Applying software-managed caching and CPU/GPU task scheduling for accelerating dynamic workloads

Mark Silberstein [*] Assaf Schuster[*] John D. Owens [†]

February 6, 2011

In this chapter we cover two difficult problems frequently encountered by GPU developers: optimizing memory access for kernels with complex input-dependent access patterns, and mapping the computations to a GPU or a CPU in composite applications with multiple dependent kernels. Both pose a formidable challenge as they require *dynamic adaptation and tuning of execution policies* to allow high performance for a wide range of inputs. Not meeting these requirements leads to substantial performance penalty.

In the first part of the chapter we describe our methodology for solving the memory optimization problem via *software-managed caching* by efficiently exploiting the fast scratchpad memory. This technique outperforms the cache-less and the texture memory-based approaches on pre-Fermi GPU architectures as well as the one that uses the Fermi hardware cache alone.

The focus of the second part is the algorithm for minimizing the total running time of a complete application comprising multiple interdependent kernels. Both a GPU and a CPU can be used to execute the kernels, but the performance varies greatly for different inputs, calling for dynamic assignment of the computations to a GPU or a CPU at runtime. The communication overhead due to the data dependences between the kernels makes per-kernel greedy selection of the best performing device suboptimal. The algorithm optimizes the runtime of the complete application by evaluating the performance of all the assignments jointly, including the overhead of the data transfers between the devices.

We demonstrate these techniques by applying them to a real application for computing probability of evidence in probabilistic networks. The combination of memory optimization and dynamic assignment results in up to three-fold runtime reduction over the non-optimized version on real inputs, and up to five-fold over a highly optimized parallel version running on Intel's latest dual quad-core 16-thread Nehalem machine.

1 Introduction, Problem Statement, and Context

This chapter endeavors to assist developers in overcoming two major bottlenecks of the high-end GPU platforms: memory bandwidth to the main (global) memory of the GPU, and the CPU-GPU communications. We faced both these problems when developing an application for computing the probability of evidence in probabilistic networks, and only by solving both we achieved the desired performance improvement. Yet, we believe that our techniques are applicable in a general context, and can be employed together and separately. In the chapter we first describe the solution for each problem, and conclude by demonstrating their combined effect on a real application as a whole.

Memory access optimization is among the main tools for improving application performance in CPUs and GPUs. It is of added importance if the algorithm has a low compute-to-memory access ratio. Often the same

[*]Computer Science, Technion – Israel Institute of Technology

[†]Electrical and Computer Engineering, University of California at Davis

data are reused many times, and reorganizing the computations to exploit small but fast on-die caches might thus reduce the main memory bandwidth pressure and improve performance.

Hardware caches employ input-independent replacement algorithms, such as Least Recently Used (LRU). Maximizing cache performance to exploit data reuse requires restructuring the code so that the actual access pattern matches the cache replacement algorithm. Unfortunately, high performance is difficult and sometimes even impossible to achieve without the ability to control the replacement decisions.

Modern NVIDIA GPUs expose fast scratchpad memory shared by multiple streaming processors on a multiprocessor. By design, the scratchpad memory lacks hardware caching support[1]; hence it is the responsibility of the kernel to implement a *software-managed cache*, which implies determining which data to stage from the main memory and when to stage it. For cases where this determination is data-dependent, the decision must be made at runtime. The main challenge, then, is to minimize the overhead of the cache management code, which resides on the critical path of every memory access. In the first part of the chapter we introduce techniques for analyzing the data access patterns and designing a read-only low-overhead software-managed cache for NVIDIA GPUs.

Kernel performance optimization, however, is only one component of making the complete application run faster. Often, despite optimizations, the kernel performance may vary substantially for different inputs. In some cases executing the kernel on a GPU may actually decrease the performance, such as when not enough parallelism is available. Furthermore, the overhead of the CPU-GPU communications over the PCI Express bus may reduce or completely cancel out the advantages of using a GPU. In the second part of the chapter we focus on optimizing the choice of the processor for the kernel execution in applications with multiple inter-dependent kernels.

A simple approach is to greedily assign the device providing the best overall performance for a given input. It will work well for isolated kernels, where both the kernel input and output must reside on a CPU. For such cases, the data will always be transferred from the CPU to the GPU and back, thus allowing for a *local* decision that considers only the performance of a given kernel on each device.

However, for applications composed of multiple kernels with data dependencies, whereby the subsequent kernels use the results of the previous ones, different assignments or *schedules* of the computations on a CPU or a GPU may decisively influence the running time of the complete application. The schedule, which optimizes the performance of each kernel separately, is no longer sufficient for obtaining the best performance of the application as a whole.

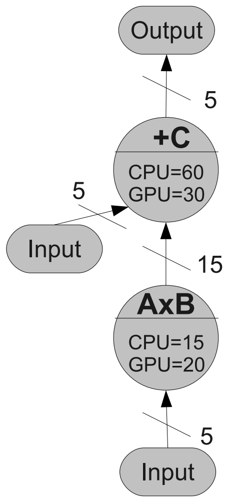


Figure 1: An illustration of the program task dependency graph for computing $A \times B + C$ of matrices A, B, C.

