Programmability: Design Costs and Payoffs using AMD GPU Streaming Languages and Traditional Multi-Core Libraries

Rick Weber, Robert J. Harrison, Gregory D. Peterson
University of Tennessee
Knoxville, TN

***Abstract* – GPGPUs and multi-core processors have come to the forefront of interest in scientific computing. Graphics processors have become programmable, allowing exploitation of their large amounts of memory bandwidth and thread level parallelism in general purpose computing. This paper explores these two architectures, the languages used to program them, and the optimizations used to maximize performance in a memory bound chemistry application.**

Keywords-Brook+, CAL, multi-core, GPU, stream computing

I. Introduction

The multi-core paradigm has become the primary method for providing performance improvements in processors. As Instruction Level Parallelism (ILP) has begun to yield diminishing returns on transistor investments [1], multiple copies of processors (cores) are being placed onto a single die to exploit Thread Level Parallelism (TLP). While desktop users see benefit in a multi-tasking environment, single-threaded applications must often be recoded to take advantage of the multiple processors appearing in desktops. This presents a new set of challenges in algorithm design; programs must now account for parallelism and the pitfalls that come with it. In a different context, Graphics Processing Units (GPUs) have become programmable. Traditionally, GPUs had fixed pipelines for assembling vertices, mapping textures, and rasterizing scenes to create 3D environments for games and CAD applications. To allow for more complex effects in rendering, such as bump mapping and complex lighting, vendors replaced the fixed pipeline with programmable shader units. In doing so, they allowed the high bandwidth memory and parallel shader units to be used for general purpose computing [2]. Previous work has shown how to optimize applications written in AMD's streaming environment [3]. While many GPU-accelerated HPC applications have focused on CUDA, some applications have been implemented on AMD hardware [4].

Our work presented in this paper compares the paradigms and languages used to develop for these platforms. Specifically, we compare the streaming model behind AMD's Brook+ and Compute Abstraction Layer (CAL) development environments and compare this to C, a traditional sequential language. To do this, the grid potential computation used in Ab Initio modeling serves as a memory bound application for exploration. This paper provides qualitative assessments of programmability, such as ease of use and how cleanly optimizations [3] are written in these languages. Additionally, quantitative performance results are given for naïve and high performance GPU and multi-core kernel implementations. These results are compared to what is theoretically achievable on these architectures.

II. AMD GPU Architecture

AMD's GPUs use a hierarchy of processors to provide massive parallelism (Fig. 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 [5]. Finally, each thread processor contains 5 stream cores, which serve as the ALUs that perform actual computation. Some

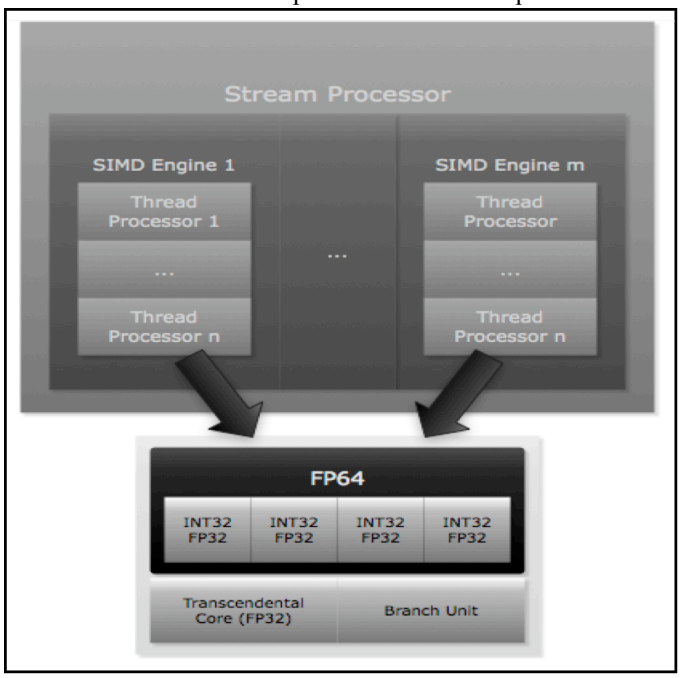

Figure 1: AMD GPU Architecture

GPUs have multiple stream processors per board, such as the Radeon 4870x2.

