A Memory Model for Scientific Algorithms on Graphics Processors

Naga K. Govindaraju [*] [†] Scott Larsen [*] Jim Gray [†] Dinesh Manocha [*]

{naga,larsene,dm}@cs.unc.edu, Jim.Gray@microsoft.com

Microsoft Technical Report MSR TR 2006 108

Abstract

We present a memory model to analyze and improve the performance of scientific algorithms on graphics processing units (GPUs). Our memory model is based on texturing hardware, which uses a 2D block-based array representation to perform the underlying computations. We incorporate many characteristics of GPU architectures including smaller cache sizes, 2D block representations, and use the 3C's model to analyze the cache misses. Moreover, we present techniques to improve the performance of nested loops on GPUs. In order to demonstrate the effectiveness of our model, we highlight its performance on three memory-intensive scientific applications – sorting, fast Fourier transform and dense matrix-multiplication. In practice, our cache-efficient algorithms for these applications are able to achieve memory throughput of 30–50 GB/s on a NVIDIA 7900 GTX GPU. We also compare our results with prior GPU-based and CPU-based implementations on high-end processors. In practice, we are able to achieve 2–5× performance improvement.

Keywords: Memory model, graphics processors, scientific algorithms.

1 Introduction

The programmable graphics processing units (GPUs) have been shown useful for many applications beyond graphics. These include scientific, geometric and database computations. The GPUs are programmable parallel architectures designed for real-time rasterization of geometric primitives. Current GPUs can offer 10× higher main memory bandwidth and use data parallelism to achieve up to 10× more operations per second than current CPUs. Furthermore, GPU performance has improved faster than Moore's Law over the last decade, so the GPU-CPU performance gap is widening.

In this paper, we address the problem of efficient implementation of scientific algorithms on GPUs. The GPUs have been used for solving sparse and dense linear systems, eigen-decomposition, matrix multiplication, fluid flow simulation, FFT, sorting and finite-element simulations [2005; 2004]. The GPU-based algorithms exploit the capabilities of multiple vertex and fragment processors along with high memory bandwidth to achieve high performance. Current GPUs support 32-bit floating point arithmetic and GPU-based implementations of some of the scientific algorithms have outperformed optimized CPU-based implementations available as part of ATLAS or the Intel Math Kernel Library (MKL). However, there is relatively less work on developing appropriate memory models to analyze the performance or designing cache-efficient GPU-based algorithms.

Current GPU architectures are designed to perform vector computations on input data that is represented as 2D arrays or textures. They achieve high memory bandwidth using a 256-bit memory interface to the video memory. Moreover, the GPUs consist of multiple fragment processors and each fragment processor has small L1 and L2 SRAM caches. As compared to conventional CPUs, the GPUs have fewer pipeline stages and smaller cache sizes. The GPUs perform block transfers between the caches and DRAM-based video memory. As a result, the performance of GPU-based scientific algorithms depends on their cache efficiency.

Main results: We present a memory model to analyze the performance of GPU-based scientific algorithms and use this model to improve their cache efficiency. We take into account architectural features of the GPUs including memory representations, data processing and memory addressing capabilities of GPUs to design our model. Our model is based on texturing hardware that uses 2D block-array representation to transfer the data between the caches and the video memory. Moreover, we use the well-known 3C's model [1989] to analyze the cache misses and evaluate the memory performance of scientific and general purpose algorithms on GPUs. Based on this model, we present efficient tiling algorithms to improve the performance of three applications: sorting, fast Fourier transform and dense matrix multiplication. We compare their performance with prior GPU- and CPU-based optimized implementations. In practice, our algorithms are able to achieve 30–50 GB/s memory throughput

[*] UNC Chapel Hill

[†] Microsoft Corporation

SC2006 November 2006, Tampa, Florida, USA

0-7695-2700-0/06 $20.00 ©2006 IEEE

on a NVIDIA 7900 GTX GPU, which costs around $600. Moreover, we have observed 2–5× performance improvement over optimized CPU-based implementations running on high-end dual 3.6 GHz Xeon processors or dual Opteron 280 processors, which cost around $2,000.

Organization: The rest of the paper is organized in the following manner. We give a brief overview of prior work on cache efficient algorithms, scientific libraries and GPU-based algorithms in Section 2. Section 3 describes our memory model and presents a technique for efficient implementation of nested loops on GPUs. We analyze our memory model in Section 4 and use it to improve the performance of sorting, matrix multiplication and FFT algorithms. We compare their performance with prior algorithms in Section 5.

2 Related Work

In this section, we give a brief overview of prior work on CPU memory models, scientific libraries and GPU-based algorithms.

2.1 CPU-based Memory Models

Modern computers use hierarchies of memory levels, where each level of memory serves as a *cache* for the next level. One of the widely used memory model is the *two-level I/O-model* defined by Aggarwal and Vitter [1988] that captures the main characteristics of a memory hierarchy. The two-level I/O-model consists of a fast memory called cache of size M and a slower infinite memory. Data is transferred between the levels in blocks of consecutive elements. By concatenating multiple two-level I/O-models, we can model a memory hierarchy with multiple levels.

The problem of designing cache-efficient algorithms has received considerable attention over last two decades in theoretical computer science, compilers and computer architecture. These algorithms include theoretical models of cache behavior [2001; 2002] , and compiler optimizations [1994]; all of these can minimize cache misses [1995]. Many of these optimizations are implemented in current compilers.

At a high level, cache-efficient algorithms can be classified as either cache-aware or cache-oblivious. Cache-aware algorithms utilize knowledge of cache parameters, such as cache block size [2001]. On the other hand, cache-oblivious algorithms do not assume any knowledge of cache parameters [1999]. There is considerable literature on developing cache-efficient algorithms for specific problems and applications, including numerical programs, sorting, geometric computations, matrix multiplication, FFT, and graph algorithms. Most of these algorithms reorganize the data structures for the underlying application, i.e., computation reordering. More details are given in a recent survey [2004].

2.2 Scientific Libraries and Compiler Optimizations

Scientific and numerical libraries are typically designed using a layered approach with good data reuse. The main idea is to identify a set of core operations for which algorithms with good data reuse are known, carefully implement these algorithms on the hardware and use those operations to develop application programs. Examples of such scientific libraries include LAPACK [1992] and ATLAS[1] for linear algebra software, FFTW[2] to compute the discrete fourier transform, and Intel's Math Kernel Library (MKL), which is highly optimized for Intel processors.

