Efficient computation of sum-products on GPUs through software-managed cache [*]

Mark Silberstein [†] [‡]
Technion—Israel Institute of Technology

Assaf Schuster
Technion—Israel Institute of Technology

Dan Geiger
Technion—Israel Institute of Technology

Anjul Patney
UC Davis

John D. Owens
UC Davis

ABSTRACT

We present a technique for designing memory-bound algorithms with high data reuse on Graphics Processing Units (GPUs) equipped with close-to-ALU software-managed memory. The approach is based on the efficient use of this memory by implementing an application-specific cache in software. We also present an analytical model for performance analysis of such algorithms.

We apply this technique to the design and implementation of the GPU-based solver of the sum-product or *marginalize a product of functions (MPF)* problem, which arises in a wide variety of real-life applications in artificial intelligence, statistics, image processing, and digital communications. Our motivation to accelerate MPF originated in the context of the analysis of genetic diseases, which in some cases requires years to complete on modern CPUs. Computing MPF is similar to computing the chain matrix product of multi-dimensional matrices, but is more difficult due to a complex data-dependent access pattern, high data reuse, and a low compute-to-memory access ratio.

Our GPU-based MPF solver achieves orders-of-magnitude speedups (up to 2700-fold on random data and 270-fold on real-life genetic analysis datasets) on GeForce 8800GTX GPU from NVIDIA over the optimized CPU version on an Intel 2.4 GHz Core 2 with a 4 MB L2 cache.

1. INTRODUCTION

Graphics Processing Units (GPUs) have emerged as a powerful platform for high-performance computation. They have been successfully used to accelerate many scientific workloads [13]. Typically, the computationally intensive parts of the application are off-loaded to the GPU, which serves as the CPU's parallel coprocessor.

Originally, GPUs were designed as a massively parallel machines for concurrent execution of thousands of independent threads, each executing the same code on different data. Such an architecture is optimized for high-throughput stream processing. It allows for high speedups on graphics-like workloads, which can be parallelized into thousands of independent identical subtasks and is characterized by low data reuse (or high reuse of a small working set) and a high compute-to-memory access ratio (*arithmetic intensity*). However, early GPUs achieved low or no performance gains on memory-bound workloads such as a matrix product, which is characterized by high data reuse and low arithmetic intensity [6]. For such workloads, the GPU cacheless memory system prevented efficient utilization of GPU computing hardware, whereas CPU utilization was amplified through the optimal use of the data cache. In fact, GPUs allow for cached memory accesses via the GPU's texture cache. However, this cache is optimized for read-only workloads with 2D spatial locality and a small working set. Thus, even sophisticated use of this cache yielded only modest speedups compared to the multithreaded cache-optimized CPU implementation [8].

The breakthrough in allowing workloads with high data reuse has been the recent introduction of a fast close-to-ALU memory. However, the memory architecture differs between the vendors. While AMD hardware includes regular L1 and L2 caches [15], NVIDIA CUDA [12] provides a special user-managed space called *shared memory*. Shared memory lacks hardware support for cache functionality and is fully managed by the application. It is divided into chunks (16KB each), each shared only among the threads of a *thread block* (up to 512 threads).

Shared memory is intended as a scratchpad for frequently used data [12]. Explicit management makes it especially useful for memory-intensive applications with complex access patterns on the one hand, but greatly complicates the application development on the ther.

We propose a general technique for designing algorithms on GPUs with explicit memory management. The idea is to decouple the data management from the computational structure.

[*] This research was supported by the SciDAC Institute for Ultra-Scale Visualization.

[†] Corresponding author: marks@cs.technion.ac.il

[‡] This research was conducted at UC Davis.

The first stage is to design a serial computation algorithm with spatial and temporal locality of accesses. Based on the data access pattern of the algorithm, the user-managed cache is devised, ensuring the data availability in the shared memory for a single thread. Finally, the serial algorithm is parallelized, and the cache management is refined to maximize the reuse of data by all the threads of a single thread block.

This approach enables us to construct an analytical model to quantitatively evaluate the effects of the cache parameters and cache implementation.

We apply this approach to the implementation of the sum-product (or marginalize a product of functions (MPF)) solver. MPF serves as a basis for many algorithms in artificial intelligence, bioinformatics, communications, signal processing, and others [14]. Our primary motivation for this research has been to accelerate the implementation of an instance of the MPF algorithm, used for inference in very large Bayesian networks. This problem arises in the context of genetic analysis of hereditary diseases [7], and may require years to complete on a modern CPU.

MPF can be considered a generalization of a matrix chain product for multidimensional matrices. However, it has more complex memory access pattern with the input-dependent memory reuse and large working set. Thus, to achieve high performance, the cache replacement policy should be determined at *run-time*, as opposed to a matrix product with static *compile-time* cache optimizations (e.g. cache blocking).

The GPU implementation with the user-managed cache achieves the average speedup of ˜500-fold for random data and ˜200-fold for real life Bayesian networks on NVIDIA GeForce 8800GTX GPU over the optimized CPU version on a single core of a 2.4 GHz Intel Core 2 processor with 4 MB L2 cache. In fact, for sufficiently large inputs the speedups reach a factor of *2700*. The main contributor to the speedup is the efficient use of the shared memory (around 24-fold for Bayesian networks and 52-fold for random data). Using the increased memory bandwidth provides further acceleration through the effective utilization of the GPU special function units.

We analyze the influence of the cache parameters on the overall performance, showing that it behaves as predicted by the model. We also compare the user-managed cache version with the one that uses a texture hardware cache, demonstrating superior performance of the former.

The paper is structured as follows. First, we introduce the GPU programming using CUDA and define the MPF problem. Then we describe the serial version of the MPF solver. We then develop a theoretical performance model for GPUs with the focus on the cache performance, and apply it to the MPF kernel. We proceed with the user-managed cache design and GPU kernel implementation. We conclude with the results and future research directions.

Related work.

The recent introduction of the IBM Cell [9] processor with software-managed per-core memory (local store) led to the development of techniques for utilizing that memory. However, Cell programming techniques are not applicable to a software-managed cache on NVIDIA's CUDA platform [12] because of the major architectural differences between the two. Still, some ideas inspired us to pursue the user-managed cache direction.

The most relevant work on Cell, by Benthin et al. [2], presents a software cache optimized for a Cell-based ray tracer. They address challenges similar to ours, such as trading an optimal cache policy for better cache logic performance and the non-uniform segmentation of the cache space for different tasks.

