Cache-Efficient Numerical Algorithms using Graphics Hardware

Naga K. Govindaraju [a] Dinesh Manocha [b]

[a] *Microsoft Corporation*

[b] *UNC Chapel Hill*

Abstract

We present cache-efficient algorithms for scientific computations using graphics processing units (GPUs). Our approach is based on mapping the nested loops in the numerical algorithms to the texture mapping hardware and efficiently utilizing GPU caches. This mapping exploits the inherent parallelism, pipelining and high memory bandwidth on GPUs. We further improve the performance of numerical algorithms by accounting for the same relative memory address accesses performed at data elements in nested loops. Based on the similarity of memory accesses performed at the data elements in the input array, we decompose the input arrays into sub-arrays with similar memory access patterns and execute on the sub-arrays for faster execution. Our approach achieves high memory performance on GPUs by tiling the computation and thereby improving the cache-efficiency. Overall, our formulation for GPU-based algorithms extends the current graphics runtime APIs without exposing the underlying hardware complexity to the programmer. This makes it possible to achieve portability and higher performance across different GPUs. We use this approach to improve the performance of GPU-based sorting, fast Fourier transform and dense matrix multiplication algorithms. We also compare our results with prior GPU-based and CPU-based implementations on high-end processors. In practice, we observe 2–10$\times$ improvement in performance.

Key words: Memory model, graphics processors, scientific algorithms.

1. Introduction

The programmable graphics processing units (GPUs) are primarily designed to achieve high rasterization performance for interactive applications. Current GPUs consist of a high number (e.g. $64-128$) of fragment processors with high memory bandwidth. In many ways, the architecture of a current GPU is similar to a *many-core* processor [2], which achieves higher parallel code performance for rasterization applications. This is in contrast with multi-core CPUs, which consist of best single-thread performing cores.

Current GPUs can offer 10$\times$ higher main memory bandwidth and use data parallelism to achieve up to 10$\times$ more floating-point throughput than the CPUs. This computational capability has been widely used to accelerate scientific, geometric, database and imaging applications, and a new area of GPGPU [1] has emerged over the last decade [29]. Most of these non-graphics applications map the the underlying problem and data structures to the rasterization hardware and achieve higher throughput.

Email addresses: nagag@microsoft.com (Naga K. Govindaraju), dm@cs.unc.edu (Dinesh Manocha).

[1] http://gpgpu.org

In this paper we address the problem of efficient implementation of scientific and numerical algorithms on GPUs. The GPUs have been used to design faster solvers for sparse and dense linear systems, matrix multiplication, fluid flow simulation, FFT, sorting and finite-element simulations [29,25]. The main goal in these algorithms is to utilize multiple vertex and fragment processors within a GPU, along with high memory bandwidth. Specifically, current high-end GPUs consist of up to 1GB of texture memory. The texture memory is primarily designed to store and perform fast accesses to 2D arrays represented as *textures* on GPUs. Moreover, GPU architectures are designed to perform vector computations on these 2D arrays and achieve high memory bandwidth using a high (e.g. 256-bit) memory interface to the texture memory. At the same time, each fragment processor has small L1 and L2 SRAM caches much smaller as compared to L1 and L2 CPU caches, respectively, (see Fig. 1). The GPUs perform block transfers between the caches and DRAM-based video memory. In order to achieve a higher performance, we need to design broad schemes to achieve higher cache efficiency.

Current GPUs support 32-bit floating point arithmetic, and GPU-based implementations of some scientific algorithms can outperform optimized CPU-based implementations available as part of ATLAS or the Intel Math Kernel Library (MKL). However, most of these algorithms were designed for specific applications, and it is hard to generalize them to a broad class of problems. Furthermore, their relative performance varies on different GPUs.

Main results: We present a general approach to design cache-efficient algorithms for numerical and scientific computations on GPUs. Our work builds on the memory model presented by Govindaraju et al. [15], and we use that model to improve the cache efficiency of numerical algorithms. Specifically, we abstract the memory addressing in nested loops in numerical algorithms to texture mapping hardware on GPUs. The texture mapping hardware is used to perform *bilinear interpolation* to compute the memory addresses for computation on nested loops. This mapping results in efficient cache utilization as it exploits spatial locality in memory accesses and better pipelining of memory accesses within the computation.

We represent the input arrays as 2D textures. We decompose the input array into multiple sub-arrays with similar memory access patterns to perform scientific computations on the array. The memory accesses to the input arrays are modeled based on a 2D block-array representation [15]. Each block-array represents a cache line transfer between the caches and the texture memory. Based on this hardware model, we apply the 3C's model [19] to analyze the cache misses and compute the memory performance of scientific algorithms on GPUs. In order to reduce the cache misses, we apply tiling algorithms to improve the performance of three numerical algorithms: sorting, fast Fourier transform and dense matrix multiplication. We compare their performance with prior GPU- and CPU-based algorithms. Our algorithms are able to achieve a performance of 65–100 GB/s effective memory performance on a NVIDIA 8800 GPU. Moreover, we have observed 2–10$\times$ performance improvement over optimized CPU-based implementations running on high-end dual 3.6 GHz Xeon processors or dual Opteron 280 processors.

Organization: The rest of the paper is organized as follows. We provide a brief overview of prior work on cache efficient algorithms, scientific libraries and GPU-based algorithms in Section 2. Section 3 presents the background on the GPU memory model and presents a technique for efficient implementation of nested loops on GPUs. We present the implementation of cache-efficient numerical algorithms using the GPU 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 designing efficient numerical algorithms such as sorting, FFT and matrix multiplication algorithms on CPUs and GPUs.

2.1. Sorting Algorithms

Sorting is a well-studied problem in the literature of computer-science. Many optimized sorting algorithms such as quicksort and radixsort decompose the problem into sub-problems and recurse on the sub-problems. The decomposition or partitioning of the input array requires writes to random memory locations and is often memory-intensive [23]. Many parallel algorithms including sorting networks [34,7,12] have also been designed to accelerate sorting on mesh-connected processor arrays. Among these, sorting networks such as bitonic sorting network [32,20,14] and periodic balanced sorting network [16] have deterministic memory

accesses and map well to GPUs. These algorithms implement the sorting network on the GPU using 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 the number of texture operations. We also note that the prior GPU-based sorting network algorithms did not apply loop-blocking optimizations to improve the memory performance.