Many algorithms have been proposed in programming languages and compiler literature to generate blocked code to achieve higher performance based on the memory hierarchies in the machines. This includes work on restructuring based on space *tiling* [1987] and *linear loop transformations* [1990; 1993]. These approaches are typically restricted to perfectly nested loops, and can be extended to imperfectly nested loops if these loops are first transformed into perfectly nested loops through the use of *code sinking* [1995]. Carr and Kennedy [1992] propose a list of transformations, including strip-mine-and-interchange, index-set-splitting, and loop distribution, which are based on the control flow of the program. Other approaches directly reason about the flow of data through the memory hierarchy [1997]. Many memory models have also been proposed to estimate program performance for nested loops [1991; 1995].

2.3 GPU-based Algorithms

GPUs have been shown useful for many scientific, geometric and database computations. This includes work on using GPUs for linear algebra computations including matrix multiplication [2001; 2003; 2004], and sparse matrix computations [2003; 2003]. Sparse matrix computations are *iterative* methods, such as Jacobi iteration [2005] and conjugate gradient methods [2003]. The core operations of these algorithms are either local stencil operations or matrix-vector multiplication and vector dot products. Recently, GPU-based algorithms have also been proposed for LU decomposition on dense matrices [2005]. Other scientific computations include fluid flow simulation using the lattice Boltzman model [2004], cloud dynamics simulation [2003], finite-element simulations [2001], ice crystal growth [2003], etc. For an overview of recent work, we refer to Lastra et al. [2004] and Owens et al. [2005].

GPUs have also been used for sorting and database operations [2004; 2005; 2003; 2004]. These algorithms implement bitonic sort on the GPU as a fragment program and each stage of the sorting algorithm is performed as one rendering pass. The efficiency of these algorithms is governed by the number of instructions in the fragment program and

[1] http://www.netlib.org/atlas
[2] http://www.fftw.org

the number of texture operations.

Some high-level programming interfaces based on streaming languages have been proposed to program the GPUs, including BrookGPU [2004] and Sh [2004]. These interfaces use the programmable features of GPUs and attempt to hide the underlying aspects of graphics hardware.

There is relatively less work on developing good memory models and cache-efficient algorithms for GPUs. Some of the work has focused on analyzing the performance of GPU-based matrix-matrix multiplication algorithms. Hall et al. [2003] propose a cache-aware blocking algorithm for matrix multiplication on the GPUs. Their approach only requires a single rendering pass by using the vector capabilities of the hardware. Fatahalian et al. [2004] have shown that matrix-matrix multiplication can be inefficient on prior GPUs due to low cache bandwidth limitations.

3 GPU Memory Model

In this section, we give a brief overview of GPU architectures. We present our memory model to analyze the performance of GPU-based algorithms and highlight some of the differences with CPU-based memory models and optimization techniques.

3.1 Graphics Processors (GPUs)

GPUs are mainly designed for rapidly transforming 3D geometric primitives into pixels on the screen. Overall, they can be regarded as massively parallel vector processors. The recent introduction of programmability enables the GPUs to perform many kind of scientific computations efficiently. In this section, we briefly describe the data representations used by these scientific algorithms and the underlying mechanisms used to access the data and perform the computations on GPUs:

- ***Memory Representation:*** The graphics processor is designed to perform vector computations on input data represented as 2D arrays or textures. Each element of a texture is composed of four color components, and each component can store one floating point value. Current GPUs only support 32-bit floating point representations. The scientific algorithms represent the input data in 2D textures and perform streaming computations on the data elements in the 2D textures.
- ***Data processing:*** In order to perform computations on a data element, a quadrilateral covering the element location is rasterized on the screen. The rasterization process generates a fragment for each covered element on the screen and a user-specified program is run for each generated fragment. Since each fragment is evaluated independently, the program is run in parallel on several fragments using an array of fragment processors. The output of the fragment processor can be written to the corresponding element location through a high bandwidth memory interface. Some of the main benefits of the GPU arises from the fact that current GPUs offer $10\times$ higher main memory bandwidth and use data parallelism to achieve up to $10\times$ more operations per second than current CPUs.
- ***Memory addressing:*** The fragment processors access the input data representations (or textures) using the texture mapping hardware. The texturing hardware maps the elements in the input 2D arrays to the data element locations on the screen. The mapping is specified by rasterizing a quadrilateral that covers the element locations on the screen and each vertex of the quadrilateral is associated with a texture or 2D array coordinates. The texture mapping hardware performs bilinear interpolation of the array coordinates to compute the mapped coordinates for each pixel that is covered or rasterized. A 2D lookup is then performed on the 2D input array, and the data element at the array location is assigned to the fragment.

3.2 GPU Memory Model

Current GPUs achieve high memory bandwidth using a 256-bit memory interface to the video memory. The textures used in GPU rendering operations are stored in a DRAM-based video memory. When a computation is invoked, the fragment processors access the data values from the DRAM using texturing hardware. In order to mask high DRAM latencies, a block transfer is performed to small and fast L1 and L2 SRAM caches, which are local to the fragment processors. These prefetch sequential block transfers utilize the memory bus efficiently. This is one of the major reasons that GPUs are able to obtain 10x higher memory bandwidth as compared to current CPUs.

Given this memory organization, the 2D texture array on GPUs is represented using a 2D block-based representation for rasterization applications [1997]. In our block-based model, we assume that the 2D array is tightly partitioned into non-overlapping 2D blocks of size $B \times B$. Each 2D block represents a cache block and when an array value corresponding to the 2D block region is accessed, our model assumes that the entire block is fetched into the cache if it is not present. This 2D block-based representation is designed to exploit the spatial locality of texture memory accesses in graphics applications. Moreover, memory addressing in GPUs is performed using bilinear interpolation capabilities of the texturing hardware. As a result, the memory accesses for a 2D region of pixels correspond to a 2D block of texture addresses that have spatial coherence. In practice, the block-based representation efficiently exploits the spatial locality in memory accesses.

