



5-2010

PERFORMANCE EVALUATION OF MEMORY AND COMPUTATIONALLY BOUND CHEMISTRY APPLICATIONS ON STREAMING GPGPUS AND MULTI-CORE X86 CPUS

Frederick E. Weber III

University of Tennessee - Knoxville, fweber1@utk.edu

Recommended Citation

Weber III, Frederick E., "PERFORMANCE EVALUATION OF MEMORY AND COMPUTATIONALLY BOUND CHEMISTRY APPLICATIONS ON STREAMING GPGPUS AND MULTI-CORE X86 CPUS. " Master's Thesis, University of Tennessee, 2010.
https://trace.tennessee.edu/utk_gradthes/671

This Thesis is brought to you for free and open access by the Graduate School at Trace: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of Trace: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a thesis written by Frederick E. Weber III entitled "PERFORMANCE EVALUATION OF MEMORY AND COMPUTATIONALLY BOUND CHEMISTRY APPLICATIONS ON STREAMING GPGPU AND MULTI-CORE X86 CPU." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Gregory D. Peterson, Major Professor

We have read this thesis and recommend its acceptance:

Jack Dongarra, Robert Harrison, Robert Hinde

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Frederick Edward Weber III entitled "A Performance Evaluation of Memory and Computationally Bound Chemistry Applications on Streaming GPGPUs and Multi-core x86 CPUs." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Gregory Peterson, Major Professor

We have read this thesis
and recommend its acceptance:

Jack Dongarra

Robert Harrison

Robert Hinde

Accepted for the Council:

Carolyn R. Hodges
Vice Provost and Dean of the Graduate
School

(Original signatures are on file with official student records.)

**A PERFORMANCE EVALUATION OF MEMORY AND
COMPUTATIONALLY BOUND CHEMISTRY APPLICATIONS ON
STREAMING GPGPUS AND MULTI-CORE X86 CPUS**

A Thesis
Presented for the
Master's of Science
Degree
The University of Tennessee, Knoxville

Frederick Edward Weber III
May 2010

ACKNOWLEDGMENTS

I would like to thank my mother for exposing to various cultures and enabling me to become a well-rounded person, and my father for bringing home that Mac Plus when I was five. Without you all, I probably would be doing something else with little enthusiasm. I would like to thank Dr. Peterson for putting up with me and guiding me in my undergrad and graduate studies. I would like to thank my committee members for providing help and feedback in writing this thesis. A special acknowledgment goes to Jimmy John's for delivering and Ray's for having \$0.75 Cokes.

ABSTRACT

In recent years, multi-core processors have come to dominate the field in desktop and high performance computing. Graphics processors traditionally used in CAD, video games, and other 3-d applications, have become more programmable and are now suitable for general purpose computing. This thesis explores multi-core processors and GPU performance and limitations in two computational chemistry applications: a memory bound component of ab-initio modeling and a computationally bound Monte Carlo simulation. For the applications presented in this thesis, exploiting multiple processors is done using a variety of tools and languages including OpenMP and MKL. Brook+ and the Compute Abstraction Layer streaming environments are used to accelerate applications on AMD GPUs. This thesis gives qualitative assertions about these languages and tools regarding ease of use and optimization in addition to quantitative analyses of performance. GPUs can yield modest performance improvements with little effort in some applications and even larger speedups with simple optimizations.

TABLE OF CONTENTS

1	Introduction.....	1
1.1	Multi-Core Processors	3
1.2	Graphics Processors for General Purpose Computation.....	4
2	X86 architecture.....	5
2.1	Multi-core/Scientific Computing APIs.....	6
2.2	OpenMP	6
2.3	Intel™ Math Kernel Library (MKL)	8
3	Graphic Processor Unit (GPU) Hardware.....	9
3.1	Advanced Micro Devices (AMD) Hardware.....	10
3.2	GPU Software	13
3.2.1	Brook+	13
3.2.2	Compute Abstraction Layer (CAL)	14
4	Applications	20
4.1	Grid Potential Computation	20
4.1.1	Naïve Implementation.....	21
4.1.2	Optimized Implementation	23
4.1.3	Results.....	26
4.1.4	Other Work	32
4.2	Quantum Monte Carlo (QMC).....	34
4.2.1	Naïve Implementation.....	36

4.2.2	Optimized QMC.....	38
4.2.3	QMC Performance	40
5	Conclusions.....	45
5.1	Future Work.....	46
	References.....	47
	Vita.....	50

LIST OF TABLES

Table 4.1: Optimizations implemented by platform.....	24
Table 4.2: Timings for QMC Kernels (Fastest in Bold)	41

LIST OF FIGURES

Figure 3.1: Stream processor layout on ATI GPUs	10
Figure 3.2: Thread layout on AMD hardware	11
Figure 3.3: Streaming Computational Model	15
Figure 3.4: Stream Addition	15
Figure 3.5: Effective Kernel Computation.....	15
Figure 4.1: A Naive Algorithm for Computing Grid Potential.....	22
Figure 4.2: Naive Brook+ Algorithm for Computing Grid Potential	22
Figure 4.3: Performance of naive and optimized implementations of grid potential computation.....	29
Figure 4.4: x86 performance scalability	29
Figure 4.5: Speedup factor over x86 architecture in naive grid potential.....	32
Figure 4.6: Performance increases due to optimization.....	33
Figure 4.7: Speedup in optimized kernels over x86 optimized implementation	33
Figure 4.8: Algorithm for Variational Monte Carlo[27].....	35
Figure 4.9: Performance Improvement over Naive Implementation	42
Figure 4.10: Performance Improvement over MKL Implementation.....	43
Figure 4.11: Performance Model of QMC Runtime.....	44
Figure 4.12: QMC Runtime	44

1 Introduction

Software development is currently undergoing a radical paradigm shift. Rather than using sequentially executed programs running on a single processor, applications are becoming increasingly parallel. Multi-core processors give little performance improvement over single processors to applications not written to execute in parallel (exploiting either task or data level parallelism). As such, application developers are recoding many of their applications to be able to perform multiple tasks simultaneously. In addition to multi-core processors, Graphics Processing Units (GPUs) are becoming general purpose enough to use in generic computation.

This thesis compares the computational speed of two chemistry applications running on an AMD graphics accelerator to their speed on x86 multi-core processors. The applications accelerated include a variational Monte Carlo application used in quantum modeling and a piece of ab-initio modeling (the computation of a potential grid using Gaussian basis functions). The former is computationally bound while the latter is memory bound. While using graphic accelerators to perform general computation has been done before [1, 2] and AMD's Compute Abstraction Layer has been used to accelerate a Monte Carlo application in gluodynamics [3], the applications presented in this thesis have not been mapped to AMD's streaming model. This thesis explains several strategies for optimizing these applications running in a streaming model and how these strategies compare to those used on conventional Intel multi-core processors. This thesis shows that AMD GPUs can be used to achieve a 7-10x speedup over multi-core

implementations of the presented applications.

The research performed in this thesis evaluates using AMD GPUs with a streaming model and multi-core Intel processors in computational chemistry applications. When taken in context with other work on these applications¹, a comparison of many emerging architectures is given. The work presented in this thesis yields expectations of how much effort is required to code an algorithm on multi-core processors and AMD GPUs, how much work is needed to optimize that code, and what performance can be achieved an application written in Brook+, CAL, or C. These expectations can be applied to other applications to determine if writing code to use multiple processors or an AMD GPU is worth the costs of doing so. Additionally, the models presented in this thesis allow performance predictions based on problem size.

This thesis is divided into three sections. The first section (chapters 1-3) serves to provide introductory information for the reader about developing software for modern multi-core processors and GPUs, how the hardware is designed, and a high level description of both the quantum Monte Carlo and Gaussian Basis Function applications. The second section describes how these applications are implemented and how the execution speed compares on x86 multi-core processors and an AMD GPU (chapter 4). The final section (chapter 5) concludes the thesis, and describes future work for these applications.

¹ Akila Gothandaraman at the University of Tennessee has implemented the variational Monte Carlo application for CUDA and FPGAs. Philip Brown from Bristol University evaluated the grid potential computation on Clearspeed, Cell, and CUDA platforms.

1.1 Multi-Core Processors

Until several years ago, performance gains in applications were generally achieved through hardware upgrades. Higher clock rates increased the instruction execution rate, while deeper pipelines and out of order execution schemes that exploited Instruction Level Parallelism (ILP) provided performance increases to legacy software. ILP allowed software developers and end users to rely on better hardware to improve an application's performance with little to no recoding or compilation. Unfortunately, ILP does have limits in its effectiveness at yielding performance gains [4].

A number of factors have caused architecture designers to focus on exploiting parallelism at the data level rather improving the execution rate of a single processor. Firstly, decreases in memory latencies have not kept up with increases in CPU clock rates [5]. This increases the amount of time a processor spends waiting for data from memory. High-speed caches have helped overcome memory latencies, but data reuse is necessary to make them effective and their relatively small size limits their effectiveness in some applications. Besides memory latencies, power consumption and heat dissipation (a byproduct of power consumption and processor size) make designing processors with higher clock rates more difficult. Multi-core design addresses power wall limitations by reducing clock frequencies while increasing throughput [6]. Its ability to overcome the memory wall is disputed; small numbers of cores do provide parallel speedup but performance drops off fairly quickly in data intensive applications [7]. However, kernels that can be efficiently blocked to efficiently exploit caches may have a less grim outlook [8]. Using multiple cores allows properly written applications to perform several tasks

concurrently. While multi-core and symmetric multi-processing are beginning to emerge as a viable means of running data parallel tasks, Graphics Processor Units (GPUs) are also becoming an effective way of performing computation.

1.2 Graphics Processors for General Purpose Computation

Graphics processors traditionally have been relegated to, as their name implies, rendering graphics. They were originally designed with a fixed pipeline used to arrange vertices into polygons and then fill these polygons [1]. The main application domains included games, computer aided drafting, and anything that required real-time 3-D modeling. However, as these applications began to demand more realism in rendering, fixed pipelines on GPUs became replaced with programmable shaders. Using vertex and pixel shaders, developers could define their own functionality for rendering polygons, allowing for more sophisticated lighting and shadowing [1]. These shader capabilities allow GPUs to be used for more than graphics.

Graphics accelerator manufacturers are now recognizing the need for general high performance general purpose computing. Both Nvidia and AMD have languages for developing programs to run on their GPUs. While many current languages, such as CUDA [9] and Brook+ [10] are vendor specific, OpenCL [11] seeks to provide cross platform GPU support. Additionally, Stanford's BrookGPU [2] project, from which Brook+ is derived, is cross platform. These languages allow developers to exploit the performance of graphics processors, which is often an order of magnitude higher than conventional processors.

2 X86 architecture

The x86 is a traditional CPU architecture with a long history. Introduced by Intel in 1978, the 8086 set the foundation for an architecture that continues to dominate the personal computing and server markets. Since its inception, the x86 has evolved from a 16-bit microprocessor running at 5-10Mhz with an optional x87 80-bit floating-point co-processor to a 64-bit superscalar, out of order, multi-core behemoth capable of several orders of magnitude more performance than its ancestor in virtually any application. Given the X86 architecture's wide deployment and familiarity with developers, this platform serves as performance baseline for comparison to other hardware [12].