2.2. FFT Algorithms

Fast Fourier Transform is one of the key routines used in signal processing, multiplying large integers, etc. At a high-level, the FFT algorithms use a divide-and-conquer strategy to quickly evaluate the discrete Fourier transform (DFT) and its inverse. The divide-and-conquer strategy on FFTs defines a butterfly network. The butterfly network consists of multiple stages and within each stage, the network defines a deterministic mapping between some elements in the input array elements to output locations. Many FFT algorithms have also been optimized for input data by exploiting symmetry and for memory requirements using in-place algorithms [35,36,39]. Researchers have also designed optimized algorithms on supercomputers including external memory algorithms [3,5,4]. Recently, algorithms exploiting the programmability and parallelism in GPUs have been proposed [28,30]. These algorithms efficiently map the butterfly network in Cooley-Tukey algorithm to the programmable pipeline in GPUs. These GPU-based implementations have not used loop-blocking optimizations to improve their overall performance.

2.3. Dense Matrix Multiplication

Dense matrix multiplication is a classic routine used to understand the efficiency of caches in CPUs [31]. Recent work has focused on analyzing the performance of GPU-based matrix-matrix multiplication algorithms [24,18,13]. Hall et al. [18] 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. [13] have shown that matrix-matrix multiplication can be inefficient on prior GPUs due to low cache bandwidth limitations.

2.4. 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 [1] and ATLAS [2] for linear algebra software, FFTW [3] 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* [37] and *linear loop transformations* [6,26]. 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* [38]. Carr and Kennedy [9] 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 [21]. Many memory models have also been proposed to estimate program performance for nested loops [22,10].

3. Background

In this section we give a brief overview of GPU architectures. We present the GPU memory model presented by Govindaraju et al. [15] to analyze the performance of GPU-based algorithms using the graphics APIs such as Microsoft DirectX and highlight some of the differences with CPU-based memory models and optimization techniques.

3.1. Graphics and Scientific Programming

GPUs are mainly designed for rapidly transforming 3D geometric primitives into pixels on the screen. GPUs can also be regarded as massively parallel vector processors suitable for general data-parallel computations—therefore, they can be used to significantly accelerate many numerical algorithms. Apart from the massive parallelism, GPUs also differ in their memory hierarchy from CPUs. Fig. 1 provides a conceptual abstraction of CPU and GPU memory hierarchies and high-

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

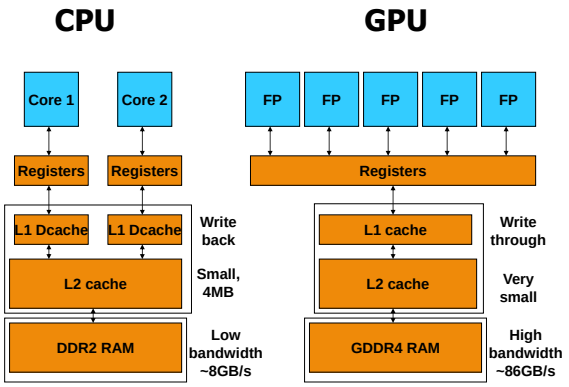

Fig. 1. A high-level abstraction for CPU and GPU memory hierarchies. On the right we show a conceptual abstraction for the GPU memory hierarchy. GPUs write to a high-bandwidth, high-latency video memory using small, write-through caches. Moreover, large number of fragment processors (FPs) on the GPU share the caches. These differences from CPU memory hierarchies indicate different memory optimizations for achieving higher performance on GPU-based scientific algorithms.

lights the main differences. At a high-level, GPUs consist of small, write-through caches and use a dedicated, high-bandwidth, high-latency memory subsystem. Due to higher latencies, efficient cache-usage can significantly improve the performance of GPU-based scientific algorithms. In addition we extend the graphics-based programming APIs to achieve scalable and portable performance for the numerical routines among different architectures, without exposing the underlying hardware complexity.

In this section we briefly describe the graphics data representations that can be used by scientific algorithms and the underlying mechanisms used to access the data and to perform the computations on GPUs using graphics programming APIs:

– ***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 can represent the input data in 2D textures and perform streaming computations on the data elements in the 2D textures.

– ***Inter-processor Communication:*** Current graphics APIs allow interprocessor communication by allowing the processors to write to textures in a global memory. The written textures can then be streamed into fragment processors for further processing. In order to avoid cache coherence issues, the input textures need to be different from the output textures. These restrictions result in simpler hardware designs and avoid many of the unnecessary coherence overheads while rendering geometry onto the screen.

– ***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. Memory Model for Graphics-Based Scientific Programming

Current GPUs achieve high memory bandwidth using a 4-6x wider memory interface to the video memory than the memory interface between CPUs and RAM. 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.

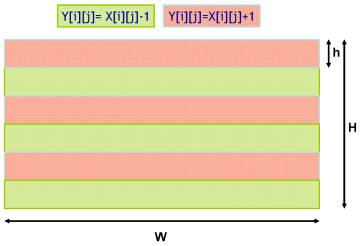

Fig. 2. This figure shows a color-coding of the regions corresponding to increment and decrement operations in Y while executing the nested loops in routine 3.1. The orange colored regions indicate increment operations, and green-colored regions represent decrement operations.

These prefetch sequential block transfers utilize the memory bus efficiently. Moreover, current GPUs also use Graphics Double Data Rate (GDDR) versions for the video memory to achieve high bandwidth per pin [4] . 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 [17]. 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 L2 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 L2 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 can execute higher number of threads in parallel using the programmable processors. Moreover, the clock speeds of the programmable processors are also lower than the CPU clock speeds. Therefore, the GPUs are able to better hide the memory 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 [19] into our memory model. In the 3C's model, cache misses are classified as:

(i) Compulsory or cold misses which are caused due to the first reference of a block that is not in cache.

(ii) Capacity misses which occur due to the limited cache sizes.

(iii) 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 [38,9]. 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.

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. 2 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 in which the elements in X are decremented. Suppose the width of the array is W and the 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.

[4] http://en.wikipedia.org/wiki/GDDR4

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.*

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 a 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 parameters for nested loops such as Algorithm 3.1 in scientific computations.