Another study is by Kamil et al. [10], where stencil kernels are optimized through the efficient use of the Cell local store. This work highlights the benefits of application-specific memory hierarchy management, though does not explicitly implement a cache.

The Cell implementation of the matrix product for the renowned LAPACK library is described by Kurzak et al. [11]. While the main focus is different from ours, the authors informally used the arithmetic intensity to analyze the performance.

Compiler-level generic cache implementations for Cell [1, 4] target general workloads and differ in scope from our work. The same holds true for the higher-level approaches such as the Sequoia [5] programming language for memory-hierarchy-aware parallel programs.

NVIDIA's CUDA programming guide [12] calls for optimizing the locality of the application to use the shared memory. Current work, however, the data access pattern is wholly known at compile time, rather than the more dynamic patterns that are the focus of this work. Also, no current work presents a general approach for designing cache organizations and evaluating their performance.

2. BACKGROUND

2.1 GPU programming and CUDA

The modern GPU is a highly data-parallel processor. The GPU features many lightweight closely-coupled *thread processors* that run in parallel. While the performance of each thread processor is modest, by effectively using many thread processors in parallel, GPUs can deliver performance that substantially outpaces a CPU.

The programming model of the GPU is "single-program, multiple data" (SPMD): many threads concurrently run the same program on different data. The GPU is most effective when thousands of threads are available to the hardware at any time; the GPU is capable of quickly switching between these threads to hide latency and keep the hardware busy.

The recent introduction of programming environments for the development of non-graphics applications on GPUs facilitated the use of GPUs for high performance computations. One such environment which we use in our work is NVIDIA's CUDA.

High-level programming environment.

CUDA programs are based on the C programming language, with extensions to exploit the parallelism of the GPU. CUDA programs are explicitly divided into code that runs on the CPU and code that runs on the GPU. GPU code is encapsulated into a *kernel*, which exemplifies the SPMD model: it looks like scalar C program, but is invoked concurrently in thousands of threads by the hardware. Typical CUDA programs will first set up input data on the CPU, transfer it to the GPU, run the kernel on the GPU data, and finally transfer the result back to the CPU.

Kernel code allows arbitrary read-write access to *global GPU memory*, which has no hardware cache. Instead, CUDA ex-

xyz	$f(x,y,z)$
000	f_{000}
......	..
112	f_{112}

wx	$g(w,x)$
0 0	g_{00}
.. ..	..
1 1	g_{11}

wy	$h(w,y)$
0 0	h_{00}
.. ..	..
1 1	h_{11}

(a)

xyzw	$\alpha(x,y,z,w) =$ $f(x,y,z) \times g(w,x) \times h(w,y)$
0000	$\alpha_{0000} = f_{000} \times g_{00} \times h_{00}$
0001	$\alpha_{0001} = f_{000} \times g_{10} \times h_{10}$
0010	$\alpha_{0010} = f_{001} \times g_{00} \times h_{00}$
....	..
1121	$\alpha_{1121} = f_{112} \times g_{11} \times h_{11}$

xz	$k(x,z) = \sum_{w,y} \alpha(x,y,z,w)$
00	$\alpha_{0000} + \alpha_{0100} + \alpha_{0001} + \alpha_{0101}$
01	$\alpha_{0010} + \alpha_{0110} + \alpha_{0011} + \alpha_{0111}$
02	$\alpha_{0020} + \alpha_{0120} + \alpha_{0021} + \alpha_{0121}$
....	..
12	$\alpha_{1020} + \alpha_{1120} + \alpha_{1021} + \alpha_{1121}$

(b)

Figure 1: Computing MPF: (a) Input functions. $|X| = |Y| = |W| = 2$, $|Z| = 3$ (b) Naive computation

poses low latency (˜1 cycle) memory shared among a subset of threads, called *thread block* (up to 512 threads per block). The threads of each block have an exclusive access to a small chunk (16 KB), and no access to the chunks of other thread blocks. No communication among the threads of different thread blocks is permitted.

Direct Compute Access.

NVIDIA GPUs feature multiple *multiprocessors* (16 multiprocessors in the GeForce 8800 GTX), each with 8 thread processors. The GPU is responsible for mapping the thread blocks to these multiprocessors, keeping thousands of threads "in-flight". If enough resources are available, each multiprocessor typically has multiple blocks resident, and can quickly switch between computation on different blocks when appropriate. For instance, if one block starts a long-latency memory operation, the multiprocessor will kick off the memory request then immediately switch to another block while those memory requests are satisfied.

2.2 Sum-product

Consider three functions, $f(x,y,z)$, $g(w,x)$ and $h(w,y)$ where w,x,y,z are variables over finite domains W, X, Y, Z of size $|W|$, $|X|$, $|Y|$, $|Z|$ respectively. An assignment to all the variables in the scope is called a *configuration*. A function is defined as a table with a single value per configuration of the function variables (Figure 1(a)). The set of variables in each function is called a function *scope*. In the rest of the paper we denote by $f_{a,b,c}$ the value of the function $f(x,y,z)$ for a particular configuration $x = a, y = b, z = c$.

The following operations are defined on the functions:

1. *Tensor product* $f \otimes g$ is a function $\alpha_{w,x,y,z} \triangleq f_{x,y,z} \times g_{w,x}$.

2. *Marginalization (summation)* over a variable x is a function
$\beta_{y,z} \triangleq \sum_{x \in X} f_{x,y,z}$.

Assume that we want to compute the following expression:

$$\sum_{w,y} f(x,y,z) \otimes g(w,x) \otimes h(w,y) \qquad (1)$$

The naive way is to first compute $\alpha(w,x,y,z)$ (Figure 1(b), top) and then marginalize out w and y (Figure 1(b), bottom). For the variables' domains of size n, this requires $O(n^4)$ operations.

xyz	$f(x,y,z)$
000	■ ■
001	▲ ▲
002	★ ★
010	■ ■
011	▲ ▲
012	★ ★
100	♠ ♠
110	♠ ♠

wx	$g(w,x)$
0 0	■ ■ ▲ ▲ ★ ★
0 1	♠ ♠
1 0	■ ■ ▲ ▲ ★ ★
1 1	♠ ♠

wy	$h(w,y)$
0 0	■ ▲ ★ ♠
0 1	■ ▲ ★ ♠
1 0	■ ▲ ★ ♠
1 1	■ ▲ ★ ♠

xz	$k(x,z)$
00	■
01	▲
02	★
10	♠

(a)

x **zy**	$f(x,z,y)$
00 0	■ ■
00 1	■ ■
01 0	▲ ▲
01 1	▲ ▲

xw	$g(x,w)$
0 0	■ ■ ▲ ▲ ★ ★
0 1	■ ■ ▲ ▲ ★ ★

(b)

Figure 2: MPF access pattern for computing k_{00}, k_{01}, k_{02} and k_{10} in Figure 1 (a) before reordering (b) after reordering of two unordered functions. Reordered variables are highlighted. Symbols denote accesses for computing respective output values.