Intel and AMD both offer x86 processors with a number of features that make them suitable for scientific computation. Firstly, the x86 processors have instructions for performing vector computations. Streaming SIMD (Same Instruction Multiple Data) Extensions (SSE) provide instructions for loading and storing (usually 16 byte aligned) 128 bit data types containing four single precisions floating point numbers or two double precision numbers. Other operations perform vectorized computation, such as adding or multiplying, four single or two double precision values at a time [13]. In addition to SSE, x86 processors have become multi-core in the past several years [14]. This makes them suitable for exploiting task level parallelism. In order to exploit these hardware features, a number of libraries exist, such as Intel Threading Blocks², OpenMP, and pthreads.

² <http://software.intel.com/en-us/intel-tbb/>

2.1 Multi-core/Scientific Computing APIs

Developers wanting to use SSE enabled processors generally write code to explicitly use this feature or invoke library calls that use SSE. While modern C compilers can perform auto-vectorization [15], quality of implementation can vary widely and usually requires very strict assumptions and relaxed constraints to ensure correctness. As such, SSE enabled code is often created using intrinsics (wrapper functions that usually correspond to assembly instructions) or assembly language. Fortunately, many linear algebra and vector routines have already been compiled into SSE-enabled libraries. Many vendor-provided implementations of Basic Linear Algebra Subprograms (BLAS), for example, use tuned SSE-enabled kernels to achieve high performance. These libraries allow applications to more easily exploit hardware functionality while writing fewer tuned linear algebra kernels. Running applications on multiple processors is typically done at using a high level library such as OpenMP [16] or using pthreads at a low level.

2.2 OpenMP

While there are several methods for achieving task or data parallelism by using multiple processors, OpenMP is among the easiest to use. OpenMP provides parallelism through a number of task-oriented mechanisms. Several different parallel mechanisms are declared using *#pragma* compiler directives. Since pragma directives not understood by the compiler are ignored, it is possible to write code that will compile with or without OpenMP support. One of the simplest parallel paradigms supported by OpenMP is the parallel for, which executes separate loop iterations concurrently on multiple processors. More generally, one can declare a parallel region and the number of threads to execute

that region of code. OpenMP handles task scheduling for the user in one of three ways [16].

Tasks can be scheduled using work-stealing queues, or statically assigned. Using the *static* scheduling approach, tasks are always assigned to the same thread. If a user runs a static section multiple times, each worker thread will always receive the same tasks. This allows for predictable cache use across threads in some applications, but threads can suffer from work starvation under this schedule (e.g. some threads finish their tasks more quickly than others, leaving them with no work to do). *Dynamic* scheduling allows threads to request work to perform. This has the advantage of reducing the effects of thread starvation, but at a cost of predictability. With dynamic scheduling, tasks are divided into equally sized chunks of work. *Guided* scheduling creates large chunks of work for threads to initially compute, but as threads' assigned tasks decrease in size as more work is requested. This allows for the best load balancing of the three scheduling methods [16].

OpenMP's memory model allows threads to share and hide data as necessary. Buffers and variables are shared by default, which means that two threads writing to the same variable are in fact writing to the same memory location. If no mutual exclusion constructs are used, this usually constitutes a race condition and is highly undesirable. While OpenMP contains constructs for ensuring exclusive use of memory regions by threads, data can alternatively be declared as private. If a variable or buffer is declared private using the *threadprivate* directive or the *private* clause, then it is allocated and copied once for each thread. This allows each thread to write to unique memory regions and negates the need for expensive mutual exclusion constructs in many cases. [16]

2.3 Intel™ Math Kernel Library (MKL)

Intel's Math Kernel Library (MKL) allows users to accelerate applications that use linear algebra routines. MKL is a collection of linear algebra and other mathematical routines. It contains BLAS, LAPACK, Vector Math Library (VML), and other libraries in a single package [17]. These libraries are tuned to exploit the performance capabilities of Intel's processors. While its optimizations allow for one of the best performing single-threaded collection of routines, MKL can also use OpenMP or pthreads to parallelize these routines. [17]

In some cases, MKL provides an effective means of exploiting multi-core processors. Some algorithms, such as iterative methods, have too many dependencies and are inherently sequential in nature. Under these conditions, coarse grain parallelism isn't achievable, meaning that the only way to achieve parallelism is to exploit it in the linear algebra routines. While some applications best achieve parallelism in this manner, many best achieve parallelism at higher levels (such as the two applications presented in this paper).

3 Graphic Processor Unit (GPU) Hardware

While graphics processors originally had a fixed functionality pipeline, this has been replaced with programmable shader units. In graphics applications, the pipeline assembles data into a scene that can be turned into an image. Typically, the pipeline first assembles vertices into geometric primitives (usually triangles). The geometry is then filled, usually by mapping a 2-D image onto its surface (e.g. texturing). Finally, lighting and shadowing are computed to yield the final scene. This scene is then projected into a 2-D bitmap that is rendered to a graphics context (usually the user's monitor). Modern GPUs allow developers to define how geometry is drawn and filled by using vertex and texel shaders respectively. Due to demand for high quality yet low latency rendering, GPUs have many processors for constructing images in a highly parallel manner [1].

While GPUs are inherently designed for 3-D graphics, many of their qualities make them exceptionally well suited for scientific computing. Graphics processors typically have large amounts of dedicated memory (256MB up to 4GB) with enormous bandwidth (10-100GB/s or more) [18]. This RAM is mainly used to store many large textures in graphics applications. In scientific computing, this RAM can store matrices and vectors. In addition to large amounts of high bandwidth memory, GPUs feature a large number of processors (64-100s, depending what one defines a "processor" to be) for computing texture filling and vertex transformations (which are actually small matrix-vector multiplications). This arrangement of many small processors is also convenient for many scientific applications. Projects such as Folding@Home and NAMD have been

implemented on GPUs to achieve large speedups [1].

3.1 Advanced Micro Devices (AMD) Hardware

AMD's GPUs use a hierarchy of processors to provide massive parallelism. This arrangement is shown in Figure 3.1. Stream processors are divided into SIMD engines. Each SIMD engine runs a number of threads concurrently on its thread processors. These threads are grouped into a *wavefront*. Each SIMD engine can time slice execution of multiple wavefronts. Within a wavefront, a number of threads execute concurrently. Four threads are multiplexed per thread processor to hide memory latencies [10] (in a manner similar to Intel's Hyperthreading technology [19]). Finally, each thread processor contains 5 stream cores.

Thread creation, termination, and switching are automatically handled by the rasterization hardware. Threads are created in a zig-zag pattern through the domain of

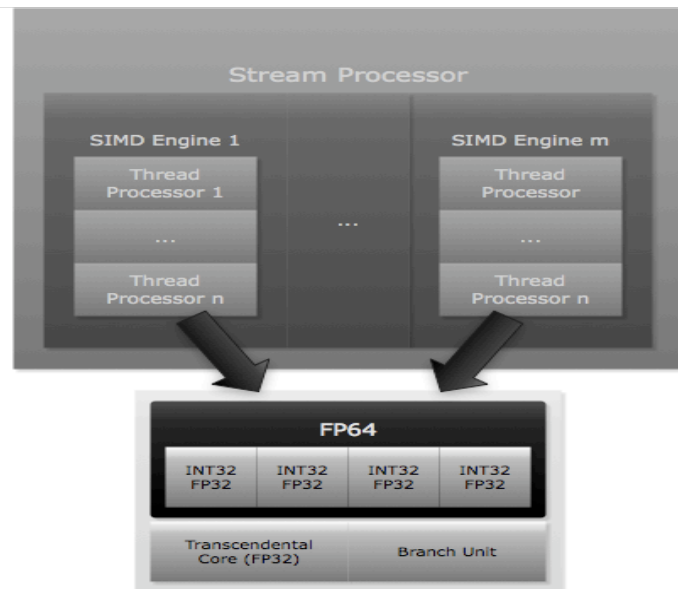


Figure 3.1: Stream processor layout on ATI GPUs

execution. This is shown in Figure 3.2. The scheduler assigns groups of four threads (quads) to be run on thread processors within a SIMD engine [10].

Threads executing on the same SIMD engine reside in a wavefront (analogous to a “warp” on Nvidia cards [20]) and execute the same instruction [10]. This implies that if a branch is taken by some threads and rejected by others within a wavefront, every thread in the wavefront must take both sides of the branch. The side of the branch that is not supposed to be executed does not save any of its results and has no effect on correctness but negatively impacts performance. As such, threads within a SIMD engine should ideally branch the same way to avoid doing extra computation.

Stream cores are the hardware components that actually perform computation.

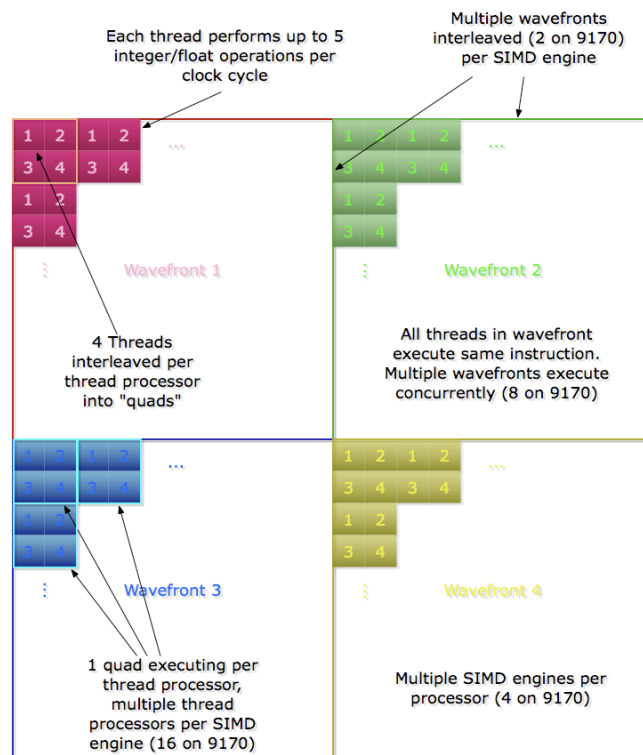


Figure 3.2: Thread layout on AMD hardware

Four of the five cores can perform basic integer or single precision floating-point calculations (add, multiply, etc.), while the fifth can additionally perform more complex calculations such as natural logarithm, power, etc. On some newer GPUs (such as the Firestream 9170), the four non-transcendental floating-point units can be chained together to perform double precision computations. Transcendental operations cannot be performed on double precision numbers in hardware and only one double precision instruction can be issued per clock cycle rather than five[10].

Data storage and movement is handled through several mechanisms on the GPU. Firstly, large amounts of high-bandwidth GDDR (Graphics Double Data Rate) memory are used to store textures (or streams in the case of general purpose use). The Firestream 9170 has 2GB of GDDR3 memory with a peak bandwidth of 51.2GB/s³. For storing intermediate calculations in a kernel, memory mapped registers are used. These registers use high-speed memory and the number used determines how many wavefronts can be run concurrently. Since using more registers reduces the number of wavefronts that can be run simultaneously, reducing register usage (if possible) can improve parallel performance. Registers are 128-bits wide and divided into four components: x, y, z, and w. This allows four single-precision floating-point numbers, four integers, or two doubles to be stored in them. Complex swizzling operations are supported, allowing specific components of a register to be used in an instruction. When using double precision instructions, two components are passed to an instruction, the first containing the upper 32 bits and the second containing the lower 32 bits.