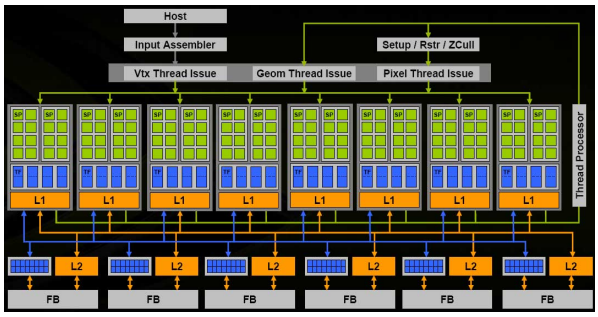

Fig. 3. High-level architecture diagram illustrating the memory hierarchy in the NVIDIA 8800 GTX GPU

4. Numerical Algorithms

In this section we present the design of efficient numerical algorithms using the graphics APIs. Specifically, we present algorithms for three applications—sorting, real and complex Fast Fourier transforms, and matrix multiplication. We also present the application of our memory model to identify the GPU block sizes and cache sizes for measuring the memory efficiency of these scientific algorithms in terms of the cache misses. In the experiments presented in this section, we use our memory model and tile the computations within the loops in our algorithms to minimize cache misses. We then profile the performance of our algorithms by varying the tile size in powers-of-two up to a user-defined maximum tile size and empirically compute the tile size that results in the best execution time.

4.1. Sorting

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 computational and memory throughput on GPUs. GPUs cannot write to arbitrary memory locations using the current graphics APIs—therefore, optimized sorting algorithms such as quicksort do not map well to GPUs. Sorting networks have deterministic access patterns and do not require writes to random locations. In this section we analyze the problem of bitonic sorting networks [14,32,20].

At a high level, GPU-based sorting algorithms read values from an input array or texture, perform data-independent comparisons using a fragment program, and write the output to another array. The output array is then swapped with the input array, and the comparisons are iteratively performed until the whole array is sorted. Due to the deterministic accesses, sorting net-

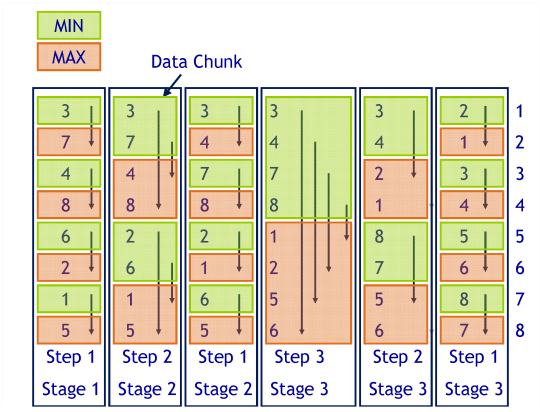

Fig. 4. This figure illustrates a bitonic sorting network on 8 data values. The sorting algorithm proceeds in 3 stages. The output of each stage is the input to the next stage. In each stage the array is conceptually divided into sorted data chunks or regions highlighted in green and red. Elements of adjacent chunks are merged as indicated by arrows. The minimum element is moved to the green region, and the maximum is stored in the red colored regions, producing larger sorted chunk.

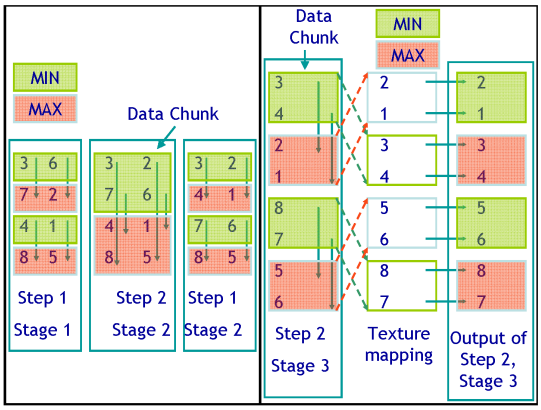

Fig. 5. The left figure shows the 1-D mapping of comparisons among array elements in step 2 and stage 3 of Figure 4. The mapping is implemented using GPU texturing hardware. For each data chunk, we pass the element indices (or vertex locations of a 1-D line) of the corresponding data chunk for comparisons. The texturing hardware fetches the data values at the corresponding locations for each pixel, and a single-instruction fragment program computes the minimum or maximum in parallel on multiple pixels simultaneously using the fragment processors. The right figure shows the 2D-representation of the 1-D array of size 8 shown in Figure 4. In this example the width of the 2D array is 2 and the height is 4. Observe that the data chunks now correspond to row-aligned quads, and the sorting network maps well to the GPU 2D texturing hardware.

work algorithms map well to GPUs. A few examples of these sorting network algorithms include periodic balanced sorting network [12], bitonic sorting network [7], etc. Among these, the bitonic sorting network has lower memory bandwidth requirements and maps well to GPUs.

Bitonic sorting network performs data-independent comparisons on bitonic sequences [7]. Given a sequence $a = (a_0, a_1, \ldots, a_n)$, the bitonic sorting algorithm proceeds bottom-up, merging bitonic sequences of equal length at each stage. It first constructs bitonic sequences of size 2 by merging pairs of adjacent data elements (a_{2i}, a_{2i+1}) where $i = 0, 1, \ldots, \frac{n}{2} - 1$. Then bitonic sequences of size 4 are formed in stage 2 by merging pairs of bitonic sequences (a_{2i}, a_{2i+1}) and $(a_{2i+2}, a_{2i+3}), i = 0, 1, \ldots, \frac{n}{2} - 2$. The output of each stage is the input to the next stage. The size of the bitonic sequence pairs doubles at every stage. The final stage forms a sorted sequence by merging bitonic sequences $(a_0, a_1, ., a_{\frac{n}{2}}), (a_{\frac{n}{2}+1}, a_{\frac{n}{2}+2}, \ldots, a_n)$ (see Figure 4).

Specifically, stage k is used to merge two bitonic sequences, each of size 2^{k-1}, and it generates a new bitonic sequence of length 2^k. The overall algorithm requires $logn$ stages. In stage k we perform 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 chunks 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 in the other data chunk. Figure 4 shows a bitonic sorting network on 8 data values. Each data chunk in a step is color-coded, and elements in adjacent data chunks are compared. The minimum is stored in the green colored region, and the maximum is stored in the red colored region. For further details on the bitonic sorting algorithm, refer to [11].