Alternatively, we can apply the distributive law:

$$\left(\sum_{y} f(x,y,z) \otimes \left(\sum_{w} g(w,x) \otimes h(w,y) \right) \right). \quad (2)$$

The computation is split into two *buckets*. The expression in the innermost parentheses (first bucket) is computed first, and the result serves as the input for computing the expression in outer parentheses (second bucket). This leads to $O(n^3)$ total operations, i.e. $O(n)$ times less than before.

Unfortunately, the efficiency often comes at the price of additional space ($O(1)$ is required in the first case, $O(n^2)$ in the second).

The general MPF problem is:

$$\sum_{\mathbf{M}} \otimes_i f^i(\mathbf{X}^i), \quad \mathbf{M} \subseteq \bigcup_i \mathbf{X}^i, f^i \in \mathbf{F}, \qquad (3)$$

where $\mathbf{M}$ is the set of variables to be marginalized, and $\mathbf{F}$ is the set of all functions in MPF. MPF algorithms aim to efficiently compute the expression in Eq. 3 for any number of functions. Determining the interleaving of summations and multiplications which minimizes the number of computations under given memory constraints is NP-hard [7]. Pakzad and Anantharam [14] provide a comprehensive overview of MPF.

3. SERIAL MPF KERNEL

In this work we do not deal with the problem of finding the optimal order of operations (see Fishelson and Geiger [7] for a possible solution). Rather, we focus on the computational part that is common for many different MPF algorithms: computation of a single bucket, defined as follows:

$$\Psi(\mathbf{O}) = \sum_{\mathbf{M}} f^1 \otimes \cdots \otimes f^n, \qquad (4)$$

where $f^i \in \mathbf{F}$ are the functions in the bucket and $\mathbf{M}$ is the set of zero or more marginalization variables, $\mathbf{O} = \mathbf{V} \backslash \mathbf{M}$, $\mathbf{V}$ is the union of the variables of all the functions in $\mathbf{F}$. In order to solve a given MPF problem, the kernel is invoked for each bucket, processing one bucket after another in sequence.

We assume that the buckets are formed by an MPF algorithm under given memory constraints. Thus, the creation of

```
1:  Function SumProductKernel
2:  Input: Set of functions $\mathbf{F}$, union of functions' scopes $\mathbf{V}$, set of marginalization variables $\mathbf{M} \subseteq \mathbf{V}$
3:  Output: Function $\Psi$ with scope $\mathbf{O} = \mathbf{V} \backslash \mathbf{M}$
4:  for all configurations $\mathbf{p}$ of $\mathbf{O}$ do
5:      $sum \leftarrow 0$
6:      for all configurations $\mathbf{m}$ of $\mathbf{M}$ do
7:          $product \leftarrow 1$
8:          for all functions $f \in \mathbf{F}$ do
9:              $product \leftarrow product \times f(\mathbf{p}, \mathbf{m})$
10:         end for
11:         $sum \leftarrow sum + product$
12:     end for
13:     $\Psi(\mathbf{p}) \leftarrow sum$
14: end for
15: return $\Psi$
```

Figure 3: MPF kernel pseudocode

intermediate functions for computing a single bucket is disallowed, due to potential violation of the memory constraints.

The pseudocode for the single bucket computation is presented in Figure 3. For each output location, defined by the configuration of output function variables $\mathbf{O}$ (line 4), all configurations of marginalization variables are traversed (line 6). We denote by $f(\mathbf{p}, \mathbf{m})$ the value of f corresponding to the configuration $p \cup m$.

Input data access.

The data of a single function is organized in memory similarly to the multidimensional arrays in the C language. For example, function $f(x, y, z)$, $x \in X$, $y \in Y$, $z \in Z$ is represented as an array of size $|X| \times |Y| \times |Z|$. The value $f_{x,y,z}$ is located in the memory at the offset $z + |Z| \times y + |Y| \times |Z| \times x$. The *least significant* variable, i.e. the one whose sequential values correspond to adjacent memory locations in the function data array, is the last variable in the function specification (for $f(x, y, z)$, z is the least significant and x is the most significant).

The access to the function value for an arbitrary configuration of variables is costly due to the offset computation. To avoid such a computation for each access, a data traversal should be optimized to sequentially access both input and output arrays.

However, in general, such a traversal may not exist (as in the example in Figure 2(a)). However it becomes possible if we impose a global order on all the variables of the bucket. In our example, if the data is restructured according to the global order $x > z > w > y$ the traversal with sequential access is from the least to the most significant variable in the bucket (see Figure 2(b)).

For the complete MPF computation, where the output of one bucket is used as an input to another one, restructuring a single bucket layout is not enough. If the order of variables in a bucket contradicts that of the next one, the output must be restructured to comply with the new order, which is too costly if done for every bucket.

The solution is to impose a global order on all the variables in MPF as follows. The MPF algorithm prescribes the buckets to be processed in a certain order. Each bucket has a unique set of marginalization variables. *We order these sets in the reverse order of the buckets, assigning arbitrary order within each set of marginalization variables.* All non-marginalization variables are placed to be the highest in the order, and arbitrarily ordered among themselves. For our example in Figure 1, if we choose the efficient computation with two buckets (I) $\beta(x, y) = \sum_w g(x, w) \otimes h(y, w)$ and (II) $\sum_y f(x, z, y) \otimes \beta(x, y)$, the global order of the variables is $x > z > w > y$ (or $z > x > w > y$).

Once the input functions in all the buckets are restructured to follow the global order, no restructuring is required for the intermediate functions. The preprocessing cost is negligible.

4. CACHE PERFORMANCE MODEL

We aim to analytically evaluate the algorithm performance on the GPU in general, and the effect of caching in particular.

Our performance criterion is the number of floating point operations per second (FLOP/s) that can be achieved on a given processor for our application. To obtain an upper bound, we assume ideal overhead-free parallelization, which allows for the optimal utilization of GPU computational and memory resources. Hardware performance upper bounds are based on two parameters: 1. *the aggregated maximum rate of computations* of the GPU, denoted as P, measured in FLOP/s; 2. *memory bandwidth* of transfers between GPU global memory and ALUs, denoted as M, measured in floats/s.