As compared to current CPUs, the GPUs have fewer pipeline stages. Therefore, the GPUs are able to better hide the mem-

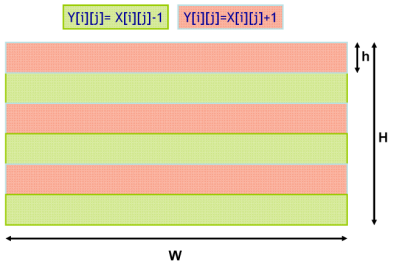

Figure 1: This figure shows a color-coding of the regions corresponding to increment and decrement operations in Y while executing the nested loops in routine 3.1. The orange colored regions indicate increment operations and green-colored regions represent decrement operations.

ory latency as compared to the CPUs. This is the main reason that GPUs have smaller cache sizes than CPUs (e.g. one order of magnitude smaller). Due to the small cache sizes on GPUs, only a few blocks can fit at any time in the cache. As a result, the GPU memory organization is quite different than that of CPUs. In order to analyze the GPU cache behavior, we incorporate the well known 3C's model [1989] into our memory model. In the 3C's model, cache misses are classified as:

1. Compulsory or cold misses which are caused due to the first reference of a block that is not in cache.
2. Capacity misses which occur due to the limited cache sizes.
3. Conflict misses that are due to multiple blocks that map to the same set.

Unlike the CPU vendors, the GPU vendors currently do not disclose the cache sizes, replacement policies, or bandwidth. As a result, the 3C's model is well-suited to analyze and improve the cache behavior of scientific applications on GPUs as it does not assume such cache information. In particular, we focus on minimizing the capacity and conflict misses because compulsory misses are unavoidable.

3.3 Nested Looping and Quadrilateral Rasterization Cache Analysis

Nested loops are commonly used in many scientific algorithms. In this subsection, we show nested loops can be implemented efficiently on GPUs. There is considerable literature on cache analysis on CPUs to optimize data locality in nested loops [Wolfe et al. 1995; Carr and Kennedy 1992]. On GPUs, implementing nested loops is analogous to quadrilateral rasterization. However, GPUs use different memory representations. Therefore, memory optimization techniques designed for CPU-based algorithms may not directly apply to GPUs. In this section, we use our memory model and CPU-based strip-mining algorithms to analyze the differences in optimized code generated for nested loops on CPUs and GPUs.

C-Based CPU Cache-Efficient Nested Loop Example

```
1   s = 0
2   for(i = 0; i < H/2h; i = i + 1)
3       for(j = 0; j < h; j = j + 1) // loop to increment
4           for(k = 0; k < W; k = k + 1)
5               Y[s][k] = X[s][k] + 1
6           s = s + 1
7       for(j = 0; j < h; j = j + 1) // loop to decrement
8           for(k = 0; k < W; k = k + 1)
9               Y[s][k] = X[s][k] - 1
10          s = s + 1
```

Analogous GPU Cache-Inefficient Nested Loops

```
1   s = 0
2   for(i = 0; i < H/2h; i = i + 1)
3       Set fragment program to increment
4       Draw a rectangular quad with co-ordinates (s,0), (s,W), (s+h,W), (s+h,0)
5       s+ = h
6       Set fragment program to decrement
7       Draw a rectangular quad with co-ordinates (s,0), (s,W), (s+h,W), (s+h,0)
8       s+ = h
```

ALGORITHM 3.1: *Implementation of nested loops on CPUs and GPUs. The different computations on the GPUs are performed using fragment programs.*

We use a simple nested loop example in C programming language (see Algorithm 3.1) to explain the difference. In this example, each data element in an input array X is accessed once. We either increment or decrement the elements in X and store the result in the output array Y. Fig. 1 shows a color-coding of Y, where the orange color represents regions in Y when the elements in X are incremented and green color represents regions where the elements in X are decremented. Suppose the width of the array is W and height of the array is H. Also, let $W \gg B$ where B is the block size. Suppose the height of the orange or green regions is h. As the data accesses are sequential and CPU cache lines are 1-D, the CPU looping code is efficient for data locality.

The GPU-based nested looping is analogous to the CPU code, where a single loop traverses the array from the top-to-bottom. Within each loop iteration, we rasterize a quadrilateral either to increment the data values using a fragment program in orange-colored regions or to decrement the data values using a second fragment program in green-colored regions. Although the CPU-code is efficient for memory accesses on CPUs, the corresponding GPU code has significant memory overhead when $h < B$ due to conflict and capacity misses. Due to the limited cache sizes, we observe that each quadrilateral rasterization could result in many cache evictions. In fact, majority of the blocks fetched earlier during rasterization are evicted by the later blocks irrespective of the cache replacement policy. Using our memory model, we analytically determine the number of cache misses to be $\frac{W \times H}{B \times h}$ and the number of cold misses to be $\frac{W \times H}{B \times B}$. In section 4, we present experimental and theoretical analysis on the GPU cache performance as a function of the cache pa-

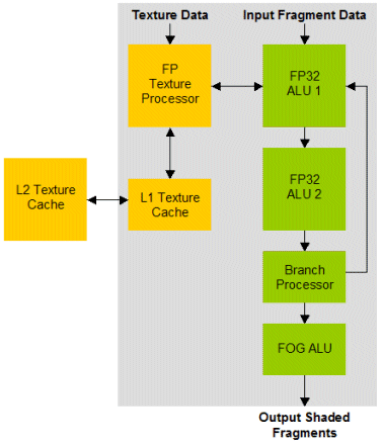

Figure 2: Texture caches on a commodity GPU: NVIDIA GeForce 7800 : It has 24 programmable fragment processors. The fragment processors have a high memory bandwidth interface to the video memory. The GPU has a core clock of 450 MHz and a memory clock of 1.2 GHz, and can achieve a peak memory bandwidth of 38.4 GBps. Each fragment processor has access to a local L1 texture cache and multiple fragment processors share accesses to a small L2 texture cache.

rameters for nested loops such as Algorithm 3.1 in scientific computations.

4 GPU Memory Model: Analysis and Applications

In this section, we first use our memory model to identify the GPU block sizes and cache sizes for measuring the memory efficiency of scientific algorithms in terms of the cache misses. We also present improved algorithms for three memory-intensive applications—bitonic sort, fast Fourier transforms and matrix multiplication algorithms and compare their performance against fast CPU-based algorithms on high-end SMP machines.

4.1 Sorting and Caching

Sorting is a fundamental data management operation and has been studied for more than five decades. Sorting is a compute-intensive and memory-intensive operation—therefore, it can utilize the high compute and memory throughput on GPUs. In this section, we analyze the problem of bitonic sorting networks [2006; 2003; 2004].