³ Personal communication with AMD

3.2 GPU Software

To develop general-purpose applications, AMD provides two APIs that use the streaming paradigm: CAL and Brook+. Streaming is a computational model designed to separate data accesses from computation. It uses two abstractions for managing each: streams are collections of data and kernels are units of execution. Streams are an n-dimensional abstract data type. An array can be represented with a one-dimensional stream, a matrix with a two-dimensional stream, etc. Each element in a stream can have multiple values (e.g. a stream of float4 types). The domain of execution is usually defined to be the size of an entire stream. Kernels are executed at each point in the domain[10].

Kernels are the abstraction representing executable code. Many copies of a kernel are run in parallel. Within a kernel, streams can be either read from or written to, but not both. Intermediate results are kept in registers or other fast memory. These restrictions allow compilers to statically schedule memory transfers and kernel executions efficiently[21].

3.2.1 Brook+

Brook+ is a high level stream computing language developed by AMD. It is based on the Brook project at Stanford University[2]. Brook+ is a C-like programming language that uses kernels running on the GPU in conjunction with host side code written in C++. Kernels are defined using the *kernel* keyword. Top-level kernels operate on streams while other kernels can be used to modularize code. If a kernel is top-level, it can read from and write to streams, but cannot return data. Kernels that are not top level cannot operate on streams. Instead, they serve as functions to return a result based on

some inputs. Kernels may not call regular functions, though functions can call kernels. When running code in Brook+, the domain of execution is defined by default to be the size of a kernel's first output stream, though the user can change this manually. Kernel instances are created for each point in the domain of execution (Figure 3.3) and may write only to the instance's location in an output stream in the standard kernel. Likewise, streams may only be read from at the current location in the domain of execution in a standard kernel. However, to overcome the limitations this model implies, Brook+ allows streams to be declared as scatter (for output streams) or gather (for input streams) using the `[]` operator instead of the `<>` operator, allowing for random access. Streams can be declared as input or output; input streams can only be read from while output streams can only be written to [10]. An example matrix/vector addition is given in Figure 3.4 [10]. If `a`, `b`, and `c` are 2x2 matrices represented as streams, this kernel will be called four times, effectively executing Figure 3.5.

Brook+ code is cross-compiled using `brcc` into backend code. Kernels are created in both C++ and Intermediate Language (IL) implementations to allow for execution using either the CPU or GPU runtime. Developers manage streams and kernel invocation through a C++ frontend. While streams and kernels can actually be statically managed in the Brook+ language, this is less flexible than using C++.

3.2.2 Compute Abstraction Layer (CAL)

The Compute Abstraction Layer (CAL) is a low-level streaming environment for performing computation. Like Brook+, it uses a streaming model for processing data.

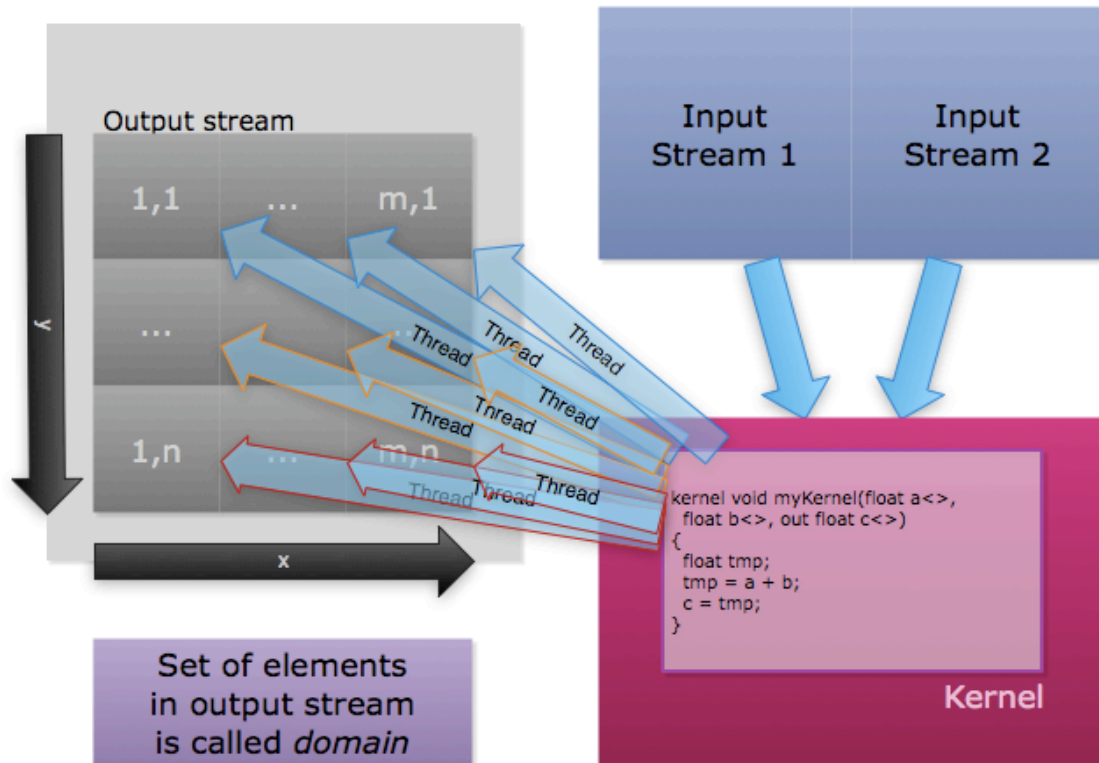


Figure 3.3: Streaming Computational Model

```
kernel void add(float a<>, float b<>, out float c<>)
{
    c = a + b;
}
```

Figure 3.4: Stream Addition

```
c[1,1] = a[1,1] + b[1,1];
c[2,1] = a[1,2] + b[1,2];
c[2,1] = a[2,1] + b[2,1];
c[2,2] = a[2,2] + b[2,2];
```

Figure 3.5: Effective Kernel Computation

However, CAL has additional constructs that can be used in kernels. Shared registers are accessible by all threads in a wavefront (rather than being accessible by only a single thread). In addition to reading and writing to streams, the global buffer can be used in scatter kernels. The global buffer is 128-bit addressed and can be read or written to by any thread at any index. It is most useful for reduce kernels [10].

CAL is divided into two components: the CAL runtime and Intermediate Language (IL). The runtime serves as the frontend for managing streams, devices, kernels, contexts, hardware counters, and kernel compilation. The runtime is implemented as a library of C functions. For handling streams, a number of options exist. A stream can be allocated locally (on the GPU) or remotely (on the host)[10]. In addition to these, a feature exists allowing a stream to be allocated on the host in any address the developer desires. This contrasts with remote allocation, which returns an address the runtime creates. Using this feature, data can be written directly into a buffer without needing to be copied. Local allocations are limited by the amount of memory on the GPU, remote allocations are limited to 64MB, and pinning memory is limited to 16MB (as of SDK v1.3 for Linux)⁴.

To create a kernel, an assembly-like language (IL) is used. Since different video cards may have different architectures, portability is difficult to achieve if programming in a particular card's assembly language. IL is a pseudo-assembly language that maps to a specified device's assembly language, which is then assembled into object code at runtime. Unlike most assembly languages, IL has high-level instructions for directly

⁴ Learned through personal communication on AMD StreamSDK forums. Topic: *Double memory copy in CAL? What about calCtxResCreate?*

performing loops, if/else blocks, and switch statements. True function calls (i.e. not inlined) are supported, allowing for some recursion, though the call depth is limited to 32 levels. While IL has several transcendental instructions, many of these are accurate to only 21 bits. This can limit their usefulness in many applications. Additionally, some Brook+ standard math functions (such as `exp()`) do not exist as transcendental instructions. Having no transcendental operations, double precision's utility is limited in many applications. This is a non-issue for many linear algebra kernels, such as matrix-matrix multiply, but does restrict CAL's usefulness in double precision when implementing scientific applications (such as the ones used in this paper). Each instruction is capable of powerful swizzling operations [22]. Swizzling allows the user to determine which components of a SIMD register are used in operations. For example, the x component of a register can be added to the y component of another register and stored in the w component in a third register.

Compiling and running IL kernels is a multi-step process that occurs at runtime. Kernels are stored in C-style strings using a newline to delimit lines. Compilation generally consists of getting device information from the CAL runtime (to determine which platform to target), compiling an IL kernel stored in a C-string via a function call, and linking objects into an image. To run a kernel, streams must first be mapped to register symbols used by the kernel. For example, if a kernel writes to register `o0`, the stream mapped to that register is written to at the current domain of execution. Unlike Brook+, the domain is always manually specified before execution. The number of streams mapped to registers must equal the number of input, output, constant buffer, and global buffer registers used in the kernel. Failing to map the correct number of streams to

registers will cause unpredictable behavior in a kernel. The same behavior is exhibited when a stream's dimensionality does not match its declaration in a kernel. Both of these cases are difficult to debug, since the function that loads the kernel returns no error message [10].

As implemented, the CAL API is extremely verbose. For example, allocating a stream resource requires a function call and another two functions to copy the stream to the GPU. To actually run a kernel, the developer must load the module that contains the kernel, get the entry point of the kernel from the module, get a memory handle for each resource, map each resource to a symbol (which takes 2 function calls), and then finally call a function that actually runs the kernel. To make the CAL implementation more readable and cleaner, a C++ library was developed to wrap functions into classes.

The *calutil* library, which I developed for this thesis, provides a simplified interface for the CAL while maintaining most of CAL's functionality. This library is divided into three essential classes, and two extension classes. The *CALgpu* class is used to handle devices and contexts. *CALbuffer* objects are a stream abstraction. They function in a fashion similar to Brook+ streams; they have *read()* and *write()* member functions that accept generic pointers. GPU memory is allocated in *CALbuffer* constructors and deallocated in destructors. The *CALkernel* class is used for managing kernels that run on the GPU. Its constructor automatically compiles and links supplied IL source code. To run a kernel, call *load()* then *assignSymbolsAndRun()*.

In addition to simplifying CAL's interface, the *calutil* library uses C++ exception handling and provides access to hardware counters. The *CALperfcounter* class creates a hardware counter. Currently, there are two counter types supported in CAL: one that

measures the cache hit rate and one that measures the idle time. Normally, these counters are hidden away in API extensions. Exceptions are thrown whenever a *CAL_RESULT_ERROR* is thrown by a CAL API function. Additional error checking not done by the CAL API (validating a stream's dimensionality and size versus its reference in an IL kernel for example) remains as future work.

In summary, CAL and Brook+ are the two APIs used to develop software on AMD GPUs. While Brook+ provides high-level abstractions, it does add overhead to kernels. CAL on the other hand can yield higher performance than Brook+ at the expense of development time and a higher learning curve. To help reduce the understanding needed to develop applications in CAL, I developed the *calutil* library, which provides C++ exceptions and class abstractions.

4 Applications

This thesis presents how to accelerate two applications using a graphics processor. The first application computes a Gaussian basis function. The second is a quantum Monte Carlo algorithm for determining an n-body atomic interaction.