The maximum performance is limited by P if the workload is CPU-bound. For memory-bound workloads, however, the memory subsystem becomes a bottleneck. The performance is limited by the memory bandwidth M multiplied by the compute-to-memory access ratio, also referred as arithmetic intensity [12], and denoted by A.

Since the memory accesses and the arithmetic operations in the GPU are overlapped, we obtain the following expression:

$$\textit{Speed} \quad = \quad \min\left[P, M \times A\right]. \tag{5}$$

Arithmetic intensity is application-dependent. To derive the general expression for the arithmetic intensity of the kernel we start with a simple MPF instance of computing the expression $k(x) = f(x) \otimes g(x)$. To produce one output k_x, *any* implementation must read two values, f_x and g_x, from the memory and write the result k_x back — total 3 memory accesses — versus one floating point operation per one output. Thus, $A = \frac{1}{3}$, for any kernel implementation. Using Eq. 5, for NVIDIA GeForce 8800GTX GPU with P=345 GFLOP/s and M=22 GFloat/s, *Speed*=7.3 GFLOP/s, which is only 2% of GPU's potential.

Note that the caching would not improve the performance because all the data is read only once. In order to incorporate caching into the performance analysis, consider the matrix product example, which is also an instance of MPF. Consider two matrices $M \times N$ and $N \times K$. For every output, the are $2N+1$ memory accesses ($2N$ reads and 1 write). However, assuming infinite cache (only compulsory misses) with zero-cost access, the *cost* of memory operations per output drops to $\frac{N}{K}$ and $\frac{N}{M}$ for the first and second matrices respectively. Thus, the arithmetic intensity for matrix product is $\frac{2N-1}{2N+1}$ without cache and $\frac{2N-1}{N(\frac{1}{M}+\frac{1}{K})+1} = \frac{2-\frac{1}{N}}{\frac{1}{M}+\frac{1}{N}+\frac{1}{K}}$ with the infinite cache. To conclude, the lack of caching leads to constant performance upper bound 1×22 GFLOP/s, whereas with caching it becomes CPU bound (if $M = K$, and N is large, then $A = K$).

We define a new parameter, *cached arithmetic intensity*, denoted A_{cache}, to account for the cost of memory accesses. We derive the expression for A_{cache} (Eq. 6) for the MPF kernel.

$$A_{cache} \quad = \quad \frac{\#m - \frac{1}{N}}{\sum_i^m c_i + \frac{1}{N}}, \tag{6}$$

where c_i is the cache miss rate for accessing function i, m is the number of input functions, and N is the number of configurations of the marginalization variables. Note that for the parameters of the first example, Eq. 6 yields the intuitive results: $m = 2$, $c_i = 1$ (100% compulsory misses), $N = 1$, $A_{\text{cache}} = A = \frac{1}{3}$.

Clearly, caching is vital for achieving high performance of MPF kernel. Even the low cache hit rate of 50% for all functions leads to a two-fold performance increase versus non-cached version, growing to ten-fold for the hit rate of 90%.

In general, the addition of caching can potentially transform the problem from memory-bound to compute-bound, delivering a substantial increase in performance.

5. USER-MANAGED CACHE

We see that the key to high performance for MPF computations is an efficient use of the memory hierarchy.

An algorithm for the GPUs with a software-managed memory hierarchy can be logically divided into the memory management part that implements the caching functionality and a computational part that is structured to use the cached data. However, this approach brings us to a chicken-or-egg problem: computations should be structured according to the data availability in the shared memory, while the data should be staged according to the structure of the computation.

This mutual dependency can be resolved as follows. Cache space limitations require the computational structure to maintain temporal locality of accesses in order to reduce capacity misses, regardless of the cache algorithm. This suggests to first determine the computational structure with temporal locality, ignoring spatial locality, and then to derive the cache management algorithm, which stages the data accessed close in time but coming from arbitrary locations.

Unfortunately, the spatial locality requirement is critical when designing a cache for NVIDIA GPUs, since the effective memory bandwidth drops by up to a factor of 16 if the kernel does not concurrently access sequential memory locations (called "coalesced accesses").

We conclude that, similarly to a CPU implementation, the GPU computational kernel should maintain both spatial and temporal locality. The caching part of the kernel should simulate the behavior of a hardware-managed CPU cache by determining which data to stage. However, the main difference between the hardware and user-managed caches is that the *replacement policy* is implemented by the software.

In the following we first analyze the locality properties of the computational kernel, determine the optimal traversal over the input data, and then derive the cache algorithm.

Spatial and temporal locality.

The spatial locality is naturally improved thanks to the restructuring of the data layout as discussed in Section 3, and traversing it from the least to the most significant variable in the global order. This is because this order results in the input and output functions to be accessed in blocks (as in the example in Figure 2(b)).

However, the traversal with the best spatial locality may conflict with the one with the best temporal locality. For example, for the bucket $f(x, y, z) \otimes g(y, z)$, the traversal over the values of x has the optimal temporal locality (a single value of $g(y, z)$ is reused for each value of x), but poor spatial locality (x is the most significant in $f(x, y, z)$).

While every bucket has a traversal order that achieves the best temporal locality, *we chose to prefer spatial locality over temporal locality* because otherwise: 1. index computation is required for every access, decreasing the common case performance; 2. data staging results in non-coalesced accesses, reducing the effective memory bandwidth significantly.

5.1 Cache design

What data to cache.

To identify the input data locations that are accessed close in time, we introduce the notion of an *index vector*. Index vector is a vector of integers with each location corresponding to a variable, and the value at each location corresponding to the value of the respective variable. It is convenient to think of an index vector as of a number represented in a mixed radix, where the numerical base for each location corresponds to the domain size of the respective variable. We call two index vectors *subsequent* if their numerical values differ by 1.

Consider the index vector over all the variables in a bucket. The variables are ordered according to the global order, discussed in Section 3. We call this vector *a bucket address*. A given bucket address identifies one value in every input and output array, hence its name. In the example in Figure 2(b), for the order $x > z > w > y$, the bucket address 1210 implies $x = 1$, $z = 2$, $w = 1$, and $y = 0$, referring to the respective values f_{120}, g_{11}, h_{10} and k_{12}. According to the traversal order determined in the previous subsection, the algorithm processes the bucket by iterating over subsequent bucket addresses, starting from zero. Thus, the data, which corresponds to a set of subsequent bucket addresses, and sharing the same values of the most significant digits reside in a contiguous chunk of the input and output arrays. Clearly, these data are accessed close in time and should reside together in the cache.