Bitonic sorting network performs data-independent comparisons on bitonic sequences [1968]. Given a sequence $a = (a_0, a_1, \ldots, a_n)$, the bitonic sorting algorithm proceeds in multiple stages and in each stage, it merges two bitonic sequences of equal length. Specifically, for each stage k performed in the order $k = 1, \ldots, \log n$, we merge two sequences of size 2^{k-1}.

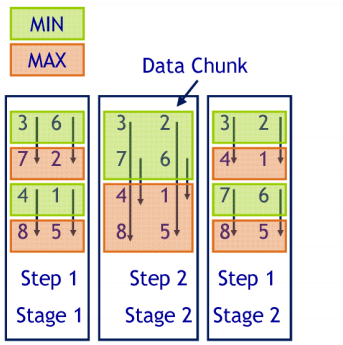

Figure 3: This figure shows the 2-D mapping of comparisons among array elements in step 2 and stage 3 on an input array of size 8. In this example, the width of the 2D array is 2 and height is 4. In each step, the array is decomposed into data chunks where minimum or maximum operations are performed. The data chunks now correspond to row-aligned quads and the sorting network maps well to the GPU 2D texturing hardware. The texturing hardware fetches the data values at a fixed distance for each pixel, and a single-instruction fragment program computes the minimum or maximum on the pixel. The minimum or maximum is computed in parallel on multiple pixels simultaneously using the fragment processors. The rasterization of the 2D quads is a nested loop similar to the code in 3.1.

In stage k, our algorithm performs k steps in the order k to 1. In each step, the input array is conceptually divided into chunks of equal sizes (size $d = 2^{j-1}$ for step j) and each elements in one chunk is compared against the corresponding element in its adjacent chunk i.e., an element a_i in a chunk is compared with the element at distance d (a_{i+d} or a_{i-d}). The minimum is stored in one data chunk and the maximum is stored other data.

The algorithm maps well to GPUs. In each step, we read values from an input array or texture, perform comparison operations using a fragment program and store the output in another array or texture. The output array is then swapped with the input array. As GPUs are optimized for 2D representations, the 1D data chunks for minimum or maximum computations are conceptually represented using row-aligned or column-aligned quadrilaterals as shown in Figure 3. We therefore, rasterize 2D quadrilaterals each corresponding to a 1-D data chunk in the step. For more details, refer to Govindaraju et al. [2006].

The overall sorting algorithm requires a large number of $O(n \log^2 n)$ compute and memory operations. Therefore, cache-analysis can significantly improve the performance of GPU-based sorting algorithms.

4.1.1 Cache Block Analysis

The 2D quadrilateral rasterization algorithm in each step is a nested loop similar to the algorithm 3.1 in Section 3.3.

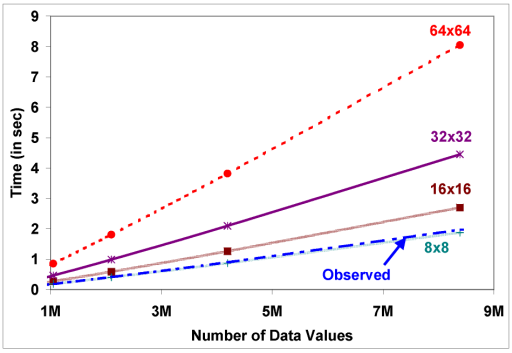

Figure 4: Our memory model for a NVIDIA 7800 GTX GPU predicts the block size for efficient sorting on GPUs. The analysis closely matches the experimental results for a cache block size of 8×8.

Each step performs two sequential read operations and one sequential write operation per data element. Using our memory model, we expect $n_{compulsory} = \frac{W \times H}{B^2}$ compulsory misses. Without loss of generality, let us assume we are rendering row-aligned quads of height h and width W. We perform cache analysis in these two cases based on the height of the row-aligned quad.

Case 1: $h \geq B$. In this case, all the cache misses in rendering the quad are compulsory misses. Note that the blocks corresponding to each row-aligned quad is accessed exactly twice. Therefore, the total number of cache misses for rendering row-aligned quads with $h \geq B$ is $2n_{compulsory}$.

Case 2: $h < B$. In this case, conflict or capacity misses can occur if n_{blocks} do not fit in the cache. This is mainly because the cache blocks fetched at the beginning of the quad are mostly evicted by the end of the quad. Within a region of $W \times B$, based on the rendering operations, each block is accessed $count(h) = \frac{2B}{h}$ times and results in $count(h)$ cache misses. As there are $n_{compulsory}$ blocks, the algorithm results in $count(h) * n_{compulsory}$ cache misses. Note that as h becomes smaller, the number of cache misses increase. Therefore, later steps in the stage have cache misses.

In the overall algorithm, step k is performed $(\log n - (k-1))$ times and $h = 2^{k-1}$ for $k = 1, \ldots, \log n$. The total number of cache misses is close to $2nf(B)$ where $f(B) = (B-1)(\log n - 1) + 0.5(\log n - \log B)^2$.

Figure 4 compares our cache model to the observed times as a function of n and B on a 7800 GTX GPU. The theoretical timings in Figure 4 is computed assuming the algorithm achieves peak sequential memory bandwidth of 40 GB/s on a NVIDIA 7800 GTX GPU. The graph indicates that our cache analysis closely matches the observed values using the block size 8×8.

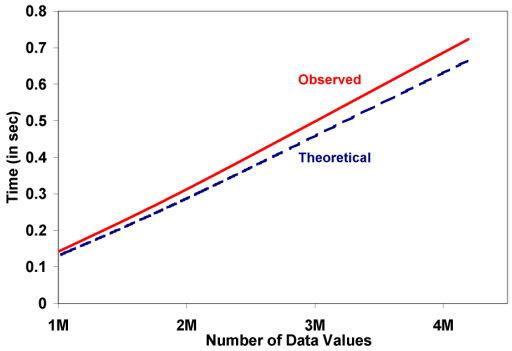

Figure 5: The computational time of our cache-efficient bitonic sort algorithm as a function of the number of data values on a 7800 GTX GPU. We observe that the experimental results closely match the theoretical results for a 64×64 block size. The graph indicates that our algorithm achieves 37 GB/s memory bandwidth, close to the 40 GB/s maximum.