III. Brook+

Brook+ is a high level stream computing language developed by AMD. It is based on the Brook project at Stanford University [6]. Brook+ is a C-like programming language using 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 are inlined at compile time and 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 output stream(s). The developer can manually change this if needed. Kernel instances are created for each point in the domain of execution and may write only to the instance's location in an output stream (shown in Fig. 2). The streaming model provides implicit parallelism and separates communication from computation [7]. Depending on how an input stream is declared, a kernel may read from any location or only the current domain instance location. Streams can be declared as input or output; input streams can only be read from while output streams can only be written to [5].

IV. Compute Abstraction Layer

The Compute Abstraction Layer (CAL) is a low-level streaming environment for performing computation. Like Brook+, it uses a streaming model for processing data. 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 computation. The global buffer is 128-bit addressed and can be read or written to by any thread at any index [5].

CAL is divided into two components: the CAL runtime and Intermediate Language (IL). The runtime serves as the front end 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). In addition to these, a feature exists allowing a stream to be allocated on the host in any address specified. 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).

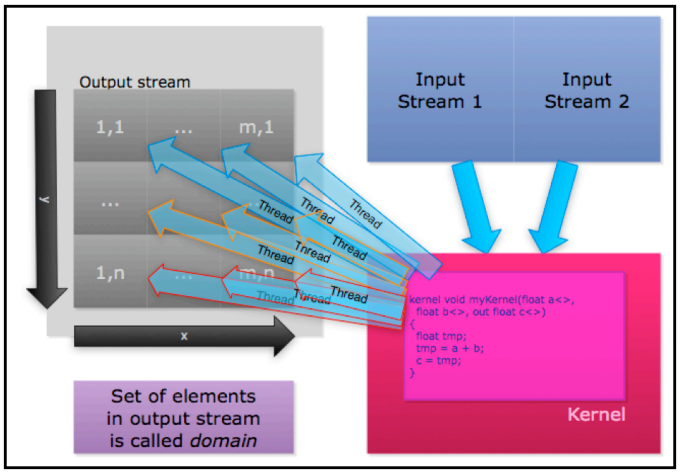

Figure 2: Streaming Model

V. Overview of Grid Potential

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 nuclii. Fortunately, Gaussian functions can serve as reasonable approximations to Slater functions and are more conducive to computation in a model. Gaussian functions take the form shown in Eq 1, where alpha is the radial spread of a Gaussian function and r is the distance between the centers of two orbitals [8].

$$\exp(-\alpha r^2) \tag{1}$$

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.

VI. Language Programmability

Streaming languages provide implicit parallelism. Since the kernel is invoked once for each element in the domain of execution, developers need only express the computation of a single element. Traditional sequential languages, like C, require users to explicitly define the parallelism granularity and patterns. This is often a difficult task, even with libraries like OpenMP [9]. Brook+ and CAL are simpler than C in this respect, since threading is handled by the hardware. Additionally, Brook+ and CAL hide architectural details, as opposed to Nvidia's CUDA, which allows explicitly using shared memory. In some cases, this may prevent the developer from fully exploiting hardware capabilities, but this does not affect the application presented in this paper. Brook+ is simpler than CAL for practical reasons; it hides API calls into elegant class abstractions while CAL requires numerous C API calls to perform operations. Both CAL and Brook+ presently lack double precision transcendental operations (as of SDK v1.4).