Cache structure.

The cache is split into multiple cache segments, one per input function (output is not cached).

The location of the data in the function's cache segment is determined by the *cache tag*. The cache tag is a subset of the least significant digits of the bucket address. We denote by $\mathbf{C}$ the variables that correspond to the cache tag, and by $\mathbf{C_f}$ the intersection of $\mathbf{C}$ with the function's variables. The size of function's cache segment is the product of the domain sizes of the variables in $\mathbf{C_f}$. The total size of the cache is the sum of the segment sizes of the functions to be cached. The algorithm for determining the number of digits in the cache tag is described later.

For the bucket in Figure 2(b) and cache tag of size three, $\mathbf{C} = \{z, w, y\}$; $\mathbf{C_f} = \{z, y\}$, $\mathbf{C_g} = \{w\}$, $\mathbf{C_h} = \{w, y\}$. The cache stores 6 values for f, 2 for g and 4 for h—a total of 12 values.

Consider the data identified by a set of subsequent bucket addresses which differ only in their cache tags. We call these data a *cache page*. The subset of the most significant digits of the bucket address, which are shared among all the data in a cache page is called a *cache page tag*. Note that the algorithm traverses the cache pages according to the order of the cache page tags, starting from first page with the cache page tag zero.

5.1.1 Conflict misses

The data from the different cache pages but with the same cache tag are mapped onto the same physical cache location. Accessing these data leads to the tag conflict misses.

The granularity of the cache invalidation upon a miss is critical. While handling each miss individually is too costly, replacing the cache page entirely upon switching to another one reduces the hit rate.

The key is *to consider the replacement policy for each function segment separately*. We note that the data of some functions are shared among multiple subsequent cache pages and should not be invalidated. In the example in Figure 1, with the cache tag $\{z, w, y\}$, and the cache page tag $\{x\}$, moving from the cache page $x = 0$ to $x = 1$ requires refreshing only the content of the f and g segments, since the function h does not depend on x. The subset of functions to be refreshed per cache page is precomputed on the CPU prior to the GPU invocation, and is checked when the next cache page is reached.

5.1.2 Capacity misses

The amount of data to be cached, as prescribed by the cache tag, might be too large to fit the available shared memory, leading to capacity misses.

Similarly to the conflict miss resolution, the capacity misses are resolved separately for each function. We *entirely disallow caching of some functions when the size of all segments together exceeds the available shared memory*. The functions that are not cached are accessed directly from the global memory, bypassing the cache.

However, this partial caching leads to the problem of choosing an optimal subset of functions to be cached.

We formally define this problem as follows. We define the cache utilization $U = \sum_{f \in functions} \frac{lifetime_f}{size_f} \times cached_f$, where $cached_f$ is 1 if function is cached and 0 otherwise, and $size_f$ is the size of the cache segment of f, $lifetime_f$ is the number of sequentially accessed cache pages that share the same segment of the function data. We aim at maximizing U under the constraint $\sum_{f \in functions} size_f \times cached_f \leq Shared\ memory\ size$.

This problem is the classical (NP-hard) binary knapsack problem where the cost of adding a function segment to the cache is $\frac{lifetime_f}{size_f}$. The well-known two-approximation algorithm is to add the items in decreasing order of cost. The greedy algorithm is exectued on the CPU prior to the GPU invocation, and the results are checked on the GPU every memory access, fetching the data either from the cache or from the global memory.

In both cases of the conflict and capacity misses,*the replacement policy is determined dynamically on the CPU prior to the GPU invocation, leaving only the policy enforcement mechanism to be executed by the GPU.*

6. CACHE-AWARE PARALLEL KERNEL

We parallelized the MPF kernel over the output domain by assigning a subset of the output values to each thread. The threads are independent, allowing as many of them to be run in parallel as the hardware permits.

However, efficient utilization of the user-managed cache requires the threads to be coscheduled, and the outputs to be assigned to allow localized data accesses to the *same* cache page.

Coscheduling, i.e., concurrent execution of a certain group of threads, is a built-in feature of CUDA, realized by a *thread block* (*TB*). The threads within a TB are executed either concurrently or in a known order. They communicate through the shared memory, accessible only to the threads of that TB.

Our cache design naturally maps onto the shared memory. Thus, the threads of a TB are coscheduled to concurrently access a certain cache page. Each TB is assigned a set of subsequent cache pages, processed sequentially by that TB.

The current implementation stores one cache page at a time in the shared memory of each TB, simplifying the memory offset computations per access at the expense of reduced hit rate.

The amount of input data per cache page is determined as follows. A certain cache page is concurrently processed by all the threads in a TB, and the respective input data must reside in the cache. To exploit all the available hardware parallelism, we assign each thread to process a single output per cache page. Thus, the size of the input data per cache page is dictated by the number of concurrently processed output locations, which in turn is limited by the number of threads in a TB.

The number of cache pages per TB deserves special attention. While assigning more pages to a TB improves the data reuse, it also reduces the parallelism, and potentially the hardware utilization, because of a sequential processing of different pages by a given TB. *We assign the pages to a single TB as long as the resulting number of threads is above the maximum number of threads which can be executed concurrently by the hardware.*

6.1 Implementation

The GPU implementation consists of two modules: the main GPU kernel that implements the parallel MPF computation, and the preparatory CPU module that creates the data structures and cache metadata, and transfers the data from the CPU to the GPU memory. Due to the space limitations we skip the details of the CPU module.

6.1.1 GPU kernel

The presented kernel is executed in each thread (Figure 4). We use the names starting with the capital letters to denote the data precomputed on CPU. The data is stored in the read-only GPU memory, which is augmented with the hardware cache (texture and constant memory). The two parameters *tBlockID* and *threadID* are supplied by the hardware, and used by each thread to determine which data to process.

The kernel can be logically split into four parts: computation of the global memory offsets (lines 3–7), cache prefetching (7–15), computation loop (15–29), and writing back the result (29).

First, the kernel determines the set of cache pages to be processed by a given TB, together with the data offsets in the global GPU memory.

