offer application scientists tools for truly groundbreaking discovery through advanced computing.

Codes and test scripts used to obtain the results presented in this paper are available from http://code.google.com/p/cufast/.

## References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, The landscape of parallel computing research: A view from Berkeley, Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley (December 2006).
URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[2] A. Ghuloum, Unwelcome advice, Intel Research Blog, http://blogs.intel.com/research/2008/06/unwelcome_advice.php (June 30 2008).

[3] NVIDIA Corp., CUDA programming guide version 2.2.1 (May 2009).

[4] L. A. Barba, A. Leonard, C. B. Allen, Advances in viscous vortex methods – meshless spatial adaption based on radial basis function interpolation, Int. J. Num. Meth. Fluids 47 (5) (2005) 387–421. doi:10.1002/fld.811.

[5] L. Greengard, V. Rokhlin, A fast algorithm for particle simulations, J. Comput. Phys. 73 (2) (1987) 325–348. doi:10.1016/0021-9991.

[6] M. O. Fenley, W. K. Olson, K. Chua, A. H. Boschitsc, Fast adaptive multipole method for computation of electrostatic energy in simulations of polyelectrolyte DNA, J. Comput. Chem. 17 (8) (1996) 976–991.

[7] J. T. Hamilton, G. Majda, On the Rokhlin-Greengard method with vortex blobs for problems posed in all space or periodic in one direction, J. Comput. Phys. 121 (1) (1995) 29–50. doi:http://dx.doi.org/10.1006/jcph.1995.1177.

[8] N. A. Gumerov, R. Duraiswami, Fast multipole methods for the Helmholtz equation in three dimensions, 1st Edition, Elsevier Series in Electromagnetism, Elsevier Ltd., 2004.

[9] L. Greengard, J. Strain, The fast Gauss transform, SIAM Journal on Scientific and Statistical Computing 12 (1) (1991) 79–94.

[10] R. Beatson, L. Greengard, A short course on fast multipole methods, in: Wavelets, Multilevel Methods and Elliptic PDEs, Oxford University Press, 1997, pp. 1–37, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.129.7826, checked August 2009.

[11] F. A. Cruz, M. G. Knepley, L. A. Barba, PetFMM —-a dynamically load-balancing parallel fast multipole library, Int. J. Num. Meth. Engineering 85 (4) (2010) 403–428. doi:10.1002/nme.2972.

[12] L. Nyland, M. Harris, J. Prins, Fast N-body simulation with CUDA, in: GPU Gems 3, Addison-Wesley Professional, 2007, Ch. 31, pp. 677–695.

[13] R. G. Belleman, J. Bédorf, S. F. Portegies Zwart, High performance direct gravitational n-body simulations on graphics processing units II: an implementation in CUDA, New Astronomy 13 (2008) 103–112.

[14] N. A. Gumerov, R. Duraiswami, Fast multipole methods on graphics processors, J. Comp. Phys. 227 (18) (2008) 8290–8313. doi:doi:10.1016/j.jcp.2008.05.023.

[15] M. Broadie, Y. Yamamoto, Application of the fast Gauss transform to option pricing, Management Science 49 (8) (2003) 1071–1088.

[16] S. Han, S. Rao, J. Principe, Estimating the information potential with the fast Gauss transform, in: Independent Component Analysis and Blind Signal Separation, Vol. 3889 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2006, pp. 82–89.

[17] C. Yang, R. Duraiswami, N. A. Gumerov, L. Davis, Improved fast Gauss transform and efficient kernel density estimation, in: Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on, IEEE Conference on Computer Vision, 2003, pp. 664–671 vol.1. doi:10.1109/ICCV.2003.1238383.

[18] NVIDIA Corp., CUDA programming guide version 3.0 (February 2010).

[19] F. A. Cruz, L. A. Barba, Characterization of the accuracy of the fast multipole method in particle simulations, Int. J. Num. Meth. Eng. 79 (13) (2009) 1577–1604. doi:10.1002/nme.2611.

[20] T. Cormen, C. Leiserson, R. Rivest, S. Stein, Introduction to Algorithms, 3rd Edition, The MIT Press, 2009.