4.1 Grid Potential Computation

Gaussian basis functions are used in the Hartree-Fock method for approximating the ground state wavefunction and energy in an n-body quantum-mechanical arrangement. While Slater functions are a natural fit for computing molecular orbitals, they are difficult to compute when orbitals are centered on different nuclei. Fortunately, linear combinations of Gaussian functions can serve as reasonable approximations to Slater functions and are more conducive to computation in a model[23]. Gaussian functions take the form $\exp(-\alpha r^2)$, where alpha is the radial spread of a Gaussian function and r is the distance between the centers of two orbitals.

When generating Gaussian functions in this application, a large number of coordinates are applied with a comparatively small number of basis functions. In the experiments run in this paper, 262,144 grid coordinates (from which the radius r is computed) were used with 640 basis functions (α). Both the n grid coordinates and m basis functions reside in vectors and the result created is an nxm (or mxn) matrix. This matrix is referred to as the grid potential. Each point in the matrix is given below.

$$g_j(r_i) = e^{\alpha_j(x_i^2 + y_i^2 + z_i^2)} \quad (1)$$

4.1.1 Naïve Implementation

A simple C algorithm for computing the grid potentials is given in Figure 4.1. The grid potentials are explicitly iterated over using two loops. In order to eliminate redundant computation, the coordinate radius is computed once in the outer loop and reused. Grid potentials are stored with the coordinates iterating in the leading or trailing dimension, depending on which method best maps to a given architecture.

The Brook+ implementation (Figure 4.2) is comparable in complexity to the C implementation. Looping is implicit over the grid potentials since the kernel is instantiated once for each point in the domain of execution. Since Brook+ does not allow more than 67,108,864 elements in a stream, the kernel had to be tiled into blocks of 8192 coordinates and 640 basis functions. The basis functions iterate in the leading dimension to make elements contiguous when stored back onto the host while maximizing the amount of work done in a kernel. This was also done to make the implementation consistent with the CAL implementation for a more direct comparison. Using tiles of 8192 coordinates, 32 kernel invocations are made to compute the entire set. Unlike the C implementation, the radius is recomputed many times. To reuse the radii requires an additional kernel call to precompute the radii into a separate stream, which is non-trivial unlike moving the computation out a loop as in the C implementation.

```

for(i = 0; i < npt; i++)
{
    float r2 = x[i] * x[i] + y[i] * y[i] + z[i] * z[i];
    for(j = 0; j < nbas; j++)
    {
        gridpot[j*npt+i] = exp(alpha[j] * r2);
    }
}

```

Figure 4.1: A Naive Algorithm for Computing Grid Potential

```

kernel void
computeBasisFunction(float alpha[], float xCoord[],
float yCoord[], float zCoord[], out float basis<>)
{
    float2 index = indexof(basis).xy;

    float x2;
    float y2;
    float z2;
    float r2;

    x2 = xCoord[index.y] * xCoord[index.y];
    y2 = yCoord[index.y] * yCoord[index.y];
    z2 = zCoord[index.y] * zCoord[index.y];

    r2 = x2 + y2 + z2;

    basis = exp(alpha[index.x] * r2);
}

```

Figure 4.2: Naive Brook+ Algorithm for Computing Grid Potential

The naïve CAL implementation functions identically to the Brook+ implementation. The kernel operates on 32 8192x640 tiles of the grid potential to compute the complete result. Unlike Brook+, CAL does not have an exponential function. As such, a power instruction was used with e as the base to mimic an exponential. The maximum error this incurred in this application with the parameters used was approximately .00037%, similar to that given by the exp function in Brook+. As an alternative method of computing the exponential, the equation below was used. This was attempted because CAL has a 2^x instruction. Numerical results and performance were similar to using the power instruction.

$$e^x = 2^{\frac{x}{\ln(2)}} \quad (1)$$

4.1.2 Optimized Implementation

To improve performance, optimizations were performed. These optimizations are listed by platform in Table 4.1. Loop unrolling transforms a loop to perform fewer iterations while doing more work per iteration. This amortizes loop overhead and can improve pipelining in several cases. Since each kernel in a streaming language writes to a single element, loop unrolling is often irrelevant. However, both Brook+ and CAL allow eight outputs to be written per kernel invocation. As such, the kernel can be written to compute an element in eight tiles per invocation. This optimization is referred to as kernel unrolling in this thesis. According to the Stream Computing User Guide in the section implementing an optimized matrix multiply, “Aggregating the memory fetches per kernel significantly increases the efficiency of the stream processor.” [10] Vectorization makes use of SIMD instructions while coalescing loads and stores. Precomputing radii, basis

Table 4.1: Optimizations implemented by platform

Platform	Loop unrolling	Kernel unrolling	Vectorization	Precompute radii	Cache alphas	Cache Blocking
x86_64	Yes	No	Yes (MKL)	Yes	Yes	Yes
Brook+	No	Yes	Yes	Yes	Implicit	Implicit
CAL	No	Yes	Yes	No	Implicit	Implicit

function caching, and cache blocking all exploit high-speed cache present on processors. Specifically, these three optimizations encourage data reuse for which caches are specifically designed. The x86 implementation of the grid potential computation uses all of these optimizations except for kernel unrolling. The inner loop of the naïve algorithm is nearly fully unrolled. This is done implicitly through vectorizing $\exp(\alpha r^2)$. In fact, the inner loop exists only to provide cache blocking. Dividing the coordinates into 128 blocks of 2,048 elements was found to yield optimal performance. When this size is used, all temporary buffers fit into L1 cache and enough room is left over to allow the write to the matrix to store its result in cache. Since *vsexp()* does not use non-polluting stores, it is written to cache. This was observed by stepping through the function’s disassembly. Vectorization is implicitly used through the MKL library. All radii are precomputed and stored in a vector so they can be fully cached. Basis functions are implicitly cached in BLAS’s *sscal()* function, using a single alpha as the scalar.

In addition to these optimizations, the host implementation is further sped up through parallelization. An OpenMP *parallel for* is used on the outer loop with private buffers. Guided scheduling is used to reduce thread starvation. Work was parallelized by

rows in the result matrix, yielding a number of tasks equal to the number of basis functions used (640).

To optimize the Brook+ implementation explicitly requires only three of the optimizations presented. Passing the basis functions in as a float4 and storing the x, y, and z components of the coordinates in a float3 allows computation to be vectorized. Multiplying four basis functions by the computed radii yields 4 grid potential elements. Results are written to 8192x160 tiles of the grid potential. Each element in the grid potential is a float4. Doing this adds a constraint on the input; the number of basis functions must be a multiple of 4. In addition to allowing coalesced reads and writes, vectorization on the AMD GPUs also allows for SIMD computation. Combined with vectorization, kernel unrolling allows kernel instances to load and store more data, making them more efficient. Additionally, unrolling reduced the number of times the compute kernel must be called from the host program by a factor of eight. This optimization requires the number of coordinates used to be a multiple of 8. Unrolling was implemented by writing to eight tiles of the grid potential rather than 1. Combining these optimizations yields a kernel that produces 8 8192x160 float4 tiles of the grid potential. Precomputing radii was performed in the optimized implementation, but yielded less than a 1% speedup after the other optimizations had been applied. This was implemented by creating another kernel that performs the radii computation.

The CAL implementation of the basis function uses the same optimizations as the Brook+ version. This provides performance comparisons between Brook+ and CAL given the same implementation. Furthermore, because of their similar computational models, Brook+ and CAL readily lend themselves to the same optimizations.

Vectorization was performed using float4 data types for the grid potentials and alphas. Since CAL does not support float3 data types, a float4 was used for the coordinates with junk data being used for the fourth element. Kernel unrolling was also implemented by writing to registers o0, o1, ... o7. Like Brook+, these two optimizations yielded a kernel that produces 8 8192x160 (with the number of coordinates and basis functions used in experiments) float4 tiles of the grid potential, with alphas varying in the leading dimension. Furthermore, they added the constraints requiring the number of basis functions to be a multiple of 4 and the number of coordinates to be a multiple of 8. Radii precomputation actually hurt performance by a marginal (<1%) amount.

4.1.3 Results

Because the x86 is a traditional architecture with widespread adoption in high performance computing markets, it serves as a baseline for performance comparison. According to top500.org, the x86_64/EM64T architecture is used in 85.8% of the 500 most powerful supercomputers in the world[24].

Benchmarking the Gaussian Basis function's performance on the x86 was performed on an idle Xeon5355 box running Ubuntu 8.04. This machine has two pseudoquad-core Xeon5355 processors running at 2.66Ghz. This machine can utilize 7.5GB/s of memory bandwidth (B_{STREAM}) in the STREAM[25] copy benchmark. This test was performed with OpenMP enabled, calling memcpy() rather than the default copy code, and with gcc's level 3 optimizations enabled (-O3 flag). Each processor has 8MB of L2 cache (4MB shared between 2 cores). For the optimized implementation on the x86, the code was compiled with -fopenmp and -O3 flags. The naïve implementation

used -O0, as any optimization level used on this kernel degraded performance by almost an order of magnitude. Output buffers were zeroed with 8 threads before execution to force page-ins. This also evicted initialized data from cache. The benchmark program was run 5 times and the results given are an average of these runs. Since threads are created to evict cache before running the kernel, their creation overhead is not considered in timings. The naïve implementation always runs with 1 thread (e.g. it has no *omp parallel* construct). Results for the optimized implementation are given using 1, 2, 4, and 8 threads.

While memory is a performance bottleneck in the Gaussian basis application, computing the exponentials also takes considerable time. Equation 2 gives the latency to perform a single exponential calculation. This equation shows that the latency of an operation is the inverse of the aggregate execution rate of an operation. The aggregate execution rate is assumed to be a core's execution rate times the number of cores. On the Xeon 5100 series processor, MKL can perform one exponential in 6.92 (E) clock cycles in high accuracy mode⁵, (both the read and write vectors need to be in cache to achieve this). The X5355 in the tested setup has 8 cores (n) running at 2.66Ghz (f). This implies a latency of 0.325ns per exponential. Equation 3 shows the latency of a data transfer as a function of the bandwidth and amount of data transferred. In this case, the latency calculated is the time to store a single precision floating point number. An optimized STREAM copy benchmark was used for the aggregate memory bandwidth of the machine, since the peak bandwidth is not sustainable. Using the empirically found

⁵ <http://www.intel.com/software/products/mkl/data/vml/functions/exp.html>

bandwidth and the size of a single precision number, a store latency of 0.533ns is derived per grid potential. The modeled execution rate is approximately the inverse of the sum of the calculation time and store time. While some memory/computation interleaving occurs on an x86 processor, its finite hardware resources prevent perfect interleaving since the rate data can be stored is less than the rate it can be computed.

$$T_{\text{exp}} = \frac{E_{\text{cycles/exp}}}{f_{\text{clock}} n_{\text{cores}}} = \frac{6.92}{2.66E9 * 8} = 0.325 \frac{\text{ns}}{\text{exp}} \quad (2)$$

$$T_{\text{memwrite}} = \frac{\text{sizeof(float)}}{B_{\text{STREAM}}} = \frac{4}{7.5E9} = 0.533 \frac{\text{ns}}{\text{float}} \quad (3)$$

$$E_{\text{theoretical}} \approx \frac{1}{T_{\text{exp}} + T_{\text{memwrite}}} = \frac{1}{0.325 + 0.533} = 1.18 \frac{\text{Gexp}}{\text{s}} \quad (4)$$

The naïve algorithm performed yielded poor performance. As shown in Figure 4.3, only 25Mexp/s were achieved. This is due to several factors. Firstly, only 12.5% of the theoretical computational performance was even achievable since it was not multi-threaded. Secondly, only $\frac{1}{4}$ of the performance of a single core is achievable because vector operations are not used. Thirdly, the exponential function given in math.h is far less optimal than the one provided by MKL. Finally, the naïve kernel is not blocked for efficient cache reuse.