In a GPU each bitonic sort step corresponds to mapping values from one chunk in the input texture to another chunk in the input texture using the GPU's texture mapping hardware, as shown in Figure 5. The texture mapping hardware fetches data values at a fixed distance from the current pixel and compares against the current pixel value, and may replace the value based on the comparison. The texturing hardware works as follows: first, a 2D array is specified to fetch the data values. Then a 2D quadrilateral is specified with lookup coordinates for vertices. For every pixel in the 2D quadrilateral, the texturing hardware performs a bilinear interpolation of the lookup co-ordinates of the vertices.

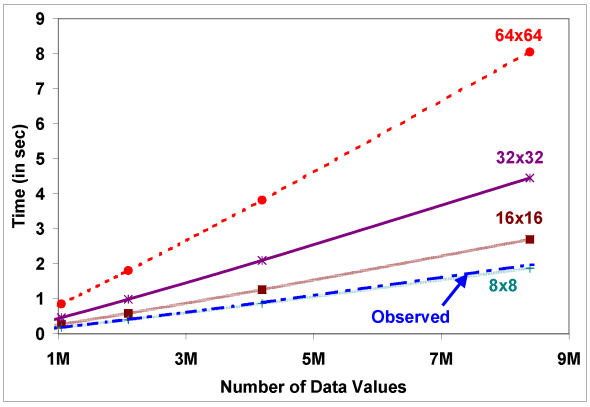

Fig. 6. Our memory model for a NVIDIA 7800 GTX GPU predicts the block size for efficient sorting on GPUs. The analysis closely matches the experimental results for a cache block size of 8×8.

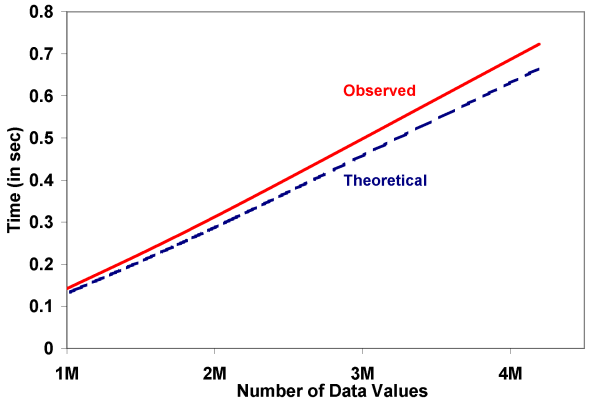

Fig. 7. The computational time of our cache-efficient bitonic sort algorithm as a function of the number of data values on a 7800 GTX GPU. We observe that the experimental results closely match the theoretical results for a 64×64 block size. The graph indicates that our algorithm achieves 37 GB/s memory bandwidth, close to the 40 GB/s maximum.

The interpolated coordinate is used to perform a 2D array lookup by the fragment processor. This results in the larger and smaller values being written to the higher and lower target pixels. The left Figure 5 illustrates the use of texture mapping for sorting. In this example, 1-D lines are specified for data chunks with appropriate lookup co-ordinates. For example, the first line segment $(0,1)$ is specified with the vertex lookup co-ordinates $(2,3)$. Then the texture mapping hardware is used to directly fetch values a_2, a_3 and compare them against a_0 and a_1 respectively, within the fragment processor.

As GPUs are primarily optimized for 2D arrays, we map the 1D array onto a 2D array as shown in Figure 5. The resulting data chunks are 2D data chunks that are either row-aligned (as shown in the right side of Figure 5) or column-aligned. The resulting algorithm maps well to GPUs.

The overall sorting algorithm requires a large number of $O(n \log^2 n)$ compute and memory references. 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. 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 6 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 6 are 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.

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 7 highlights the observed and theoretical performance 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 8 illustrates the algorithm's performance as a function of the decomposition block or tile size. In our implementation, we vary the tile size and profile the execution time. We achieved high performance for our implementation at a tile size of 64×64. This tile size corresponds to 64KB of data since each element consists of four components and each component is represented using 4 bytes. These four components store two key-pointer pairs per element. Based on our memory model and the bitonic sort algorithm, for computation at each key-pointer location, we effectively read a key-pointer from and write an output key-pointer to the texture. Based on these empirical results, we suggest that the cache size on a 7800 GTX GPU can be close to 128 KB.

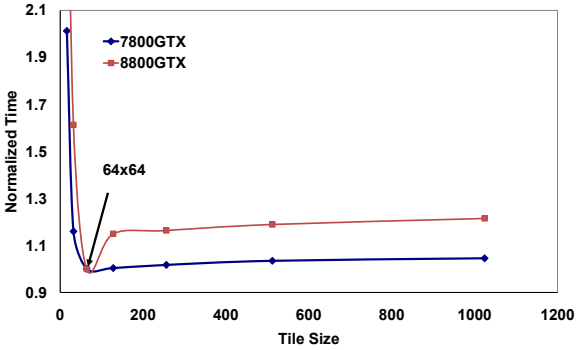

Fig. 8. Normalized performance of our cache-efficient sorting algorithm as a function of tile size sorting 8M floating point key-pointers using a 7800 GTX GPU and 8800 GTX GPU. 64×64 tiles had the best performance. As the tile size decreases, the vertex overhead dominates the memory bandwidth savings. As the tile size increases beyond 64×64, memory performance degrades due to cache misses.

4.2. Fast Fourier Transforms

Fast Fourier transform 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 complex FFTs on GPUs. We use the 1-D power-of-two FFT algorithms to compute real and higher dimensional FFT algorithms.

FFT algorithms use a butterfly network to map input elements in multiple stages, and perform complex multiplications and additions on the input elements. Since the output array needs to be distinct from input array for GPU computations, we use out-of-place radix-2 FFT algorithms. Moreover, in order to avoid the expensive bit-reversal operations in the standard Cooley-Tukey algorithm [35], we use a standard Stockham formulation of the FFT.

The Stockham FFT algorithm has been designed to work well on SIMD architectures such as GPUs. At a high level, the FFT algorithm reads the complex values from an input array. We represent the real-components and the imaginary components of the complex values using two separate arrays. We store the output results using two more arrays. The output results are swapped with input arrays, and the FFT algorithm is performed iteratively, until the FFT is performed on the entire data.