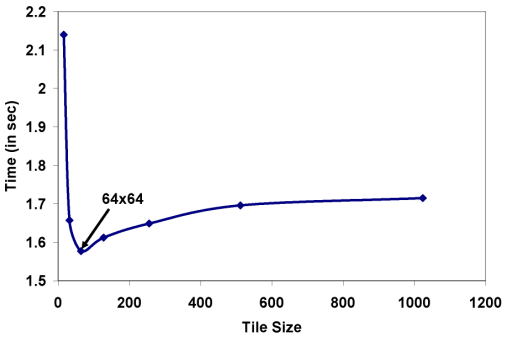

Figure 6: The performance of our cache-efficient sorting algorithm as a function of tile size sorting 8M floating point key-pointers using a 7800 GTX GPU. 64×64 tiles had the best performance. As the tile size decreases, the vertex overhead dominates the memory bandwidth savings. As the tile size increases beyond 64×64, memory performance degrades due to cache misses.

4.1.2 Cache Sizes and Cache-Efficient Algorithm

We present an improved sorting algorithm that maximizes cache utilization for given block and cache sizes. It minimizes the number of conflict or capacity misses using a technique similar to blocking. We decompose row-aligned quads with width W and height h into multiple quads of width B and height h if $h < B$. Similarly, we decompose column-aligned quads with width w and height H into multiple quads of width w and height B if $w < B$. We then perform computation on all the quads lying within the $B \times B$ block. For the remaining quads, we do not perform any row or column decomposition.

Our row and column decomposition algorithm reduces the number of cache misses to $2n_{compulsory}$ misses per step if the decomposition size matches the hardware cache size. This decomposition has an additional advantage of reducing the cache misses by fetching relevant blocks into the caches. Figure 5 highlights the observed and theoretical performance

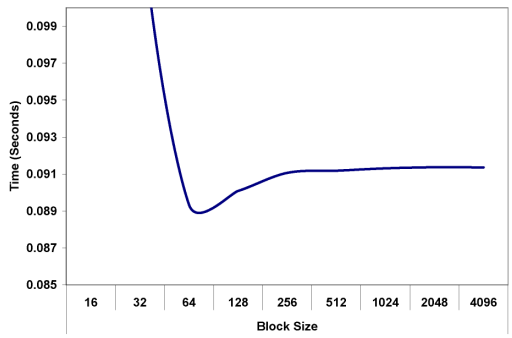

Figure 7: The performance of our cache-efficient FFT algorithm as a function of tile size on 4M complex floating point values using a 7800 GTX GPU. We obtained the best performance using $T \times T = 64 \times 64$ tiles.

of our cache-efficient algorithm as a function of n, memory and clock speeds on the 7800 GTX GPU. The graph indicates that our algorithm achieves nearly 37 GB/s memory bandwidth. This is almost 97% of the peak memory bandwidth on the 7800 GPU.

Figure 6 illustrates the algorithm's performance as a function of the decomposition block or tile size. We achieve an optimal performance at a tile size of 64×64. Our results suggest that the cache size on a 7800 GTX GPU can be close to 128 KB.

4.2 Fast Fourier Transforms

Fast fourier transforms (FFTs) is a basic building block to signal processing and frequency analysis applications. In this section, we consider the problem of large 1-D power-of-two FFTs on GPUs. Many FFT algorithms such as the Cooley-Tukey algorithm [1997] require expensive bit-reversal operations. In order to avoid bit-reversal, we use a standard Stockham formulation of the FFT. Given a sequence with n values, the Stockham FFT proceeds in multiple steps, similar to steps used in the bitonic sorting algorithm. Specifically, in step k, it performs data-independent transformations on two subsequences of size 2^{k-1} and generates a new sequence of size 2^k. The transformations are performed on an input array X and the output is stored in another array Y. At the end of each step, we swap the input and output arrays. The overall FFT algorithm proceeds in $\log n$ steps, in the order $k = 1, \ldots, \log n$. In each step k, we conceptually partition both the *output* array Y and input array X into data chunks of size 2^{k-1}. This results in $2m$ data chunks where $m = \frac{n}{2^k}$. Each element in a data chunk $2i^{th}$ or $(2i+1)^{th}$ in Y is mapped to the elements at a fixed distance in data chunks i^{th} and $(i+m)^{th}$ in X. A multiply-and-add (MAD) operation is then performed on the two elements fetched from data chunks i and $i+m$ in X and the output is stored appropriately in Y.

GPUs support vectorized MAD operations. Therefore, we can exploit the high data parallelism to compute the transformations on each data chunk in Y. The mapping of data chunks in Y to data chunks in X is a nested loop similar to the routine 3.1. We map the 1-D operations into 2D arrays on GPUs similar to the bitonic-sorting network. The mapping however, has more complex memory access patterns than in sorting. Given a 2D array representation with width W and height H, the memory access pattern for FFTs is dependent on the size of the data chunk in a step. It can be described as follows:

- Data chunk size $2^{k-1} < W$: In this case, Y is divided into column-aligned quads of width 2^{k-1} and height H. X is divided into column-aligned quads of width 2^{k-1} and height $\frac{H}{2}$. It can be seen that each column aligned quad in Y maps to four column-aligned quads in X.
- Data chunk size $2^{k-1} \geq W$: Both Y and X are conceptually divided into row-aligned quads of width W and height $\frac{2^{k-1}}{W}$. In this case, each row-aligned quad in Y maps to two row-aligned quads in X.

The overall FFT algorithm requires $O(n \log n)$ memory and compute operations. Furthermore, the memory access patterns are more complex than sorting—therefore, we can expect more benefit by performing cache analysis on FFTs.

Nested Loop Stockham FFT

1 for($step = 1; step \leq \log n; step = step + 1$)
2 for($j = 0; j < 2^{\log n - step}; j = j + 1$)
3 for($k = 0; k < 2^{step-1}; k = k + 1$)
4 $angle = -\frac{2\pi k}{2^t}$
5 out[$j2^t + k$] = in[$j2^{t-1} + k$]+ $e^{i\ angle}$ in[$j2^{t-1} + k + \frac{n}{2}$]
6 out[$j2^t + k + 2^{t-1}$] = in[$j2^{t-1} + k$] - $e^{i\ angle}$ in[$j2^{t-1} + k + \frac{n}{2}$]
7 swap(in,out)

ALGORITHM 4.1: This pseudo-code illustrates the nested loop implementation of Stockham FFT algorithm. The FFT algorithm proceeds in $\log n$ steps and during each step j, the output array is conceptually divided into data chunks of size 2^j (lines 5 and 6). Similarly, the input chunks are conceptually divided into data chunks of size 2^{j-1} and two input data chunks are mapped onto the appropriate output chunks. In terms of a GPU-based algorithm, these data chunks correspond to texture mapping row-aligned or column-aligned quadrilaterals onto row-aligned or column-aligned regions. The overall FFT algorithm involves no data reordering, requires significant computation at each data element and maps well to the affine memory addressing and vector-processing capabilities of GPUs.