Actual performance for the optimized x86 implementation is given in Figure 4.4. Performance scaled linearly when doubling the number of threads, but did not scale linearly thereafter. This is due to the fact that additional threads speedup the exponential computation without increasing the maximum memory bandwidth of the machine, as the model assumes maximum bandwidth usage for any given number of cores used.

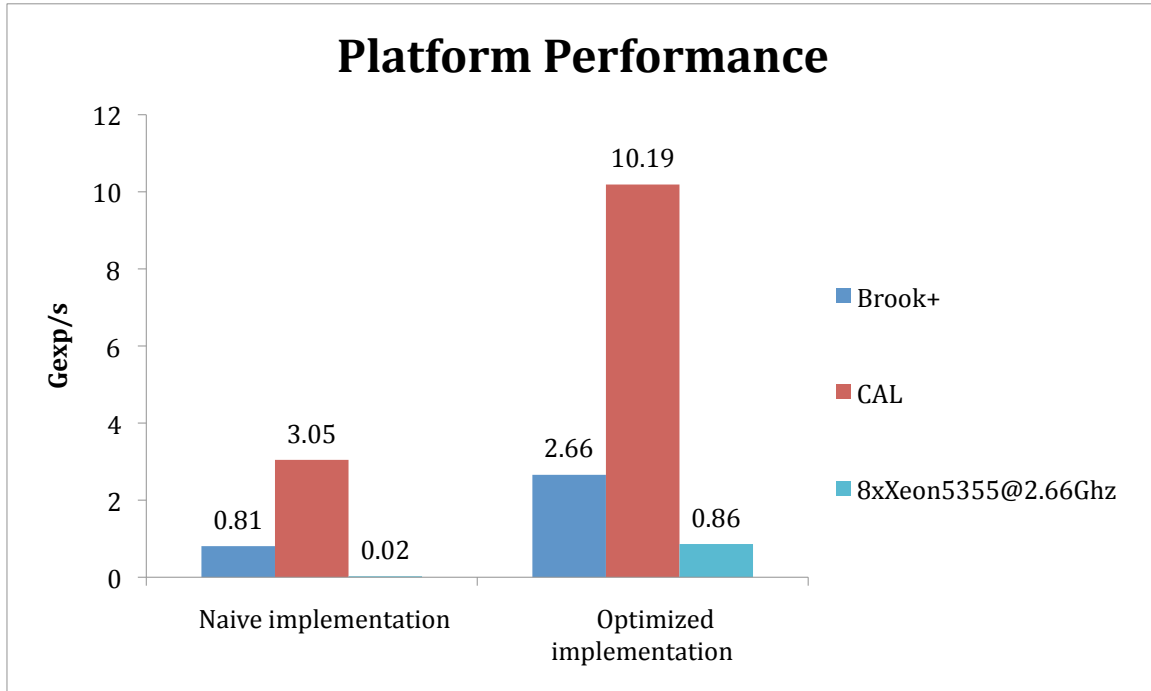


Figure 4.3: Performance of naive and optimized implementations of grid potential computation

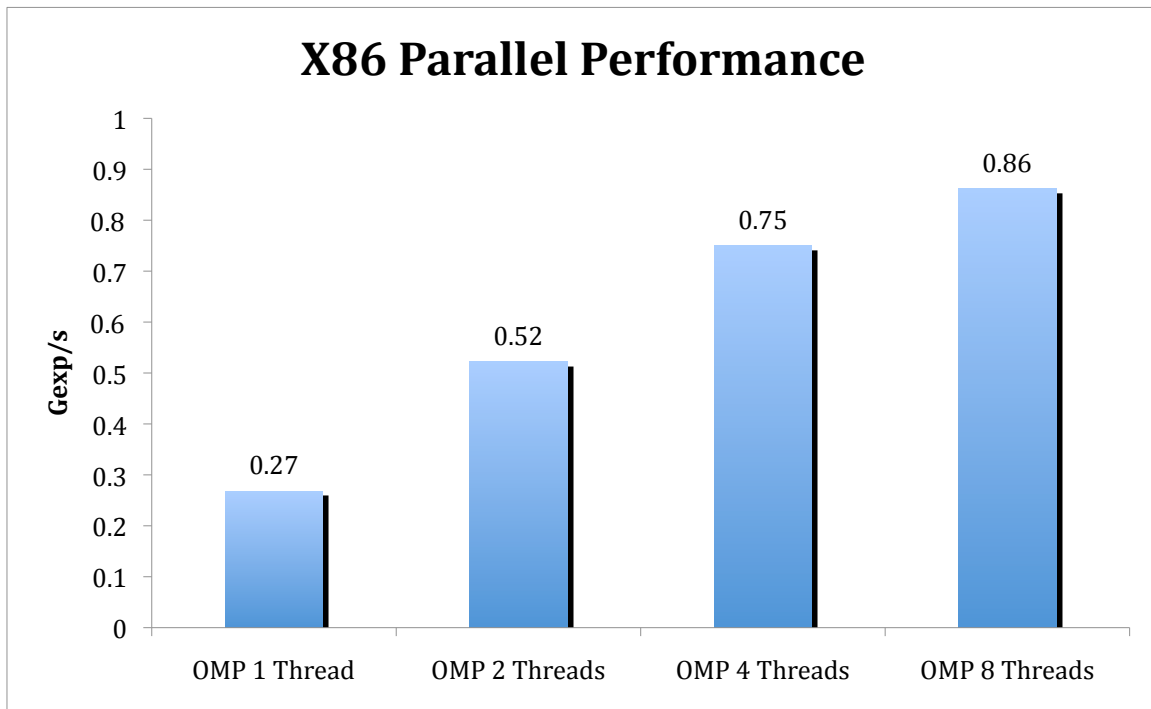


Figure 4.4: x86 performance scalability

Assuming infinite speedup in the exponential calculation, the throughput is still limited to 1.88Gexp/s on this machine as per Amdahl's law. The time to store the exponentials becomes the limiting factor. Using 8 threads, an execution rate of 862Mexp/s is achieved on this machine. This translates to 74% of the achievable performance given by the model. Since the model is optimistic in memory bandwidth, doesn't account for thread overheads, doesn't factor the time to precompute the radii, and ignores time to performed the fully cached multiplication of the basis functions by radii, 100% of the theoretical performance is not achieved in practice.

The Brook+ performance benchmark requires fewer assumptions to make a synthetic benchmark than the x86 test bench. The kernel is timed by forcing synchronization after data is transferred and after the kernels are complete. Time taken is measured starting immediately after data transfer to the GPU until all kernels complete execution (e.g. the second synchronization occurs). As such, data is assumed to reside on the GPU in these timings. Data transfers to and from the GPU are not included in the computation time. To amortize data transfer costs and realize the speedup given by this assumption, additional components of the simulation need to be moved to the GPU. This benchmark was run on a Firestream 9170 using the AMD stream SDK v1.3.

Since GPUs are able to achieve a high degree of computational parallelism (e.g. n is large in equation 2), performance is mostly limited by memory bandwidth in this application. With 51.2GB/s of bandwidth, this means the Firestream 9170 can compute at most 12.8 Gexp/s. Additionally, it has 2GB of GDDR3 memory, allowing all data to be stored on the GPU.

The Brook+ naïve implementation was easy to develop and sped up the grid

potential computation by a factor of 108 (shown in Figure 4.5). It's important to note that parallelism is inherent to the Brook+ language, unlike C. As such, the Brook+ implementation is able to achieve data-level parallelism with no additional effort. The C implementation requires explicitly declaring how to achieve parallelism through OpenMP or a threading library, which is beyond the scope of a naïve algorithm. Running this kernel on the GPU yielded 810Mexp/s (Figure 4.3). This naïve algorithm is comparable in performance to the x86 implementation using full optimization and parallelism.

When optimized, Brook+ more than tripled its performance (Figure 4.3). It was able to achieve 2.66Gexp/s. This translates to using 20.8% of the Firestream 9170's memory bandwidth. Writing to 8 tiles rather than 1 reduced the number of kernels that had to be executed and increased the efficiency of each individual kernel execution. Some of the discrepancy between theoretical and achieved performance is attributable to Brook+ overhead and compiler inefficiencies. Using more basis functions can improve performance by a small amount. Using 4096 basis functions and 65,536 coordinates yielded over 3Gexp/s of performance.

CAL's benchmark application makes the same assumptions that the Brook+ version makes. The same Firestream 9170 with SDK v1.3 is used in these tests.

The naïve CAL based algorithm yielded higher performance than the other two environments (even with optimized implementations). It achieved 3.05 Gexp/s (Figure 4.3), which is a 414-fold speedup (Figure 4.5) over the naïve reference algorithm. The high performance over Brook+ is likely due to fewer overheads associated with the kernel. When the Brook+ code was disassembled, it yielded significantly more IL instructions than the CAL kernel.

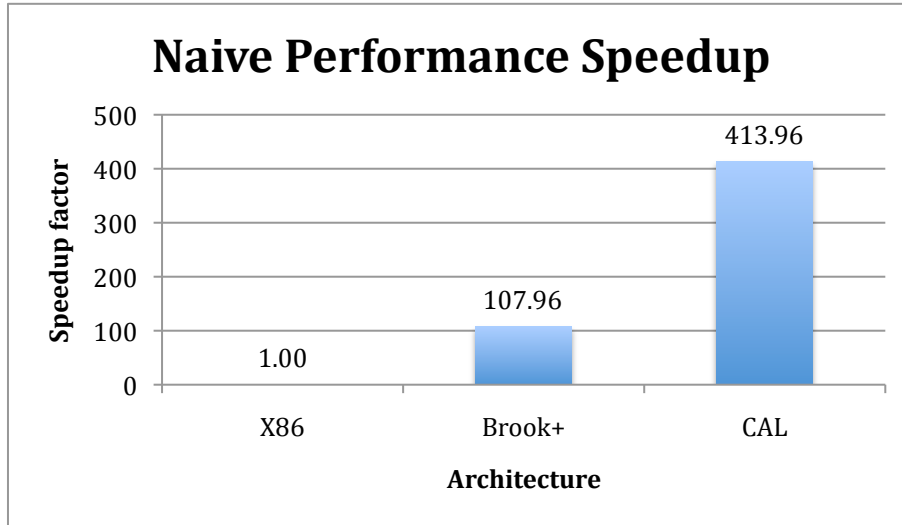


Figure 4.5: Speedup factor over x86 architecture in naive grid potential

When optimized, the CAL grid potential computation kernel furthered its performance advantage and approached the theoretical bound. A 3.35x performance gain (Figure 4.6) was had through kernel unrolling and vectorization. Radii precomputation hurt performance by approximately 0.5%. As shown in Figure 4.3, this kernel achieved 10.19 Gexp/s, 79.7% of the theoretical upper bound. When comparing all optimized kernels with one another, this implementation was 11.83x faster than the x86 implementation (Figure 4.7).