Next, the input data for a given cache is staged into the shared memory (line 11). If the function segment has to be refreshed upon switching to the next page (line 9), the thread must wait until all the threads in its TB stop using that cache page, before rereading its content from the global memory

```
1:  Function SumProductGPUKernel
2:  Input: thread block ID tBlockID, thread ID threadID, functions $\mathbf{F}$,
    marginalization variables $\mathbf{M}$, #cache pages per TB TBPages
3:  outputOffset $\leftarrow$ call computeOutputOffset(tBlockID)
4:  for all input functions $f \in \mathbf{F}$ do
5:      inputOffsets$[f] \leftarrow$ call computeOffset(tBlockID, $f$)
6:  end for
7:  for page $= 0$ to TBPages do
8:      for all input functions $f \in \mathbf{F}$ do
9:          if CacheValidArray[page]$[f]$ is false then
10:             call barrier()
11:             call populateCache($f$, CacheSegmentSizes$[f]$, inputOffsets$[f]+$
                PageOffsets$[f]$[page])
12:         end if
13:     end for
14:     sum $\leftarrow 0$
15:     for sumPtr $= 0$ to #configurations of variables in $M$ do
16:         product $\leftarrow 1$
17:         for all input functions $f \in \mathbf{F}$ do
18:             offset $\leftarrow$ call computeBase(ThreadOffsets[f][threadID],
                sumPtr)
19:             if CachedFunctionsArray$[f]$ is false then
20:                 value $\leftarrow$ call cacheFetch(offset)
21:             else
22:                 offset $\leftarrow$ offset $+$ inputOffsets$[f]$ $+$ PageOffsets$[f]$[page]
23:                 value $\leftarrow$ call memoryFetch(offset)
24:             end if
25:             product $\leftarrow$ product $\times$ value
26:         end for
27:         sum $\leftarrow$ sum $+$ product
28:     end for
29:     output[outputOffset $+$ page $\times$ ThreadBlockSize $+$ threadID] $\leftarrow$ sum
30: end for
```

Figure 4: GPU kernel pseudocode

(line 10).

The main computation starts by reading the input data either from the cache or from the main memory, depending on whether the function is cached or not (line 19). The rest is similar to the serial algorithm in Figure 3.

Finally (line 29) the value is stored in the main memory, and the kernel starts processing the next cache page.

7. RESULTS

7.1 Experimental setup

We evaluate the MPF kernel on NVIDIA's GeForce GTX8800 graphics card, with 128 thread processors and 750 MB of global memory. The CPU version is invoked on a single core of an Intel Core 2 2.4 GHz CPU with 32KB L1 and 4 MB L2 cache. The CPU version is optimized for optimal caching and performs within 5% of the performance of the matrix multiplication routine of the ATLAS [16] library when applied to a regular matrix product.

The performance criterion for a single kernel invocation is the number of operations per second, measured in FLOP/s. The number of operations—the input complexity—includes only the multiplications and summations required by the algorithm.For example, the multiplication of two 2×2 square matrices requires 12 FLOP.

We invoke the kernel on the same input until the accumulated running time exceeds five seconds, and then derive the time for a single invocation. Kernel invocation overhead (˜10 μs) is ignored.

Unless otherwise stated, we report only the pure GPU execution time, excluding the time for data transfers between the CPU and the GPU, and for computing the cache metadata on CPU. The results of GPU and CPU runs are compared to ensure correctness.

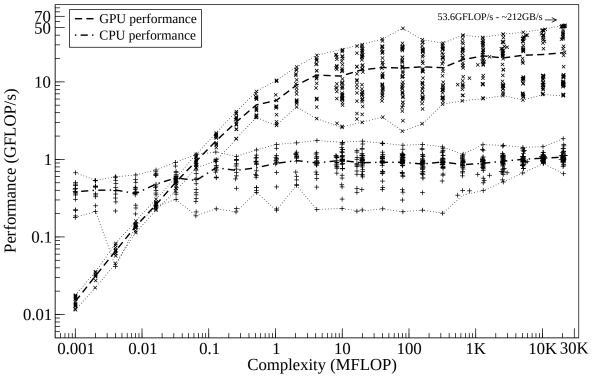


Figure 5: Linear-domain performance on random data sets

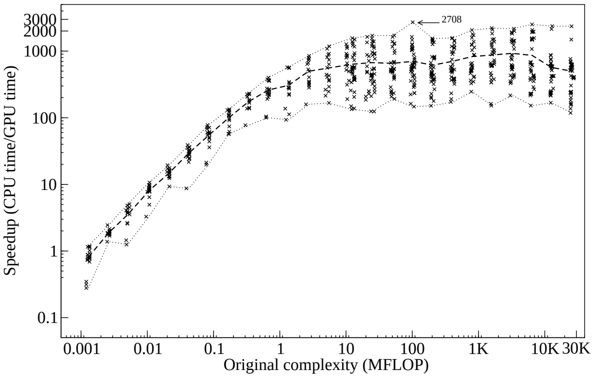


Figure 6: Log-domain speedups on random data sets

7.2 GPU versus CPU performance

7.2.1 Summary of the results

The following table summarizes the speedups obtained in the experiments described in this section.

Benchmark	GPU speedup over CPU		
	peak	**average**	**min**
Random benchmark (log-domain)	2708	536	0.2
Random benchmark (linear-domain)	52	15	0.02
Bayesian networks (log-domain)	274	200	153
Bayesian networks (linear-domain)	24	15	12

7.2.2 Random datasets

The tests aim to evaluate the kernel performance on the inputs having different amount of available parallelism and different reuse patterns. For the small inputs with low parallelism, and for the inputs with limited data reuse, the GPU performance is expected to be low.

We generated 700 buckets of different computational complexity with the parameters randomly chosen from the following ranges: 2–4 functions per bucket, 1–10 values per variable, 2–32 summation values, 1–18 variables shared between the functions, and 5–25 variables per function. These are the typical ranges for the Bayesian networks in the genetic analysis domain. The function data is strictly positive.

Figure 5 shows the kernel performance on random data as a function of the input complexity. Each dot represents the kernel invocation for computing one bucket.

On average, the GPU performs an order of magnitude better than the CPU for most inputs above 100 KFLOP (about the complexity of a product of two 40×40 matrices). Below this threshold, the input size does not allow full utilization of the hardware.

The peak performance of 53.5 GFLOP/s corresponds to the effective memory bandwidth of about 212 GB/s, which would be impossible without the use of shared memory.

7.2.3 *Bayesian networks*

The complete instance of the MPF problem often contains thousands of buckets of various complexity. To evaluate the performance on real MPF instances, we used Superlink [7] to generate Bayesian networks from the real-life genetic analysis data and created the buckets for MPF computation. We then invoked GPU and CPU versions of the kernel, and summed the single bucket running time for all buckets in the MPF. *We obtained an average speedup of 15* (ranging between 12 and 24) over 20 networks of different complexity.