4.2.1 Cache-Efficient GPU-FFT

The FFT algorithm suffers from similar cache issues as bitonic sort. However, our GPU-FFT algorithm is more memory intensive for column-aligned steps than bitonic sorting. Similar to the bitonic sorting network, we partition Y into tiles of size $T \times T$ if the height or width of the quadrilateral is less than T or $\frac{T}{2}$ respectively. We perform computation within the tile before proceeding to the next tile. Fig. 7 highlights the performance of the FFT algorithm as a function of the tile size.

The FFT algorithm is also more compute-intensive than our

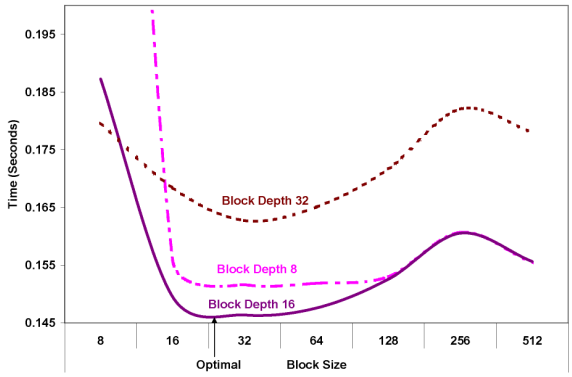

Figure 8: The performance of our cache-efficient matrix multiplication algorithm as a function of tile size for multiplying two 2K $\times$ 2K floating point matrices using a 7800 GTX GPU. We obtained the best performance using $T \times T = 64 \times 64$ tiles and a depth of 16. As the block depth decreases, the vertex overhead dominates the memory bandwidth savings. The performance at $T = D = 32$ degrades due to increase in time due to more fragment operations.

sorting algorithm. The overall FFT algorithm requires ~ 24 operations per data element whereas sorting requires ~ 10 operations per data element.

4.3 Dense Matrix-Multiplication

The problem of dense matrix multiplication is inherently parallel and highly memory intensive - therefore, it can greatly benefit from the high computational throughput and memory performance of GPUs.

Let X_{ij} denote the element at the i^{th} row and j^{th} column. Then, matrix multiplication $Z = XY$ computes elements Z_{ij} using the dot product between i^{th} row in X and j^{th} column in Y. Suppose X and Y are $n \times n$ matrices. The simplest algorithm to implement matrix-multiplication uses three nested loops. The pseudo-code for the algorithm is shown in Algorithm 4.2. Larsen and McAllister [2001] implemented the unblocked algorithm using simple blending and texture mapping functionality on GPUs. Their algorithm has $O(n^3)$ compute and memory references and is memory-bound. Hall et al. [2003] analyzed the performance of block-based matrix-multiplication algorithm for GPUs using an algorithm similar to the CPU-based algorithms. However, blocking is done only along one dimension and the resulting algorithm uses cache-efficiently if the underlying hardware performs implicit blocking.

We perform explicit blocking to avoid hardware dependencies. Moreover, our improved block-based matrix-multiplication algorithm takes into account the graphics pipeline architecture. Our algorithm decomposes the matrix Z into blocks of size $T \times T$. Computation on the tiles of size $T \times T$ is invoked by drawing quadrilaterals of size $T \times T$ on the screen. Then a single fragment program evaluates the dot product from vectors of size D in X and Y. Therefore, the time spent per element in Z depends on D and

L-M GPU-based Unblocked Nested Loop Matrix Multiplication

1 for($i = 0; i < N; i = i + 1$)
2 for($j = 0; j < N; j = j + 1$)
3 $Z_{ij} = 0$
  //Each iteration in the following loop is a quadrilateral rasterization of size $N \times N$
4 for($k = 0; k < N; k = k + 1$)
5 $Z_{ij} = Z_{ij} + X_{ik} * Y_{kj}$

Hall et al.'s GPU-based Blocked Nested Loop Matrix Multiplication

1 for($kb = 0; kb < N; kb = kb + T$)
 //following two loops invoked using a quadrilateral of size $N \times N$
2 for($i = 0; i < N; i = i + 1$)
3 for($j = 0; j < N; j = j + 1$)
4 for($k = 0; k < T; k = k + 1$) //loop performed inside a fragment program
5 $Z_{ij} = Z_{ij} + X_{ik} * Y_{kj}$

Our GPU-based Blocked Nested Loop Matrix Multiplication

1 for($ib = 0; ib < N; ib = ib + T$)
2 for($jb = 0; jb < N; jb = jb + T$)
3 for($kb = 0; kb < N; k = kb + D$)
  //following two loops invoked using a quadrilateral of size $T \times T$
4 for($i = ib; i < ib + T; i = i + 1$)
5 for($j = jb; j < jb + T; j = j + 1$)
6 for($k = kb; k < kb + D; k = k + 1$)//loop performed inside a fragment program
7 $Z_{ij} = Z_{ij} + X_{ik} * Y_{kj}$

ALGORITHM 4.2: *This pseudo-code shows the differences between our GPU-based explicit blocking algorithm and prior GPU-based matrix multiplication algorithms. The Larsen-McAllister algorithm is unblocked and performs $O(n^3)$ memory references. Hall et al. proposed an improved algorithm that performs implicit blocking. We perform explicit blocking and use a different blocking parameter for each of the inner loops. The innermost loop (line 6) in our algorithm is performed using a fragment program. The loop length in line 6 determines the number of fragment operations performed per data element per sequential write.*

achieves maximal performance when fragment processing time matches the sequential write time to the video memory. In contrast, the CPU-based matrix-multiplication algorithm performs uniform blocking along the three nested loops.

Fig. 8 highlights the performance of matrix-multiplication on GPUs as a function of T and D using a matrix of size 2K $\times$ 2K.

5 Analysis and Comparisons

In this section, we analyze the performance of our algorithm on different GPUs and compare its performance to optimized scientific libraries on CPUs.