4.1.4 Other Work

Philip Brown has implemented the grid potential computation on a variety of other architectures. He has compared performance on the Clearspeed X620, Cell processor, and Nvidia 8800GTX.⁶ Applying minor modifications to his CUDA implementation (e.g.

⁶ Personal communication with Phillip Brown

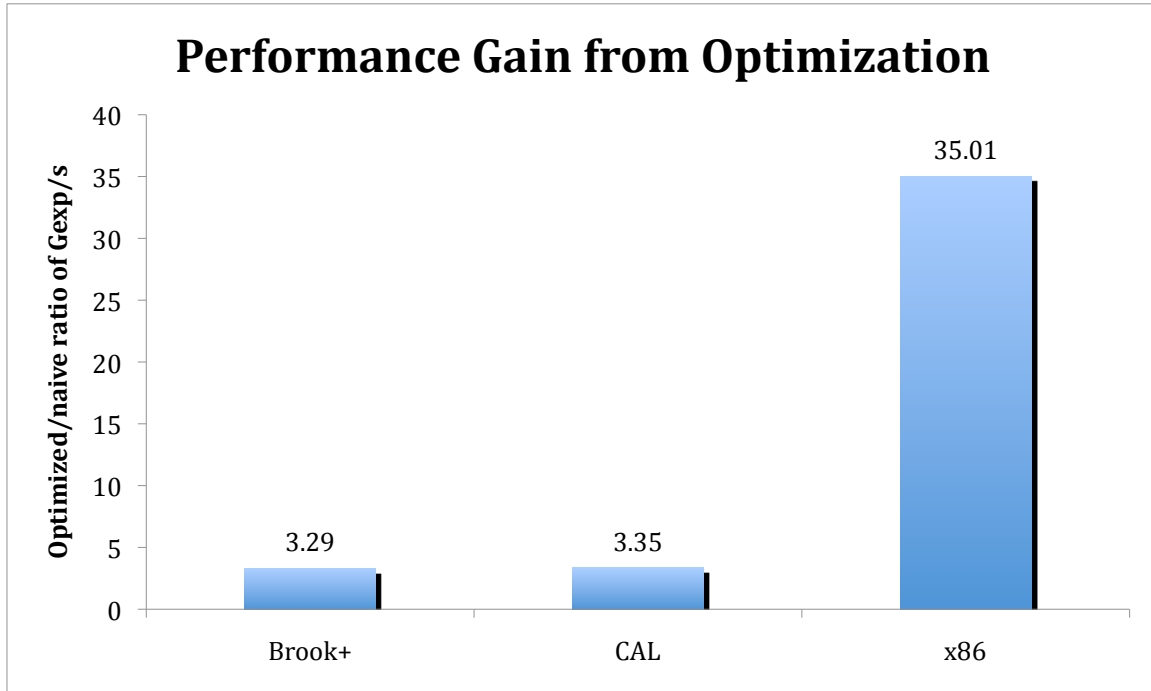


Figure 4.6: Performance increases due to optimization

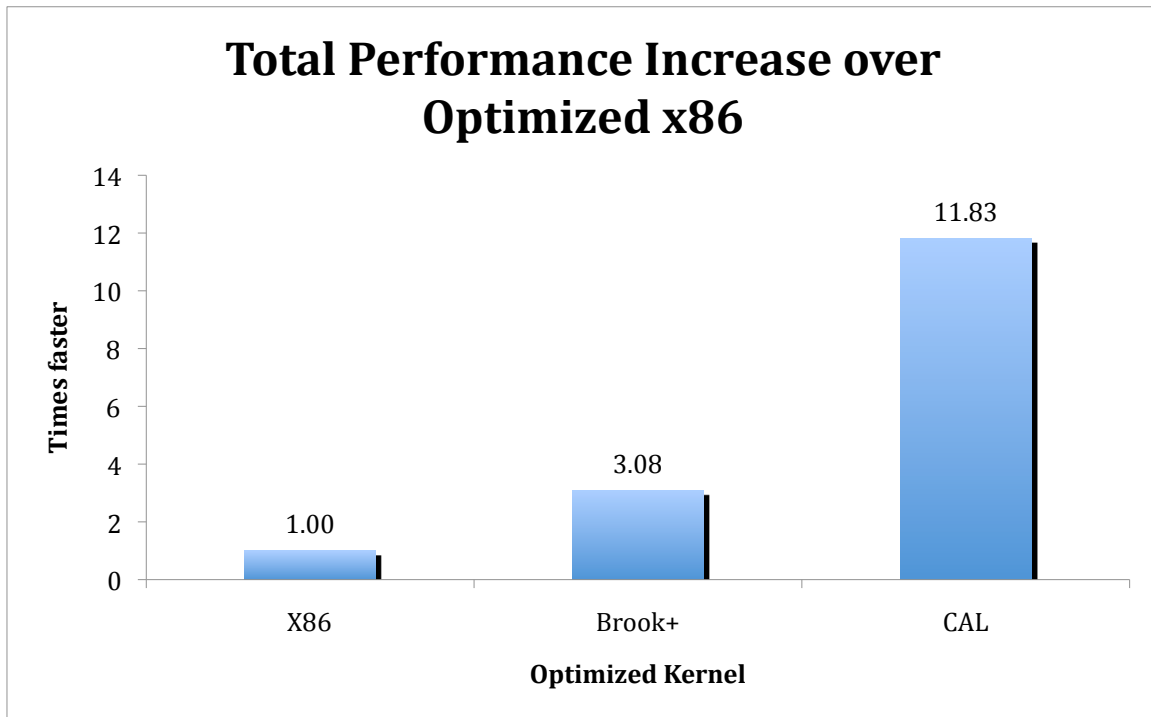


Figure 4.7: Speedup in optimized kernels over x86 optimized implementation

using `#pragma unroll` to unroll the basis function iteration) and using an Nvidia 8800 Ultra, 11.8 Gexp/s were achieved in an optimized kernel. This is slightly higher than the optimized CAL implementation. However, the 8800 Ultra has 103.7 GB/s of memory bandwidth, nearly double that of the Firestream 9170. Furthermore, the kernel is conceptually more complex, using shared memory and thread blocking.

4.2 Quantum Monte Carlo (QMC)

Monte Carlo methods are used in computation when a deterministic algorithm is impractical due to computational complexity. This can be extended to approximate integration of multi-dimensional functions. Monte Carlo integration replaces an integration grid with a sampled version of the grid. The points in this sampled grid are modified over time using random numbers. At each discrete time interval, an estimate of the grid interval is formed. The mean and variance of these estimates can then be computed. Increasing the number of samples improves the estimate by allowing the grid to be more accurately represented.[26]

This paper looks at accelerating the variational Monte Carlo method. Typically, the term “quantum Monte Carlo” refers to either the variational or diffusion method. The variational method uses a trial wavefunction (e.g. a guess as to what $\psi(x)$ should be). The trial wavefunction parameters are then tweaked to find the most accurate representation of the true wavefunction ($\psi_0(x)$). A Monte Carlo approach is then used to compute the high-dimensional integrals.[26]

An algorithm for establishing the minimum energy is given in Figure 4.8. The most computationally intensive part of this is step 2, which requires running the Monte Carlo

method to get the expected value. Only step 2 was implemented for this paper to give a performance expectation on a variety of architectures; minimizing the wave function remains as future work and is beyond the scope of this thesis.

To find the expected energy of the system, the grid is randomly sampled a number of times and atomic movements are simulated for some number of time steps. Each simulation of time steps is referred to as a walker. Having each walker run on a separate processor trivially parallelizes QMC. While each iteration time step performs N^2 computations, buffers can be accumulated in place, using $O(N)$ memory. Since memory use is limited to small reusable buffers, cache can be efficiently used and memory traffic can be minimized.

The wave function, potential, and kinetic energy can be computed in a single kernel call, which reuses radius calculations, or broken up into several calls. While using several calls requires recomputing atomic distances, separating the wave function calculation from the other two can reduce computation in the long run. Since atomic perturbations are accepted or rejected based on the computed wave function, the potential and kinetic energy calculations can be omitted if the atom's movement is rejected (future work).

1. Create a trial many-particle wave function parameterized by the n parameters $\alpha=(\alpha_0, \alpha_1, \dots, \alpha_n)$. This wave function is dependent on the coordinates of the particles.
2. Find the expected energy
3. Apply some minimization function on α to revise the wavefunction.

Figure 4.8: Algorithm for Variational Monte Carlo[27]

4.2.1 Naïve Implementation

The quantum Monte Carlo algorithm computes the wave function, potential, and kinetic energies of a molecular system. This algorithm is implemented in Brook+ using single precision arithmetic. Akila Gothandaraman and Lee Warren at the University of Tennessee originally wrote a naïve implementation using double precision for CPUs⁷. Single precision was found to be significantly slower than double on the CPU (taking approximately 66% longer to complete) and far too inaccurate to be useful. This same issue occurred in the Fortran reference implementation. The performance issue was investigated, but not resolved. A single precision MKL implementation was also attempted, which was still slower than the double precision MKL implementation. The single precision accuracy issues occurred when performing the reduction of pair-wise contributions into the total calculation.

The naïve C implementation of QMC serves as a comparison baseline. The pair-wise computations are iterated over using two loops. The outer loop determines which atom is first in the each pair-wise computation. The second loop iterates only over all preceding atoms. This saves unnecessary computation (the interaction of particle m with n is the same as n with m). Computed values are updated in place to minimize memory traffic and improve cache reuse.

Because values are updated in place, the naïve QMC compute kernel is computationally bound by the speed at which the potential, Laplacian, wave function, and gradient can be calculated. These functions are called $O(N^2)$ times, as there are $O(N^2)$

⁷ Personal communication with Akila Gothandaraman

pair-wise particle interactions. For problems sizes feasibly computed on today's architectures, memory accesses are low cost because all data can fit in cache (possibly L1). The quantum force vectors and the particle coordinates are the largest items used by the compute kernel, each of which uses $O(N)$ memory. The remaining objects used in the naïve computation are of size $O(1)$. However, translating this kernel into a streaming model in an intuitive way sacrifices the low memory access overhead by storing all pair-wise contributions and summing them.

A natural way to express the QMC algorithm in the streaming model is to create a domain encompassing all pair-wise interactions. Doing this yields 2-D streams of individual potential, kinetic, wave function, and quantum force contributions. Each element in the potential stream, for example, represents potential energy contributed by a single pair-wise interaction. These matrices of partial contributions can be reduced to produce the potential energy, kinetic energy, or wave function of the whole system. Unfortunately, this naïve approach uses $O(N^2)$ memory, which may require fetching data from main GPU memory rather than cache.

The partial potential energy, wave function, and kinetic energy calculations are stored in a lower triangular matrix with zeros on the diagonal. The element at row i column j denotes the resulting interaction of particles i and j . The diagonal is zero because these elements imply a particle interacting with itself. When computing the partial energies and wave functions, $F(r_{ij})$ is equal to $F(r_{ji})$, (where F is either the potential energy, wave function, or kinetic energy). This would imply that the partial computation matrix is symmetric. However, each interaction is included only once so they are not doubly counted in the total interaction. Computing quantum forces is more

complicated in a streaming environment.