The Stockham FFT algorithm is very similar to the bitonic sort algorithm. It builds the FFT bottom-up, performing FFTs on sequences of equal length and outputting FFTs of sequences twice the length at each stage. Given a sequence with complex values $a = (a_0, a_1, \ldots, a_n)$, the algorithm proceeds in $\log(n)$ stages. Within each stage an element in the input sequence is mapped to another element at a fixed distance, and a transformation is performed between the two elements and stored in the element's location. The transformation is independent of transformations on the remaining elements in the input sequence and exploits the high data parallelism in GPUs.

The pseudo-code for Stockham FFT is shown in Algorithm 4.1. The algorithm proceeds in steps 1 to $\log n$. Specifically, in step t it performs transformations on two input sub-arrays of size 2^{t-1} and generates a new output sub-array of size 2^t. The elements read from the input

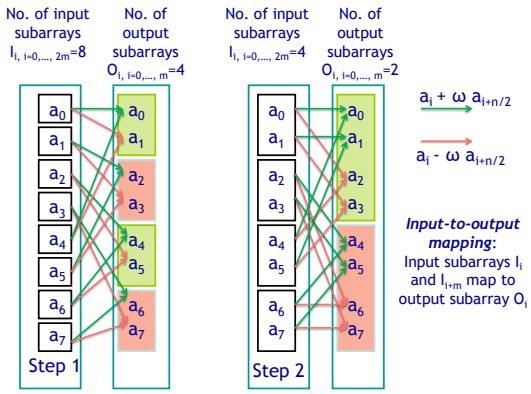

Fig. 9. This figure illustrates the first two stages of a Stockham butterfly network to compute FFTs on 8 data values. The overall FFT algorithm proceeds in 3 stages. The output of each stage is the input to the next stage. In each stage the output array is conceptually divided into data chunks or sub-arrays based on similar computation. The input sub-arrays map onto the output sub-arrays as shown. A green arrow indicates that the computation is a positive addition between the transformed inputs obtained after multiplication with their corresponding twiddle factors. Similarly, a red arrow indicates a subtraction between the transformed inputs.

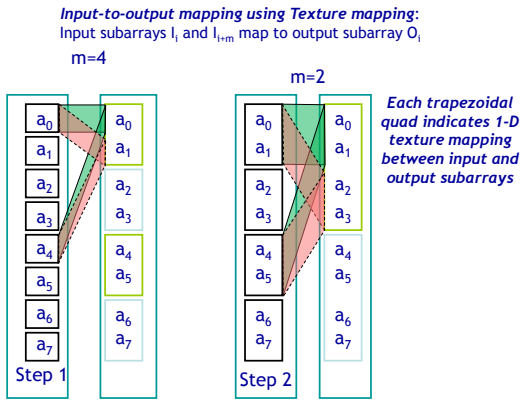

Fig. 10. This figure shows the mapping between the input arrays and output sub-arrays in Stockham FFT network on 8 input values shown in Fig. 9. The figure highlights two steps. Within each step, the input arrays are texture mapped onto the output sub--arrays using trapezoidal quadrilaterals as shown. Each textured quad bilinearly interpolates the addresses from the input, providing a 1-1 mapping between input and output locations. Since each output is obtained by transforming two input values, two textured quads are drawn from the input sub-arrays to the output sub-arrays. The green regions indicate an addition between the corresponding transformed input values and red regions indicate a subtraction between the corresponding transformed input values.

sub-arrays are separated by $\frac{n}{2}$ elements in the original input array. This computation repeats for every 2^t elements in the output array. Based on this observation, we can decompose the output array in each step t into equi-sized sub-arrays of size 2^t. The decomposition results in $m = \frac{n}{2^t}$ sub-arrays in the output. Similarly, based on the computational mapping in step t, we also decompose the input array into sub-arrays of equal size 2^{t-1}, and there are $\frac{n}{2^{t-1}} = 2m$ sub-arrays in the input sequence. We label the input sub-arrays and the output sub-arrays sequentially in the ranges $[0,2m)$ and $[0,m)$, respectively.

We will now describe the computational mapping of FFTs on GPUs in more detail. From Algorithm 4.1 we can observe that for each output sub-array $j, 0 \leq j < m$, two input sub-arrays labeled j and $j+m$ map onto it. Each output sub-array can further be decomposed into two sub-arrays of equal size. In the first half of the output sub-array j, the transformation includes an addition between the input sub-array j and the array obtained by multiplying each element of input sub-array $j+m$ with a corresponding twiddle factor (see Fig. 9). In the second half of the output sub-array j, the transformation includes a subtraction between the input sub-array j and the array obtained by multiplying each element of input sub-array $j+m$ with the same twiddle factor. This mapping is similar to the decomposition of input array into regions corresponding to minimum and maximum operations in the bitonic sort algorithm. The transformation in each sub-array includes multiply-and-add (MAD) instructions. GPUs support fused MAD instructions and consist of multiple units to concurrently execute many MAD instructions. Moreover, the computational mapping between the input sub-arrays to the output sub-array can be performed efficiently using the texture mapping hardware on GPUs (see Fig. 10). Therefore, FFT algorithms map well to GPUs.

Since GPUs are optimized for 2D representations, we map the 1D 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 1D power-of-two array, we compute a 2D array representation with a power-of-two width W and height H, respectively. The memory access pattern for FFTs is dependent on the size of the sub-array in a step t. It can be described as follows:

– $2^{t-1} < W$: If the input sub-array size in a step t is less than W, then the output array can be decomposed into column-aligned quads of width 2^t and height H based on similar access patterns. The input array *in*

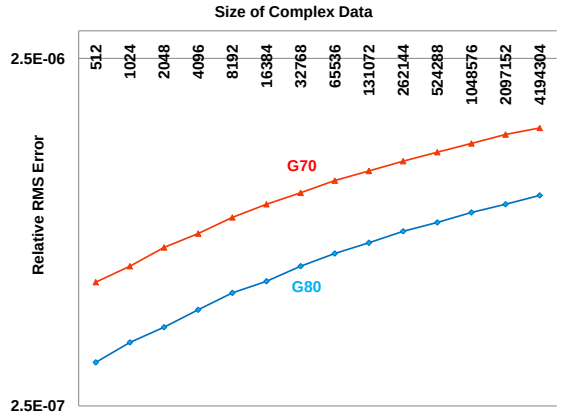

Fig. 11. This figure highlights the relative root mean square (RMS) error obtained by performing FFT computations on complex data sets. The graph indicates that our FFT algorithm is able to obtain a RMS error less than 10^{-6} on 4M complex values and is comparable to the accuracy of CPU-based FFT algorithms.