The analysis of the bucket complexity distribution revealed that over 85% of all buckets in each Bayesian network are below a 100 KFLOP threshold. In fact, using GPU for these inputs slows down the execution. However, the overall performance is accelerated due to a few large buckets that contribute over 98% to the running time.

7.2.4 *Log-domain performance*

In practice, inference in Bayesian networks requires all operations to be performed in log domain in order to prevent accumulation of numerical errors. The input functions are converted to the log domain before the computation; multiplications are replaced by summations; summations convert the partial results to the linear domain and write back the results in log domain.

Figure 6 shows the speedups obtained when invoking CPU and GPU log-domain versions on the same set of random buckets as in Figure 5. Observe that the Figure 5 is scaled by a factor of about 100, resulting in speedups of *three orders of magnitude*. This is due to the use of the GPU special function units, which supports fast computing for logarithms and exponents in hardware. However, *disabling the user-managed cache reduces the speedup by a factor of 25 on average*.

For Bayesian networks we achieve a speedup of up to *274*.

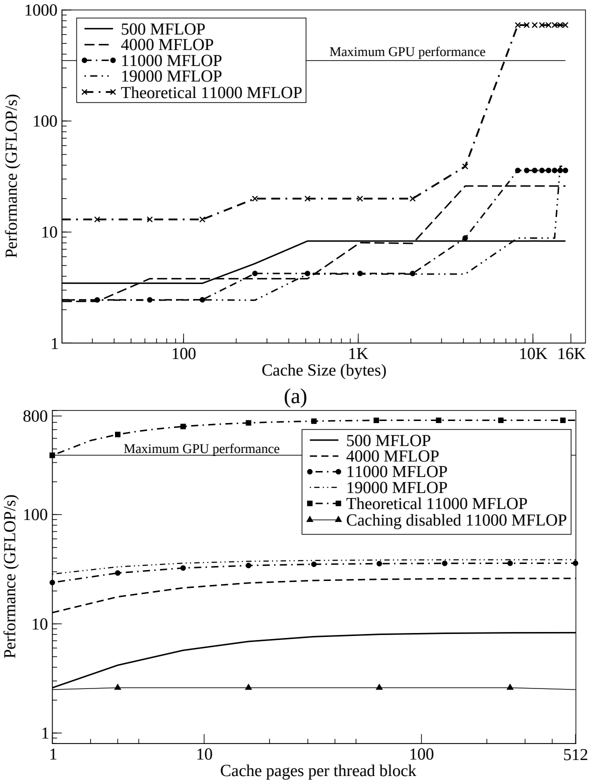
While we note that the real applications will certainly use log domain computations, we evaluate the linear domain implementation to emphasize the contribution of the user-managed cache to the kernel performance.

7.3 Cache performance

We analyzed the impact of cache size and the number of cache pages per TB on kernel performance. We used several buckets of different complexity with 3 functions per bucket, and fixed them throughout the experiment, while varying the parameter of interest. All the functions in the bucket together fit the cache in the default kernel configuration.

The results are presented in Figure 7. Both graphs also show the theoretical performance for one of the inputs as predicted by the model in Section 4. While the absolute predicted values are much higher than those observed, the form of the experimental graphs closely follows the one predicted by the model.

Increasing the cache space (Figure 7(a)) allows for caching more functions, as can be seen from the staircase form of the graph. Three “knees” in the graph match the number of functions in the bucket. According to the model, the hit rate changes from 0 (no cache), to 33.3%, 65.5% and 98.9% (all functions cached). Consequently, the A_{cached} increases from 0.99 to 1.4, 2.14 and 52 respectively, explaining the sharp improvement in the performance.



(a)

(b)

Figure 7: Cache performance

The impact of the number of cache pages processed by one TB is depicted in Figure 7(b). As expected, the performance improves due to the increased data reuse. Thus, as the graph shows, this parameter has no effect when the caching is disabled. The improvement becomes less significant with the higher values of the parameter due to the increased number of tag conflict misses and decrease in the data reuse between different cache pages. For the analyzed input, the hit rate changes only for the first 64 cache pages, from 96.8% to 98.8%. Without the cache collisions (only compulsory misses), the hit rate would reach 99.9%, doubling the performance.

The asymptotic behavior of the graph in Figure 7(b) can be explained by the model as follows. Theoretically, if the data allowed for a 100% cache hit rate, the arithmetic intensity would tend to infinity, as no global memory read accesses would be required. However, this is not the case due to the summand $\frac{1}{N}$ in the denominator in the expression in Eq. 6. This summand results from the cost of writing the results back to the global memory. Although in the GPU write accesses are asynchronous and assumed to have zero latency, they do consume bandwidth, which is apparent in the theoretical and the experimental performance graphs.

7.3.1 *Loop unrolling*

The impact of loop unrolling is shown in Figure 8. The chosen inputs fit the cache. We measured the performance as a function of the input complexity. There are two types of

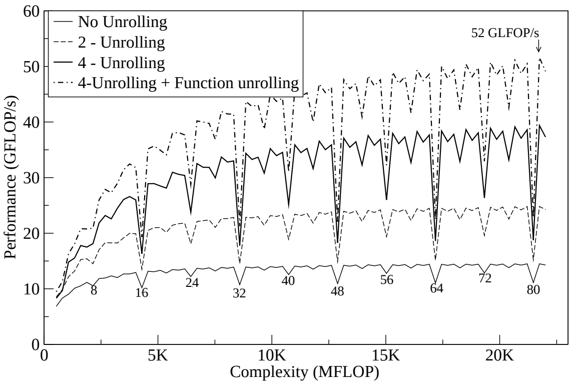


Figure 8: Loop unrolling

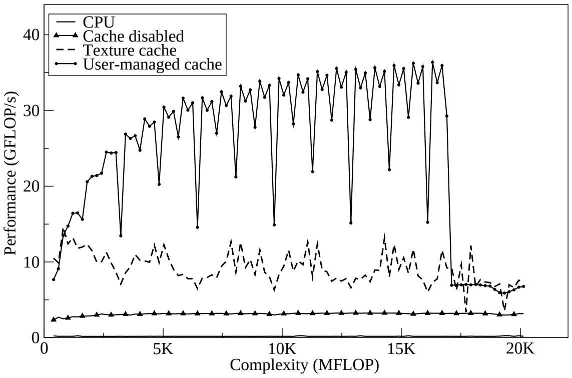


Figure 9: User-managed vs. texture cache

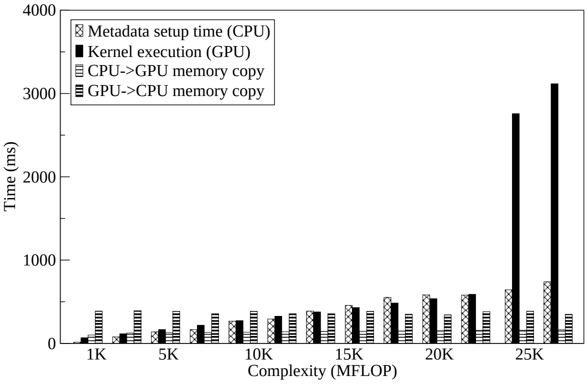


Figure 10: Overhead analysis

unrolling presented—dynamic unrolling of the loop over the summation variables (line 15 in Figure 4, and static unrolling over input functions (line 17).