VII. Optimization

We created naïve and fast (though not necessarily optimal) implementations of the grid potential kernel in Brook+, CAL, and C. The naïve C algorithm is given in Fig. 3. The optimizations applied to each kernel are given in TABLE 1. Kernel unrolling refers to having the kernel write to eight streams instead of one. This increases kernel efficiency [5] by increasing texture unit utilization [3]. SIMD instructions are exploited in this application by using Intel's Math Kernel Library (MKL) vector routines on the host machine and float4 data types on the GPU. Precomputing the radii eliminates redundant computation. Its performance benefit in Brook+ was very negligible and actually hurt performance in CAL. Cache blocking is done on the host machine to maximize L1 cache reuse in the inner loop. Finally, the host code was parallelized using an OpenMP parallel for construct with guided scheduling.

TABLE 1: Optimizations Performed on Platforms

Optimization	Platform		
	C (x86)	Brook+	CAL
Kernel unrolling	No	Yes	Yes
SIMD	Yes (Intel Math Kernel Library)	Yes	Yes
Precompute radii	Yes	Yes	No
Cache alphas	Yes	No	No
Cache Blocking	Yes	No	No

VIII. Results

Performance results shown in TABLE 2 highlight the performance advantage of AMD's GPU over the multi-core CPU, despite a simpler programming model. The C implementations were run on an 8 core X5355 machine. The CPU implementations were compiled using gcc 4.2.4. Using optimizations (O1, O2, or O3) significantly reduced the computation rate in the naïve kernel. As such, results given for this kernel have optimizations disabled. Our results for the fast CPU implementation use gcc's –O3 flag, though this

```
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 3: Naive Grid Potential Kernel

provided little benefit (most of the optimizations are already in MKL). Memory latency (modeled with an optimized STREAM copy benchmark) in storing the basis function and MKL's published exponential computation time account for 74% of the execution time. GPU results were taken on a Firesream 9170 with Stream SDK version 1.3. The optimized CAL implementation used 80% of the Firestream's 51.2GB/s of memory bandwidth. Brook+, while outperforming the CPU even in the naïve implementation, suffered from compiler overhead not present in the CAL version. The short nature of this kernel prevented Brook+ from amortizing this overhead. All results given are for single precision computation on an idle machine. Brook+ and CAL results do not include data transfers to and from the GPU. When this is included, the kernel becomes PCIe bound and becomes slower than performing computation on the CPU. As such, more of the application needs to be moved to the GPU to minimize PCIe traffic

.

TABLE 2: Platform Performance of Naive and Optimized Kernels

Billion points per second	Kernel	
	Naïve	Optimized
C	0.02	0.86
CAL	3.05	10.19
Brook+	0.87	2.66

IX. Conclusions

We have shown that the streaming model simplifies computation without sacrificing performance in the grid potential application. Using CAL, we were able to use 80% of the card's memory bandwidth. The simplicity of Brook+ and CAL programming languages provide large speedups with little effort. The streaming model eliminates the need to explicitly parallelize applications. By far, the most complicated optimizations were in the C implementation, which required complex cache blocking and leveraging libraries to most easily exploit parallelism (both SIMD and thread parallelism). While CAL is conceptually simple, its driver level API makes programming more difficult than Brook+.

[1]. D.W. Wall, "Limits of instruction-level parallelism," Digital Equipment Company. Palo Alto, 1993.

[2] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, "GPU computing". In Proceedings of the IEEE, vol 96, **pp**. 879-899, May 2008.

[3] B. Jang, S. Do, H. Pien and D. Kaeli, "Architecture-Aware Optimizations Targeting Multithreaded Stream Computing," Proceedings of GPGPU-2, ACM Digital Library, 2009.

[4] V. Demchik and A. Strelchenko, "Monte Carlo simulations on graphics processing units," unpublished 2009. http://arxiv.org/abs/0903.3053

[5] AMD Stream Computing User Guide. 2008.

[6] I. Buck, T. Foley., D. Horn, J. Sugerman, "Brook for GPUs," December 2003.

[7] W. Dally, Stream Computing. June 2008.

[8] A.R. Leach, Molecular Modeling, Harlow, Essex, England: Addison Wesley Longman Limited, 1996.

[9] OpenMP Application Programming Interface. OpenMP Architecture Review Board. 2008,