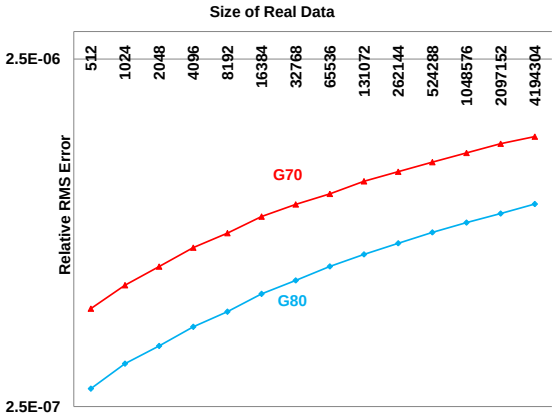

Fig. 12. This figure highlights the relative root mean square error obtained by performing FFT computations on real data sets. The graph indicates that our FFT algorithm is able to obtain a RMS error less than 10^{-6}. The graph indicates that our real FFT algorithm achieves higher precision on new GPU hardware such as NVIDIA G80.

Nested Loop Stockham FFT

```
1  for(t = 1; t ≤ log n; t = t + 1)
2      for(j = 0; j < n/2^t; j = j + 1)
3          for(k = 0; k < 2^(t-1); k = k + 1)
4              angle = −2πk/2^t
5              out[j2^t + k] = in[j2^(t-1) + k] + e^(i angle) in[ j2^(t-1) + k + n/2]
6              out[j2^t + k + 2^(t-1)] = in[j2^(t-1) + k] - e^(i angle) in[ j2^(t-1) + k + 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.

is divided into column-aligned quads of width 2^{t-1} and height $\frac{H}{2}$. Computation is performed by mapping the input quads onto the output quads and executing a fragment program on the quads. In this case each column-aligned quad in *out* maps to four column-aligned quads in the input array *in*. Although four quads are mapped, only two input quads are selected based on whether the row number of the output is even or odd.

– $2^{t-1} \geq W$: If the sub-array size of the *in* is greater than or equal to W, the output array *out* is decomposed into row-aligned quads of width W and height $\frac{2^t}{W}$. The input array is decomposed into quads of half the width and same height as the corresponding output quads. In this case each row-aligned quad in the output maps to two row-aligned quads in input array.

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 a greater benefit by performing cache analysis on FFTs.

Our algorithm has also been extended to perform two key algorithms:

– **Real FFT:** A naive implementation of real FFTs can be performed by treating the imaginary component of the complex FFTs as zeros. We improve the performance of real FFTs using the symmetry in computation of FFTs for real values and eliminate redundant computations. This improves the real FFT performance by nearly a factor of two. Given an input sequence, we compute a complex sequence of half-the-length by packing the even and odd elements of the input sequence. A complex FFT is then applied on the complex sequence using Algorithm 4.1, and the data is then readback to the CPU. We then perform a linear-time post-processing operation to compute the real FFT of the sequence. As the complex FFT requires half the number of operations as the naive implementation, the resulting algorithm is nearly 2x

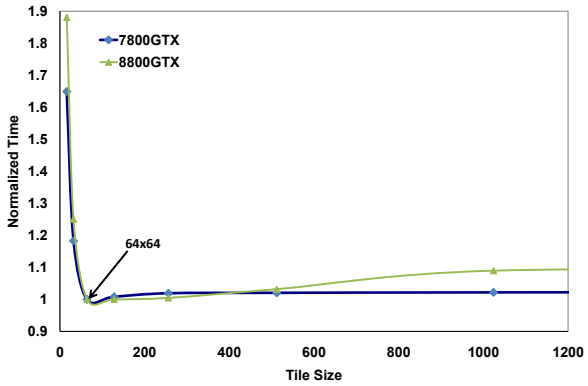

Fig. 13. Normalized performance of our cache-efficient FFT algorithm as a function of tile size on 4M complex floating point values using a 7800 GTX and 8800 GTX GPU. We obtained the best performance using $T \times T = 64 \times 64$ tiles.

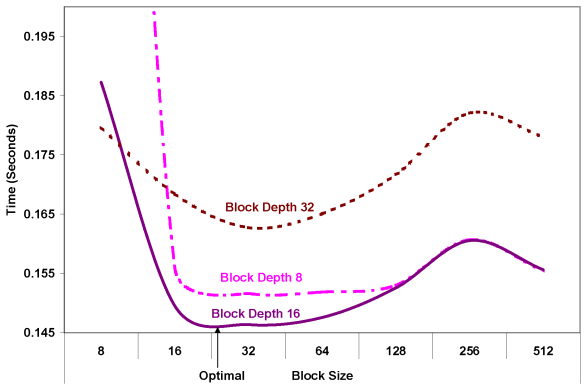

Fig. 14. The performance of our cache-efficient matrix multiplication algorithm as a function of tile size for multiplying two 2K $\times$ 2K floating point matrices using a 7800 GTX GPU. We obtained the best performance using $T \times T = 64 \times 64$ tiles and a depth of 16. As the block depth decreases, 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.

faster than the naive implementation. Moreover, the optimized implementation also reduces the memory requirement of the naive algorithm by a factor of 2.

– **Many FFTs:** In order to compute many FFTs of the same length L, we pack the input arrays into a single 2D texture and perform algorithm 4.1. However, we only perform the first $\log L$ steps. The resulting output array consists of the FFTs of the input arrays. The output FFTs are stored into the output texture in the same order as the corresponding input arrays in the 2D texture.

4.2.1. *Accuracy and Memory Requirements*

Our goal is to design fast FFT algorithms without trading off the precision. The twiddle factors used for multiplication in Algorithm 4.1 can be precomputed and stored in a texture of the same size. This doubles the memory requirements of the algorithm. In addition, it also increases the memory bandwidth requirements of the algorithm by 33%. We reduce the memory bandwidth by computing the twiddle factors on GPUs and avoid storing precomputed tables. Our choice of computing the twiddle factors also follows the trend that computational power is cheaper than memory bandwidth. We further improve the precision in computation by using interpolation capabilities of texturing hardware to compute the angles used in twiddle factors across the quad. The linear interpolation of angles across the output quads eliminates unnecessary computations inside the fragment program and pipelines the angle computations using the texturing units of GPUs. The interpolation improves the precision by eliminating redundant division operations. Figs. 11 and 12 highlight the accuracy of our algorithm on random complex values in the range [-0.5 0.5]. The graph indicates that the precision obtained by our algorithm is comparable to the optimized CPU-based FFT algorithms [5].

4.2.2. *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 *out* 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. 13 highlights the performance of the FFT algorithm as a function of the tile size.

The FFT algorithm is also more compute-intensive than our 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}

[5] CPU results available at http://www.fftw.org