Quantum forces are more complex to compute. Ultimately, the quantum force needs to be stored as three vectors (in this implementation, the quantum force is stored in a single vector containing float3 elements) so it may be used to compute the total kinetic energy. To accomplish this, the partial quantum forces are computed and stored in the lower triangle of a matrix with the upper triangle and diagonal being zero. The easiest way to create the quantum force vectors is to perform a row gather operation on the matrix. This matrix can be created from the partial quantum forces by doing a column reduction on them and placing the resulting vector on the diagonal of the quantum forces matrix. To reduce memory bandwidth usage, this matrix isn't actually created. Rather, the column gather is stored in a temporary register and then subtracted from the row gather.

4.2.2 Optimized QMC

How one optimizes the quantum Monte Carlo algorithm is architecture dependent. The x86 implementation can be optimized to better exploit data level parallelism by using SIMD computations and multiple processors. In a streaming environment, where data parallelism is expressed implicitly through the language, optimizing memory usage (usually at the expense of redundant computation) can speed up the application.

To substantially improve performance on the host CPU, data and task level parallelism must be exploited. This is achieved in two ways: running several instances of the application to exploit multi-core processors, and using MKL to implicitly use the SIMD units on the x86 processor. The former optimization is trivial to implement; n copies of the applications evaluate m/n ensembles with a different seed, where m is the

total number of ensembles. As such, this is also performed in the naïve implementation. The results can then be averaged between the n processes. The latter requires removing the inner loop and replacing it with vector operations. This is simple to do when branching doesn't occur. Since the potential function is defined to be 1 for $x \geq D$, a branch instruction is needed to check for this. To overcome this, elements in the vector are all computed assuming this is false. The algorithm then iterates over this vector again, writing the value 1 at elements for which this condition is true. While this creates extra computation, it allows optimized MKL vector math library functions to be used. To keep memory use down, the potential energy, kinetic energy, wave function, and quantum forces are updated in place by adding vector reductions. By performing the entire loop in vectors, cache is efficiently reused which minimizes memory traffic (for a feasibly computable number of particles). Because of the increased number of buffers over the naïve algorithm, cache performance may be worse than the naïve implementation for larger clusters.

Optimizing QMC in Brook+ requires rethinking what an element in a stream should be. Memory use can be reduced to $O(N)$ by performing a partial reduction in place and storing data in a 1-d stream. Under this strategy, element i of the potential energy stream represents the sum of the potential energy calculations that result from the pair-wise interactions of particle i with particles i through N .

Quantum forces can also be stored in a 1-d stream. Since individual pair-wise interactions are not stored, each pair-wise quantum force contribution is computed twice (once for the i^{th} row gather and once for the j^{th} column gather). Since the number of computations performed is still $O(N^2)$, this redundant computation is well worth the

memory use reduction as the number of particles evaluated increases. Furthermore, this memory reduction allows up to 65 million particles to be simulated simultaneously without marshalling data.

Reducing memory usage has the side effect of reducing the implicit parallelism expressed in the Brook+ algorithm. When using $O(N^2)$ memory, N^2 threads are created. Reducing the number of elements per stream to $O(N)$ reduces the number of threads to N . This can cause the optimized implementation to perform worse than the intuitive implementation for small numbers of particles.

4.2.3 QMC Performance

The CPU implementations of QMC use double precision floating point due to the aforementioned single precision performance and accuracy issues. The GPU implementations do not suffer from accuracy problems to the same degree as the x86 because Brook+ makes associativity and commutivity guarantees in reduction kernels[10]. Since AMD's Brook+ development environment lacks double precision transcendental functions⁸, QMC was implemented in single precision on this platform. A double precision implementation remains as future work when Brook+ is updated to support 64-bit floating-point transcendental functions.

CPU timings were achieved in the same environment as was used in the grid potential application. Eight Xeon Clovertown cores running at 2.66GHz with 4MB L2 cache were used. These were distributed across two sockets. The test machine has 32GB of DDR2 RAM and can achieve an optimized STREAM copy benchmark of 7.5GB/s.

⁸ As of Brook+ version 1.4, described in the release notes

GCC 4.2.4 for X86_64 linux-gnu was used to compile all benchmarks. All CPU and GPU implementations used the `-O3`, `-g`, and `-m64` flags.

When run on the CPU, 50 iterations were used per process with 10 sample points in the integral grid. Eight copies of the process were run concurrently to use multiple cores for both the MKL and naïve implementations. CPU implementation timings given in Table 4.2 list the maximum runtimes of the eight execution times. Together, these eight processes performed 400 iterations of 10 points in the QMC method. A processes execution time is measured as the time between the main function’s stack frame creation and just before the application’s termination.

Timings gathered on the GPU were run using Brook+ v1.3 on a Firestream 9170. To perform the same amount of work as the CPU benchmark, 400 iterations were run on the GPU with 10 grid samples. The time listed encompasses the entire process runtime in seconds.

Performance varied across implementations with the number of particles. MKL achieved highest performance when 512 atoms were used. Indeed, even the naïve CPU implementation beat out both GPU implementations for this cluster size. There are

Table 4.2: Timings for QMC Kernels (Fastest in Bold)

Time (s)	512 Atoms	1024 Atoms	2048 Atoms	4096 Atoms
Naïve CPU	13.086871	54.040897	217.229559	852.711762
MKL CPU	9.799197	39.472391	168.918917	819.450457
Naïve Brook+	28.215723	33.475018	89.333456	316.697359
Optimized Brook+	27.825212	45.438193	52.141224	139.652555

several factors that contribute to this: Brook+ runtime overhead and data transfers. When a 1024 atom cluster was simulated, the memory-bound GPU implementation performed best. This is likely due to the increased parallelism and ability to hide memory latencies. This implementation was slightly faster than the optimized CPU implementation. As the number of atoms increased to 2048 and 4096, the computationally bound Brook+ implementation performed computation most quickly. The performance speedup of each implementation is compared against the naïve CPU and MKL implementations in Figure 4.9 and Figure 4.10 respectively.

The gap between naïve and MKL CPU implementations narrowed when the number of atoms increased to 4096. MKL only improved the naïve implementation by 4% at this size. At this size, the MKL implementation can't fit all its vectors in L1 cache.

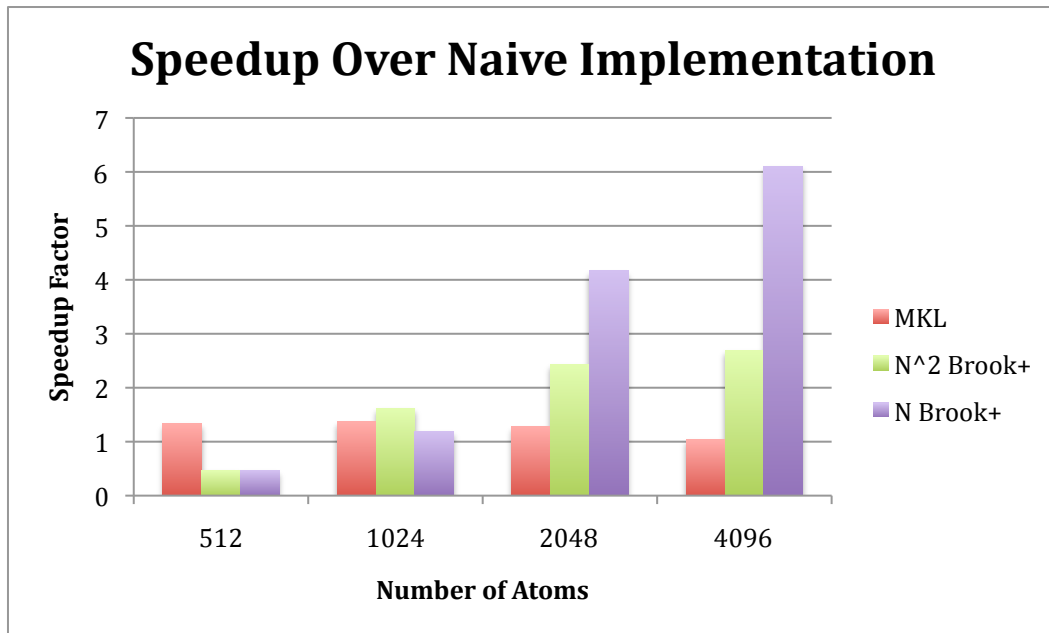


Figure 4.9: Performance Improvement over Naive Implementation

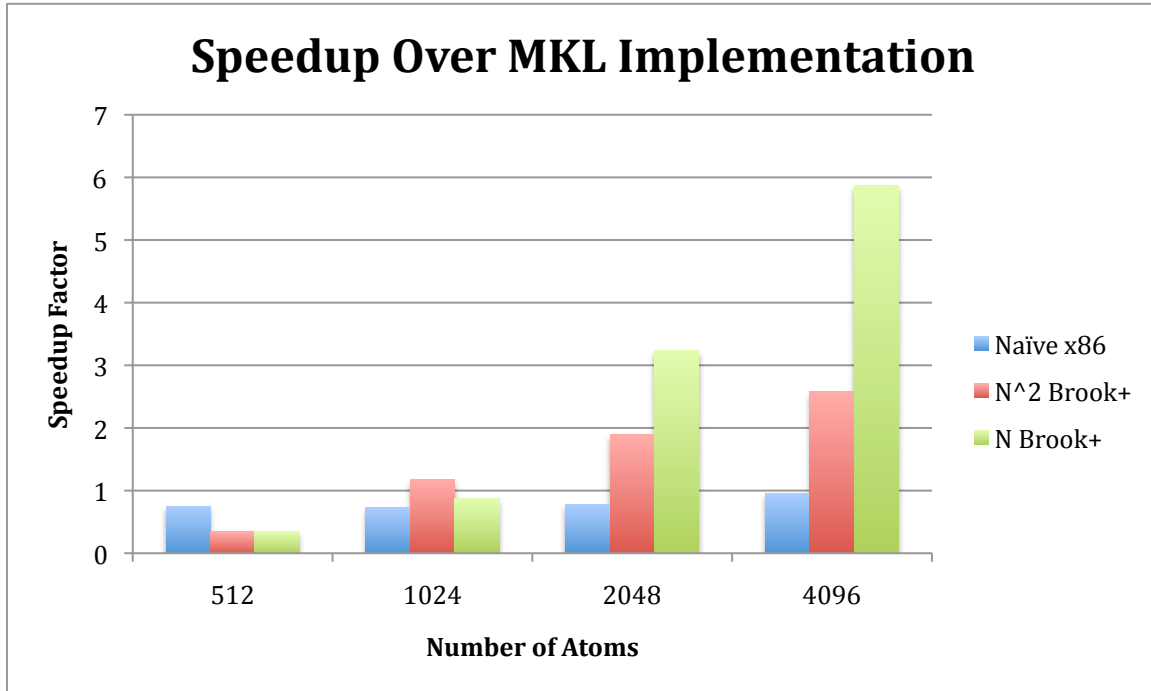


Figure 4.10: Performance Improvement over MKL Implementation

To understand where time is being spent on execution of the QMC algorithm, a performance model (Figure 4.11) is used. This model allows extrapolation of how performance will change with additional computation resources or a larger number of particles. As the number of particles increases, computation begins to dominate the runtime. This means the runtime follows a quadratic relation (Figure 4.12) to the number of atoms in the cluster. The GPU implementations of this algorithm execute more quickly than the C versions. This is because the computation is distributed across more processors. The naïve Brook+ algorithm is slower for large numbers of atoms compared to the optimized implementation because memory accesses are more expensive. In the optimized algorithm, memory accesses can better fit in cache since the size of the data is less; this reduces load and store latencies.