Dynamic unrolling is required for the loops for which the upper bound is unknown at compile time. It is performed by creating the unrolled loop versions with 1, 2 and 4 unrolled iterations, and by dynamically choosing the appropriate version depending on the number of remaining iterations.

Dynamic unrolling of both loops was not possible due to the increased register use per thread, which led the compiler to spill out some of the register to the global memory, significantly decreasing the performance. Thus, we created different versions of the kernel with static unrolling of the loop over the input function.

Dynamic unrolling (curves for 2- and 4-unrolling) boosts the performance significantly (up to a factor of 3), because of the reduced number of accesses to the indexing tables and cache metadata, performed once per iteration, instead of once per access. The static unrolling yields an additional speedup of about 25% over the 4-unrolled version.

The saw-like form of the graph in Figure 8(a) is due to the difficulty in utilizing the shared memory hardware efficiently. Shared memory consists of 16 memory banks, and the best performance is achieved if the threads concurrently access different banks, or the same data of the same bank. Otherwise, *bank conflicts* result in the serialization of accesses. For some inputs the data layout in the cache led to bank conflicts (in particular when the number of summation values is divisible by 16).

7.3.2 GPU texture cache comparison

We compare the user-managed cache with the hardware texture cache, by removing all the cache logic and replacing the global memory reads by the texture reads. Such a comparison is not completely fair, as the algorithm is not optimized for the texture cache. On the other hand, the texture cache has many other advantages, such as the ability to cache data across thread blocks and the implementation of the cache logic in hardware.

The results are depicted in Figure 9. As long as the data fits the cache, both implementations perform equally well. However, for the inputs with a working set of about 1 KB per thread block, texture cache performance deteriorates, despite the working set of the texture cache being 8 KB [12]. For the inputs of complexity above 18 GFLOP, the user-managed cache is no longer able to cache all the functions, exhibiting performance similar to that of the texture cache.

7.3.3 Analysis of the overheads

In all the results above we analyzed only the GPU execution time. In this section (Figure 10) we analyze all the overheads: setup time for computing the cache metadata and determining kernel parameters, and data transfers between the CPU and GPU.

We see that the kernel execution does not dominate the overall execution time in the cases when the data fits the cache. However, these high overheads can be hidden in the repetitive kernel invocations on many buckets. First, the metadata can be computed on the CPU in parallel with the GPU processing of the previous bucket. Furthermore, CUDA soon will allow asynchronous memory transfers both to and from the GPU. Finally, we will be able to completely avoid memory transfers of the intermediate results by using the MPF algorithm that make the whole MPF computation fit the available memory [7].

8. CONCLUSIONS AND FUTURE WORK

In this work we have demonstrated the design and implementation of a direct-mapped read-only user-managed cache on a data-parallel graphics processor. The speedups of up to three orders of magnitude allow greatly improved performance of a wide range of applications of sum-product kernel, including the one for Bayesian inference for genetic disease analysis.

Our short-term future work includes the integration of the kernel into a real Bayesian networks solver through further exploitation of GPU-CPU parallelism, and continued improvement of cache utilization.

An open question is whether the explicit cache management on GPUs is applicable to algorithms with a more irregular access pattern, in which the cache decisions must be made dynamically as a part of the computation kernel, as opposed to the compile-time static decisions in the previous works, extended in this research to the data-dependent decisions made during the preprocessing step on the CPU. It seems that the current hardware makes such management rather difficult, first, because there is insufficient fast memory for cache metadata, and second, because there is a lack of specialized hardware to implement the basic cache primitives. Furthermore, the lack of fine-grained synchronization between the threads within a thread block would make the dynamic update of individual memory locations very inefficient, limiting the effective parallelism. The addition of specialized hardware mechanisms is likely to widen the range of high performance applications that could benefit from the GPU's computing power.

9. REFERENCES

[1] J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, Z. Sura, T. Chen, T. Zhang, K. O'brien, and K. O'brien. A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor. In *LCPC '07: Proceedings of the 2007 Workshop on Languages and Compilers for Parallel Computing*, 2007.

[2] C. Benthin, I. Wald, M. Scherbaum, and H. Friedrich. Ray Tracing on the Cell Processor. *IEEE Symposium on Interactive Ray Tracing 2006*, pages 15–23, Sept. 2006.

[3] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *DAC '00: Proceedings of the 37th Conference on Design Automation*, pages 416–419, New York, NY, USA, 2000. ACM.

[4] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing Compiler for the CELL Processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[5] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 83, 2006.

[6] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Graphics Hardware 2004*, pages 133–138, Aug. 2004.

[7] M. Fishelson and D. Geiger. Exact genetic linkage computations for general pedigrees. *Bioinformatics*, 18(Suppl. 1):S189–S198, 2002.

[8] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, page 89, Nov. 2006.

[9] IBM Corporation. Cell Broadband Engine Architecture. `http://www.ibm.com/techlib/techlib.nsf/techdocs`.

[10] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *MSPC '06: Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, pages 51–60, New York, NY, USA, 2006. ACM.

[11] J. Kurzak, W. Alvaro, and J. Dongarra. Fast and small short vector SIMD matrix multiplication kernel for the synergistic processing element of the CELL processor. Technical Report LAPACK Working Note 189, University of Tennessee, 2007.

[12] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide. `http://developer.nvidia.com/cuda`, Jan. 2007.

[13] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[14] P. Pakzad and V. Anantharam. A new look at the generalized distributive law. *IEEE Transactions on Information Theory*, 50(6):1132–1155, June 2004.

[15] M. Peercy, M. Segal, and D. Gerstmann. A performance-oriented data parallel virtual machine for GPUs. In *ACM SIGGRAPH 2006 Conference Abstracts and Applications*, Aug. 2006.

[16] R. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27:3–35, 2001.