5.1 Performance

We have tested the performance of our applications on three different GPUs – NVIDIA 6800 Ultra, NVIDIA 7800 GTX and NVIDIA 7900 GTX GPU released in successive generations (see Fig. 9). For sorting 8M key-pointer pairs, we obtain an average of 14.3 GOPS on the 7900 GTX GPU while

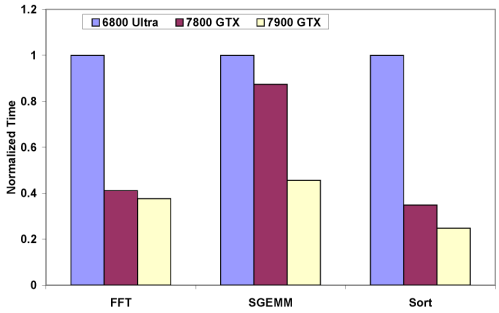

Figure 9: Normalized Performance of our cache-efficient applications on three successive generation GPUs—NVIDIA 7900, 7800 and 6800 GPUs. Our sorting, FFT and matrix multiplication algorithms are able to achieve 50, 30 and 40 GB/s memory performance respectively on a single NVIDIA 7900 GTX GPU.

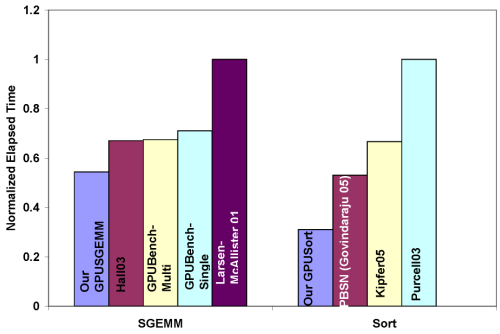

Figure 10: Normalized elapsed time of our cache-efficient applications against prior GPU-based algorithms on a NVIDIA 7900 GPU. Our cache-efficient algorithms are able to achieve 2–3× performance improvement over prior GPU-based scientific algorithms.

the 7800 GTX and 6800 Ultra GPUs achieve 10.1 and 3.5 giga-operations per second. The observed bandwidth in the sorting benchmark is nearly 50 GB/s on a 7900 GTX GPU and is nearly 92% of the peak memory throughput.

In the FFT benchmark, we have measured the GFLOPS obtained using our algorithm using the standard FFTW metric[3]. On a 4 million single precision complex FFT benchmark, we are able to obtain 6.1 GFLOPS on a 7900 GTX GPU, 5.7 GFLOPS on a 7800 GTX GPU and 2.14 GFLOPS on a NVIDIA 6800 Ultra GPU. We are able to attain a memory bandwidth of 32 GB/s on a NVIDIA 7900 GTX GPU for performing complex FFTs.

Our matrix multiplication algorithm achieves 17.6 GFLOPS on a NVIDIA 7900 GTX GPU and 9.2, 8 GFLOPS on a 7800 GTX and 6800 Ultra GPUs respectively. Our cache-efficient matrix multiplication algorithm is able to achieve nearly 40 GB/s memory performance on a single NVIDIA 7900 GTX GPU. Using NVPerfKit[4], we are able to experimentally verify that the percentage of texture cache misses in our applications is less than 6%—thus verifying that our algorithm is able to utilize the cache performance efficiently.

5.2 Comparison with Prior CPU-based and GPU-based Algorithms

We have compared the performance of our algorithm against prior GPU-based sorting and matrix-multiplication algorithms. Fig. 10 highlights the performance of our cache-optimized algorithms to sort 4M floating point key-pointer pairs or to multiply 2K × 2K floating point matrices. We observe that our matrix multiplication algorithm performs around $23-80\%$ better than prior GPU SGEMM algorithms. In the sorting application, our algorithm achieves 2–3× performance improvement over prior GPU-based sorting algorithms. We also compared the performance of our algorithm against libgpufft. Our algorithm does not require bit-reversal-based data rearrangements and therefore, it is able to achieve higher performance than libgpufft.

We have measured the performance of our algorithm against optimized `cfft1d` and `SGEMM` implementations in the Intel Math Kernel library. We used the optimized Intel quicksort routine[5] using hyperthreading and function inlining. We measured the performance of our algorithms on a SMP machine with dual 3.6 GHz Xeon processors with hyperthreading, another SMP machine with two dual-core Opteron 280 processors and a high-end 3.4 GHz Pentium IV PC with hyperthreading. We have used four threads to perform the CPU-based computations on dual Xeon and opteron processors, and two threads on the Pentium IV processor. Our results highlighted in Fig. 11 indicate that our GPU matrix multiplication algorithm on a single 7900 GTX GPU performs comparably to the dual Xeon and Opteron processors. In terms of performance/cost, the 7900 GTX GPU is 3-4x better than Xeon or Opteron processors. Our sorting algorithm is able to achieve 1.5–2× performance improvement over MKL implementation on high-end Intel processors and performs comparably to MKL routines on the AMD Opteron 280 processor. Our FFT algorithm is able to achieve 4–5× performance improvement over Xeon or Opteron processors.

6 Conclusions and Future Work

We presented a novel memory model for analyzing and improving the performance of GPU-based scientific algorithms. We have applied our memory model to three scientific applications and compared their performance against prior optimized CPU-based and GPU-based algorithms. Our results indicate a significant performance improvement using a single NVIDIA 7900 GPU.

There are several avenues for future work. We would like to

[3] http://www.fftw.org/speed

[4] http://developer.nvidia.com/object/nvperfkit_home.html

[5] http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/threading/hyperthreading/20372.htm

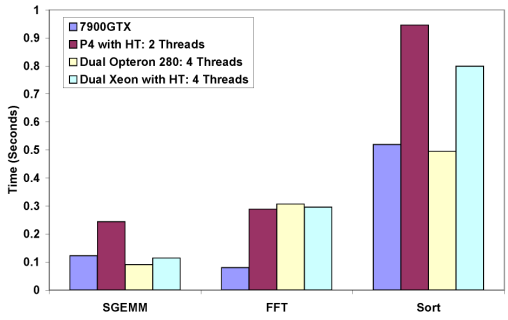

Figure 11: Performance of our cache-efficient applications on a NVIDIA 7900 GPU against optimized scientific algorithms on high-end SMP machines with dual Xeon or two dual-core Opteron processors.

incorporate our cache models into GPGPU compilers such as BrookGPU [2004] and Sh [2004]. We are interested in applying our memory model to other scientific applications and streaming architectures such as the IBM Cell processor.