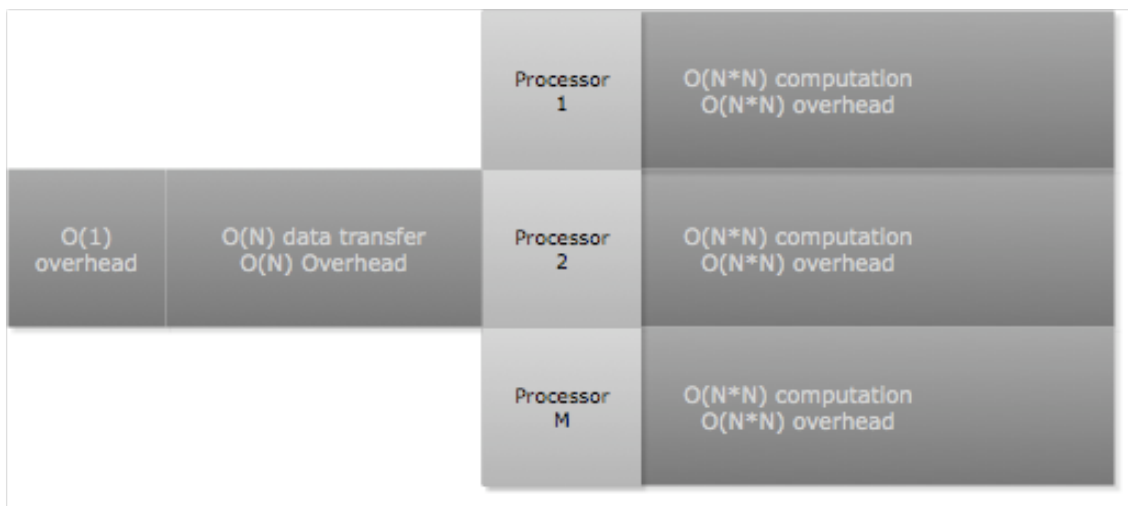


Figure 4.11: Performance Model of QMC Runtime

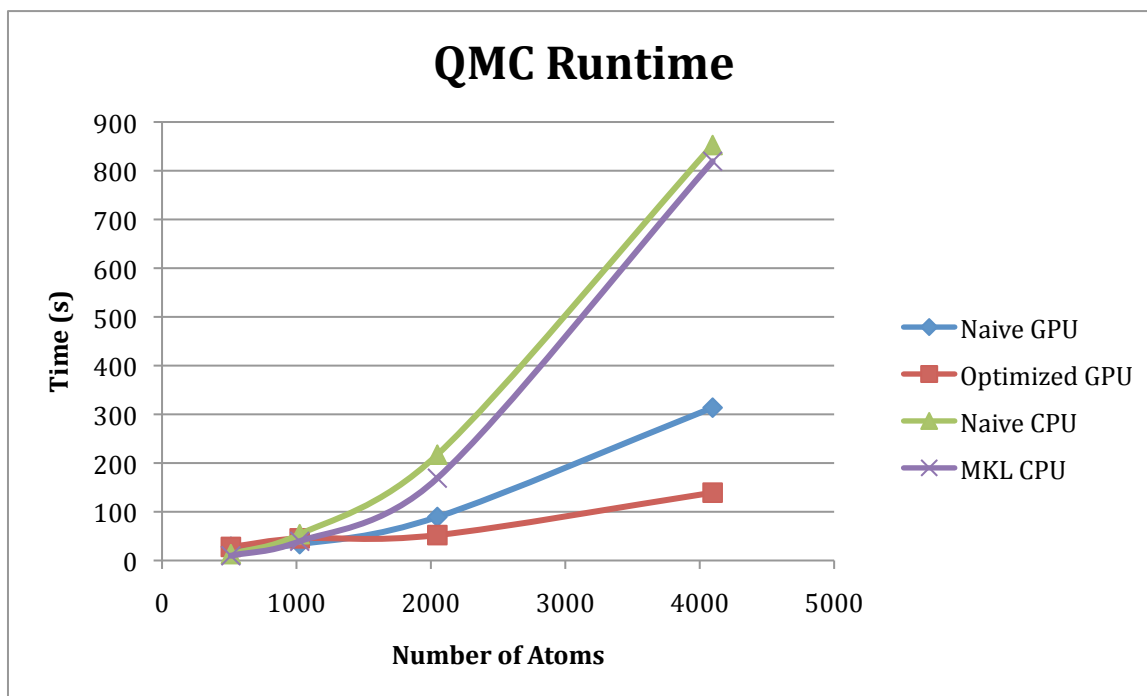


Figure 4.12: QMC Runtime

5 Conclusions

This thesis has shown that AMD GPGPUs can be used to obtain speedups in both memory and computation bound applications of sufficient complexity. Using AMD's CAL runtime, a 10-fold speedup was achieved over eight Clovertown processors in the grid potential calculation. Again, this speedup assumes this is part of a larger compute kernel that can either amortize or eliminate the data transfer of the grid potential back to the host. A 5.86x speedup was obtained in the QMC application for cluster sizes of 4096.

Brook+ and CAL were evaluated (the latter in only the first application) for performance. This thesis has shown how to map two different algorithms to the streaming model and how to optimize implementations in Brook+ and CAL. In the grid potential computation, I was able to use almost 80% of the GPU's memory bandwidth (the limiting factor) in CAL. This was done without explicitly managing texture caches, shared memory, or designating how threads should block the problem. Few optimizations were needed to give this performance and they were simple to implement. Brook+, while lagging behind CAL's performance in the grid potential computation, was still able to outpace the x86 implementation using MKL and multi-threading. The optimizations performed to the Brook+ kernel to maximize performance were conceptually simpler to implement and fewer in number. In the QMC application, Brook+ yielded almost a 6x speedup over QMC running on 8 x86 cores (which were achieving a nearly linear speedup over a single core).

5.1 Future Work

Some work remains with these applications. To complete the comparison of Brook+ to CAL, the QMC code should be implemented in CAL. In addition to CAL, OpenCL would provide an interesting language comparison for both applications presented in this thesis. This will demonstrate the price paid for portable algorithms and partially answers how well OpenCL can be mapped to GPU and multi-core architectures. Intel's Larrabee can run the x86 versions of the algorithms presented in this paper. This would provide a more even comparison, given that the performance of Larrabee will be more comparable to that of a GPU than current multi-core processors[28]. This will also partially answer the following question of how multi-core processors should be implemented in terms of heterogeneity.

References

1. John D. Owens, M.H., David Luebke, Simon Green, John E. Stone, and James C. Phillips, GPU Computing. Proceedings of the IEEE, 2008. 96(5): p. 879-899.
2. Ian Buck, T.F., Daniel Horn, Jeremy Sugerman, *Brook for GPUs*. 2003.
3. Vadim Demchik, A.S. *Monte Carlo simulations on Graphics Processing Units*. 2009 [cited 2009 March 19]; Available from: <http://arxiv.org/abs/0903.3053>.
4. Wall, D.W., *Limits of Instruction-Level Parallelism*. 1993, Digital Equipment Company: Palo Alto.
5. McKee, S.A. *Reflections on the Memory Wall*. in *Proceedings of the 1st conference on Computing frontiers*. 2004. Ischia, Italy.
6. Reinders, J. *Are multicore processors here to stay?* [cited 2009 April 2]; Available from: <http://www.zdnetasia.com/videos/whiteboard/0,3800010902,62032144,00.htm>.
7. Moore, S.K. *Multicore is Bad News for Supercomputers*. IEEE Spectrum Online November 2008 [cited 2009 April 2]; Available from: <http://spectrum.ieee.org/nov08/6912>.
8. Hagersten, E. *Finding the Door in the Memory Wall Part 1*. 2009 March 3 [cited 2009 April 2]; Available from: <http://www.hpewire.com/features/Finding-the-Door-in-the-Memory-Wall-Part-1-40654172.html?viewAll=y>.
9. Nvidia, *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. 2.0 ed. 2008.
10. *AMD Stream Computing User Guide*. 2008, AMD.
11. Group, K., *The OpenCL Specification*. 2009.
12. Anthes, G. *Timeline: A brief history of the x86 microprocessor*. 2008 [cited 2009 April 2]; Available from: <http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9091018>.
13. *SSE Performance Programming*. 2008 [cited 2009 April 2]; Available from: <http://developer.apple.com/hardware/drivers/ve/sse.html>.
14. Intel. *Intel(R) Microarchitecture Codename Nehalem*. 2009 [cited 2009 April 2]; Available from: <http://www.intel.com/technology/architecture-silicon/next-gen/>.

15. Dorit Nuzman, R.H. *Multi-platform Auto-vectorization*. in *Proceedings of the International Symposium on Code Generation and Optimization*. 2006: IEEE Computer Society.
16. *OpenMP Application Programming Interface*. 2008, OpenMP Architecture Review Board.
17. *Intel® Math Kernel Library (Intel® MKL) 10.1 In-Depth*. [cited 2009 April 2]; Available from: <http://software.intel.com/en-us/intel-mkl/>.
18. *Board Specification: Tesla C1060 Computing Processor Board*. 2009, Nvidia.
19. Stokes, J. *Introduction to Multithreading, Superthreading and Hyperthreading*. 2002 [cited 2009 April 2]; Available from: <http://arstechnica.com/old/content/2002/10/hyperthreading.ars>.
20. *Nvidia CUDA: Programming Guide*. 2008, Nvidia.
21. Dally, W. *Stream Computing*. [Lecture] 2008 June 10, 2008 [cited 2009 March 23]; Available from: <http://www.youtube.com/watch?v=8x7OqiUNbyo>.
22. *Compute Abstraction Layer (CAL) Technology Intermediate Language(IL) Reference Manual*. 2008, Advanced Micro Devices.
23. Leach, A.R., *Molecular Modeling*. 1996, Harlow, Essex, England: Addison Wesley Longman Limited.
24. *Processor Family share for 11/2008*. 2008 [cited 2009 April 8].
25. McCalpin, J.D., *Memory Bandwidth and Machine Balance in Current High Performance Computers*. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, 1995.
26. Warren, G.L., *Overcoming Statistical Error and Bias in Quantum Monte Carlo: Application to Metal-Doped Helium Clusters*, in *Chemistry*. 2005, University of Tennessee: Knoxville. p. 165.
27. Thijssen, J.M., *Computational Physics*. 1999, Cambridge, UK: The Press Syndicate of the University of Cambridge. 546.
28. Larry Seiler, D.C., Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jerry Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Pat Hanrahan, *Larrabee: A Many-Core x86 Architecture for Visual Computing*. ACM Trans. Graph, 2008. **27**(3).

Vita

Rick Weber was born and raised in Knoxville, TN. After graduating Powell High School in 2003, he went on to complete a Bachelor's degree at the University of Tennessee in 2007. His research interests include embedded systems, reconfigurable hardware, heterogeneous architectures, and high performance computing. Weber is currently pursuing a PhD at the University of Tennessee.