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 [24] 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. [18] 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 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. 14 highlights the performance of matrix-multiplication on GPUs as a function of T and D using a matrix of size 2K $\times$ 2K.

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 × 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 × 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 × 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.*

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 implemented our algorithms using the OpenGL graphics API on a Windows XP PC with a 3.4GHz Pentium IV CPU and 2GB RAM. We have tested the performance of our applications on four high-end GPUs – NVIDIA 6800 Ultra, NVIDIA 7800 GTX, NVIDIA 7900 GTX and NVIDIA 8800 GTX GPU released in successive generations (see Fig. 15). Table 1 summarizes the memory architecture of the three GPUs. Our algorithms achieved high performance for a tile-size of 64×64 on the GPUs irrespective of the data size. In our experiments, we measured the time taken by the GPU to perform the computation assuming the data is present in the GPU memory. For sorting 8M key-pointer pairs, we obtain an average of 42.4 giga-operations per second on a NVIDIA 8800 GPU. The observed bandwidth in the sorting benchmark is nearly 100 GB/s on a 8800 GTX GPU, close to the peak memory throughput of the 8800 GPU.

In the FFT benchmark, we have measured the GFLOPS obtained using our algorithm using the standard FFTW metric [6]. The benchmark indicates that the floating point operations on a complex array with n values is

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

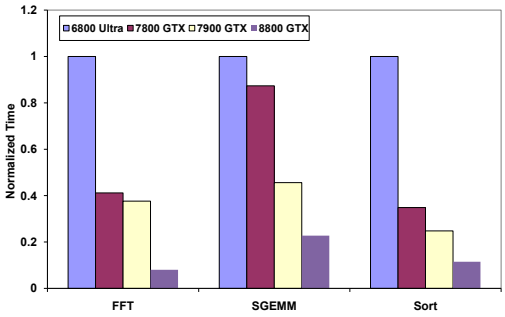

Fig. 15. Normalized Performance of our cache-efficient applications on four successive generation GPUs—NVIDIA 8800, NVIDIA 7900, 7800 and 6800 GPUs. Our sorting, FFT and matrix multiplication algorithms are able to achieve 100, 65 and 80 GB/s memory performance respectively on a single NVIDIA 8800 GTX GPU.

GPU	Peak Memory Bandwidth (GB/s)	Memory Interface (bits)	Transistor Count (Millions)
6800 Ultra	35.2	256	222
7800 GTX	38.4	256	302
7900 GTX	51.2	256	278
8800 GTX	86	384	681

Table 1
This table highlights the peak memory bandwidth, the memory interface width and the transistor count for NVIDIA 6800, 7800, 7900, and 8800 GPUs

computed as $5n\log n$. On a 4 million single precision 1-D power-of-two complex FFT benchmark, we are able to obtain 16 GFLOPS of performance on a NVIDIA 8800 GPU. We are able to obtain an effective memory bandwidth of 65 GB/s on a NVIDIA 8800 GTX GPU for performing complex FFTs.

Our matrix multiplication algorithm achieves 35 GFLOPS on a NVIDIA 8800 GTX GPU. Our cache-efficient matrix multiplication algorithm is able to achieve nearly 80 GB/s memory performance on a single NVIDIA 8800 GTX GPU.

Using NVPerfKit [7], 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. The near-peak memory bandwidth obtained by our algorithm indicates an efficient mapping between our algorithms and GPUs. Fig. 15 indicates the scalable performance and portability of our algorithms on different generation GPUs.

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

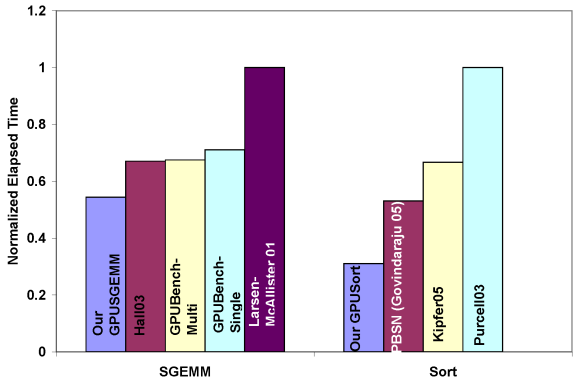

Fig. 16. Normalized elapsed time of our cache-efficient applications against prior GPU-based algorithms on a NVIDIA 7900 GPU. Our cache-efficient algorithms are able to achieve 2–3× performance improvement over prior GPU-based scientific algorithms.

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. 16 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[30]. Libgpufft is implemented using the BrookGPU compiler and used the Cooley-Tukey FFT radix-2 FFT algorithm. Our algorithm does not require bit-reversal-based data rearrangements as explained in Section 4.2 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 [8] 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, on another SMP machine with two dual-core Opteron 280 processors and on 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

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

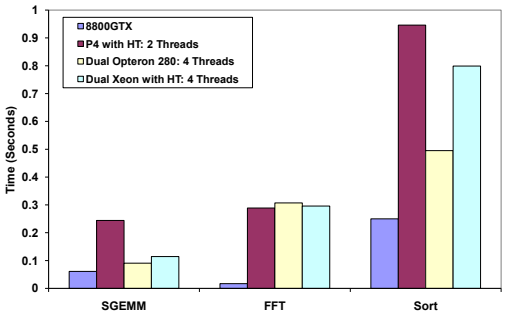

Fig. 17. Performance of our cache-efficient applications on a NVIDIA 8800 GPU against optimized scientific algorithms on high--end SMP machines with dual Xeon or two dual-core Opteron processors.

results highlighted in Fig. 17 indicate that our GPU matrix multiplication algorithm on a single 8800 GTX GPU performs 1.5x times better than the dual Xeon and Opteron processors. Our sorting algorithm is able to achieve 2–4$\times$ performance improvement over MKL implementation on high-end Intel processors and AMD Opteron 280 processor. Our FFT algorithm is able to achieve 10$\times$ performance improvement over Xeon or Opteron processors.

6. Conclusions and Future Work

We presented the design of fast cache-efficient scientific algorithms on GPUs. Our algorithms map nested loops in scientific algorithms to the texture mapping hardware, and exploit the data parallelism and high memory bandwidth in GPUs. We analyze the performance of our algorithms using a memory model on GPUs and further improve their performance using cache-efficient tiling strategies. We present three scientific applications – sorting, fast Fourier transforms and dense matrix multiplication, and compared their performance against prior optimized CPU-based and GPU-based algorithms. Our results indicate a significant performance improvement using a single NVIDIA 8800 GPU.