Acknowledgements

This work is supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134 and 0118743, DARPA/RDECOM Contract N61339-04-C-0043, ONR Contract N00014-01-1-0496 and Intel Corporation. We would like to thank Craig Peeper, Peter-Pike Sloan, and David Blythe of Microsoft Corporation, Mike Houston, Mark Segal and Alpana Kaulgud of ATI Corporation for useful feedback. Many thanks to John Owens and Daniel Horn for providing performance numbers of libgpufft and for valuable feedback. We would also like to thanks David Tuft and other members of UNC GAMMA group for useful suggestions and support.

References

AGGARWAL, A., AND VITTER, J. S. 1988. The input/output complexity of sorting and related problems. *Commun. ACM 31*, 1116–1127.

ANDERSON, E., BAI, Z., BISCHOF, C., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., AND SORENSEN, D. 1992. *LAPACK User's Guide, Release 1.0*. SIAM, Philadelphia.

ARGE, L., BRODAL, G., AND FAGERBERG, R. 2004. Cache oblivious data structures. *Handbook on Data Structures and Applications*.

BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. 1994. Compiler transformations for high-performance computing. *ACM Comput. Surv. 26*, 4, 345–420.

BANERJEE, U. 1990. Unimodular transformations of double loops. *Proc. of the Workshop on Advances in Lanugages and Compilers for Parallel Processing*, 192–219.

BATCHER, K. 1968. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*.

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph. 22*, 3, 917–924.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph. 23*, 3, 777–786.

CARR, S., AND KENNEDY, K. 1992. Compiler blockability of numerical algorithms. *Proc. of ACM/IEEE Conference on Supercomputing*, 114–124.

COLEMAN, S., AND MCKINLEY, K. 1995. Tile size selection using cache organization and data layout. *SIGPLAN Conference on Programming Language Design and Implementation*, 279–290.

FAN, Z., QIU, F., KAUFMAN, A., AND YOAKUM-STOVER, S. 2004. GPU cluster for high performance computing. In *ACM / IEEE Supercomputing Conference 2004*.

FATAHALIAN, K., SUGERMAN, J., AND HANRAHAN, P. 2004. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association.

FRIGO, M., LEISERSON, C., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. *Symposium on Foundations of Computer Science*.

GALOPPO, N., GOVINDARAJU, N., HENSON, M., AND MANOCHA, D. 2005. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proc. ACM/IEEE SuperComputing Conference*.

GÖDDEKE, D. 2005. GPGPU performance tuning. Tech. rep., University of Dortmund, Germany. `http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/`.

GOVINDARAJU, N., LLOYD, B., WANG, W., LIN, M., AND MANOCHA, D. 2004. Fast computation of database operations using graphics processors. *Proc. of ACM SIGMOD*.

GOVINDARAJU, N., RAGHUVANSHI, N., AND MANOCHA, D. 2005. Fast and approximate stream mining of quantiles and frequencies using graphics processors. *Proc. of ACM SIGMOD*.

GOVINDARAJU, N., GRAY, J., KUMAR, R., AND MANOCHA, D. 2006. GPUTeraSort: High performance graphics coprocessor sorting for large database management. *Proc. of ACM SIGMOD*.

HAKURA, Z., AND GUPTA, A. 1997. The design and analysis of a cache architecture for texture mapping. *Proc. of 24th International Symposium on Computer Architecture*, 108–120.

HALL, J. D., CARR, N., AND HART, J. 2003. Cache and bandwidth aware matrix multiplication on the GPU. Technical Report UIUCDCS-R-2003-2328, University of Illinois at Urbana-Champaign.

HARRIS, M., BAXTER, B., SCHEUERMANN, G., AND LASTRA, A. 2003. Simulation of cloud dynamics on graphics hardware. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*.

HILL, M. D., AND SMITH, A. J. 1989. Evaluating associativity in cpu caches. *IEEE Transactions on Computers 38*, 12, 1612–1630.

KIM, T., AND LIN, M. 2003. Visual simulation of ice crystal growth. In *Proc. of ACM SIGGRAPH / Eurographics Symposium on Computer Animcation*.

KIPFER, P., SEGAL, M., AND WESTERMANN, R. 2004. Uberflow: A gpu-based particle engine. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*.

KODUKULA, I., AHMED, N., AND PINGALI, K. 1997. Data-centric multi-level blocking. *Proc. of ACM SIGPLAN*, 346–357.

KRÜGER, J., AND WESTERMANN, R. 2003. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph. 22*, 3, 908–916.

LAM, M., ROTHBERG, E., AND WOLF, M. 1991. The performance and optimization of blocked algorithms. *Proc. of 4th International conference on Architectural support for programming languages and operating systems*, 63–74.

LARSEN, E. S., AND MCALLISTER, D. 2001. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, ACM Press, 55–55.

LASTRA, A., LIN, M., AND MANOCHA, D. 2004. ACM workshop on general purpose computation on graphics processors.

LI, W., AND PINGALI, K. 1993. Access normalization: loop restructuring for numa computers. *ACM Transactions on Computer Systems 11*, 4, 353–375.

MCCOOL, M., TOIT, S. D., POPA, T., CHAN, B., AND MOULE, K. 2004. Shader algebra. *ACM Trans. Graph. 23*, 3, 787–795.

OWENS, J., LUEBKE, D., GOVINDARAJU, N., HARRIS, M., KRUGER, J., LEFOHN, A., AND PURCELL, T. 2005. A survey of general-purpose computation on graphics hardware.

PURCELL, T., DONNER, C., CAMMARANO, M., JENSEN, H., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, 41–50.

RUMPF, M., AND STRZODKA, R. 2001. Using graphics cards for quantized FEM computations. In *Proc. of IASTED Visualization, Imaging and Image Processing Conference (VIIP'01)*, 193–202.

SEN, S., CHATTERJEE, S., AND DUMIR, N. 2002. Towards a theory of cache-efficient algorithms. *Journal of the ACM 49*, 828–858.

TOLIMIERI, R., AN, M., AND LU, C. 1997. *Algorithms for Discrete Fourier Transforms and Convolution*. Springer.

VITTER, J. 2001. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 209–271.

WOLFE, M., SHANKLIN, C., AND ORTEGA, L. 1995. *High performance compilers for parallel computing*. Addison-Wesley.

WOLFE, M. 1987. Iteration space tiling for memory hierarchies. *Proc. of the Third SIAM Conference on Parallel Processing for Scientific Computing*, 357–361.