There are several avenues for future work. We would like to incorporate the caching strategies into new programming APIs from graphics vendors such as AMD CTM [9] and NVIDIA CUDA [10]. An interesting avenue for research is incorporating caching strategies into runtime systems in GPGPU compilers and libraries such as Microsoft accelerator [33], BrookGPU [8] and Sh [27]. We are interested in applying the memory model to other scientific applications and streaming architectures such as the IBM Cell processor.

[9] http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf

[10] http://developer.nvidia.com/cuda

Acknowledgements

We dedicate our work to Jim Gray of Microsoft Corporation. This work would not have been possible without his support and guidance. 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, David Tuft, Mike Marr, and David Blythe of Microsoft Corporation, Mike Houston of Stanford university, and 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 thank Whitney Vaughan and other members of UNC GAMMA group for useful suggestions and support.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, and D. Sorensen. *LAPACK User's Guide, Release 1.0*. SIAM, Philadelphia, 1992.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 18 2006.

[3] D. H. Bailey. A high-performance fast Fourier transform algorithm for the Cray-2. *The Journal of Supercomputing*, 1(1):43–60, 1987.

[4] D. H. Bailey. FFTs in external or hierarchical memory. In *Proceedings of the Supercomputing 89*, pages 234–242, New York, NY, 1989. ACM Press.

[5] David H. Bailey. A high-performance FFT algorithm for vector supercomputers. *International Journal of Supercomputer Applications*, 2(1):82–87, 1988.

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

[7] K.E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computer Conference*, 1968.

[8] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs:

stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[9] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. *Proc. of ACM/IEEE Conference on Supercomputing*, pages 114–124, 1992.

[10] S. Coleman and K. McKinley. Tile size selection using cache organization and data layout. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 279–290, 1995.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.

[12] M. Dowd, Y. Perl, L. Rudolpg, and M. Saks. The periodic balanced sorting network. *Journal of the ACM*, pages 738–757, 1989.

[13] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–138. Eurographics Association, 2004.

[14] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High performance graphics coprocessor sorting for large database management. *Proc. of ACM SIGMOD*, 2006.

[15] N. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. *Proc. of ACM SuperComputing*, 2006.

[16] N. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. *Proc. of ACM SIGMOD*, 2005.

[17] Z. Hakura and A. Gupta. The design and analysis of a cache architecture for texture mapping. *Proc. of 24th International Symposium on Computer Architecture*, pages 108–120, 1997.

[18] J. D. Hall, N.A. Carr, and J.C. Hart. Cache and bandwidth aware matrix multiplication on the GPU. Technical Report UIUCDCS-R-2003-2328, University of Illinois at Urbana-Champaign, 2003.

[19] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 38(12):1612–1630, 1989.

[20] P. Kipfer, M. Segal, and R. Westermann. Uberflow: A gpu-based particle engine. *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2004.

[21] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. *Proc. of ACM SIGPLAN*, pages 346–357, 1997.

[22] M. Lam, E. Rothberg, and M. Wolf. The performance and optimization of blocked algorithms. *Proc. of 4th International conference on Architectural support for programming languages and operating systems*, pages 63–74, 1991.

[23] A. LaMarca and R. Ladner. The influence of caches on the performance of sorting. *Proc. of SODA*, pages 370–379, 1997.

[24] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 55–55. ACM Press, 2001.

[25] A. Lastra, M. Lin, and D. Manocha. ACM workshop on general purpose computation on graphics processors. 2004.

[26] W. Li and K. Pingali. Access normalization: loop restructuring for NUMA computers. *ACM Transactions on Computer Systems*, 11(4):353–375, 1993.

[27] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.

[28] Kenneth Moreland and Edward Angel. The fft on a gpu. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[29] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, September 2005.

[30] John D. Owens, Shubhabrata Sengupta, and Daniel Horn. Assessment of graphic processing units (gpus) for department of defense (dod) digital signal processing (dsp) applications. Technical Report ECE-CE-2005-3, Department of Electrical and Computer Engineering, University of California, Davis, October 2005.

[31] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[32] T. Purcell, C. Donner, M. Cammarano, H. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. *ACM SIGGRAPH/Eurographics Conference on Graphics Hardware*, pages 41–50, 2003.

[33] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, New York, NY, USA, 2006. ACM Press.

[34] C. D. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer. *Commun. ACM*, 20(4):263–271, 1977.

[35] R. Tolimieri, M. An, and C. Lu. *Algorithms for Discrete Fourier Transforms and Convolution*. Springer, 1997.

[36] Shmuel Winograd. On computing the discrete Fourier transform. *Mathematics of Computation*, 32:175–199, 1978. URL: `http://cr.yp.to/bib/entries.html#1978/winograd`.

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

[38] M. Wolfe, C. Shanklin, and L. Ortega. *High performance compilers for parallel computing*. Addison-Wesley, 1995.

[39] R. Yavne. An economical method for calculating the discrete Fourier transform. In *Proc. AFIPS Fall Joint Computer Conference*, pages 115–125, 1968.