Figure 1 shows a *task dependency graph* of a program for computing $A \times B + C$ for three matrices A, B, C. The nodes and edges of the graph denote kernels and their data dependencies respectively. Computations are performed by traversing the graph according to the directionality of the edges. The computations of a node can be started only if all its predecessors in the graph are complete. In this example the first kernel computes $A \times B$ and the second one adds C to the result. The respective graph node labels denote the expected running time (the lower the better) of the kernel on a CPU or a GPU. Edge labels denote the data transfer times given that the adjacent nodes are executed on different devices. Input data nodes represent the original input data residing in CPU memory.

Were the schedule to consider the performance of each kernel alone, it would assign the product kernel to a CPU and the summation kernel to a GPU, yielding an execution time of 65 time units. (We assume that

[1]The on-die memory in the Fermi architecture is partitioned into a hardware cache and a scratchpad; in this chapter we focus on the efficient use of the latter.

input transfer of matrix C for the summation kernel can be overlapped with the execution of the product kernel on matrices A and B.) However, the best schedule requires only 60 time units to complete, assigning both kernels to a GPU. Note that the higher cost of the data transfer between two kernels would increase the performance gap between the greedy and the optimal schedules.

We show a simple and fast algorithm which solves this scheduling problem for task dependency trees (task graphs without undirected cycles). Although the algorithm does not produce an optimal schedule (finding the optimal schedule is known to be computationally hard), it has been shown to improve the performance in real-life computations. Its main advantage is that it does not require changing the original sequential program flow, complementing other optimizations such as overlapping the data transfers with the kernel execution.

Combining the software-managed caching and GPU-CPU scheduling yields marked performance improvements over the version which does not use them. We compared the performance on random and real-life inputs using three generations of NVIDIA GPUs: GeForce 8800 GTX, GeForce GTX 285, and the Fermi-based Tesla C2050. Finally, with these techniques we obtained up to a factor of 5 speedup over the CPU-only parallel version executed on the latest dual quad core Intel Nehalem E5540 CPUs.

2 Core Method

We first demonstrate an efficient software-managed caching scheme which provides a structured approach to using the scratchpad memory. We emphasize that our method is applicable to applications where static prefetching is not possible due to the input-dependent data access pattern. Cache management at runtime would incur high overhead, counteracting the benefits of using the scratchpad memory. Our key idea is to *precompute the access pattern on a CPU for each input* before the kernel execution and make the results available to a GPU via *cache policy* in the form of lookup tables used by the kernel at runtime. Not only does such a structured approach yield substantial speedups even over the implementation that uses the hardware cache alone, it also facilitates the development process by allowing a separation of concerns between data management and computation.

We then apply a graph-theoretical approach to optimizing the execution of multi-kernel composite applications with inter-kernel data dependencies and input-dependent performance of each kernel on CPU-GPU platforms. We show a fast algorithm which assigns the kernels for execution on a CPU or a GPU at runtime, while taking into account the joint impact of the assignments of all kernels on the entire application performance rather than just the impact of assigning each separately.

We conclude by showing the application of these techniques to the computation of probability of evidence in large probabilistic networks.

3 Algorithms, Implementations, and Evaluations

We now present a "recipe" for designing a kernel with a scratchpad-based software-managed cache. We then apply this recipe to build a software-managed cache for sum-product kernel.

3.1 Software-manged cache recipe

Optimize for locality. As in a CPU implementation, the GPU implementation also requires optimization for spatial locality (for coalesced memory accesses when fetching data to the cache), and temporal locality (for the working set reduction) of memory accesses.

Divide into thread blocks with regular memory accesses and high reuse. The number of threads in the thread block may be dictated by the need to minimize the size of metadata tables used by the caching mechanisms. For example, if every third thread reuses the data of the first one, the number of threads in a thread block should be a multiple of three. Then, the access pattern would be the same for all thread blocks and can be computed only once. Internal reuse is important since the cache is private to a single thread block. For example, in the matrix product kernel, assigning threads of the same thread block to compute the entire output row (instead of a block) is suboptimal since the data in the columns are not reused within the thread block.

Note that the first two "ingredients" above are also important for making optimal use of a hardware cache.

Define cache page, determine the cache replacement policy and granularity. Input blocks used concurrently by all the threads in a thread block must reside in the cache *at the same time*. We will call such a resident set a *cache page*. The policy determines when to switch to a new cache page, which part of the cache page is to be replaced, and which part of the reused data should remain in the main memory without being cached at all. The granularity of the replacement decisions is critical to cache performance. A fine-grained replacement policy might improve the cache hit rate, but would incur higher overheads at runtime.

We emphasize that organizing the computations so that the accesses are localized in a cache page is useful when using a hardware cache too. However the size of the cache page as well as the specific access pattern within the page must be adjusted to the hardware replacement policy in order to avoid cache thrashing. Furthermore, the same L1 cache is shared by multiple concurrently running thread blocks, as opposed to the disjoint spaces per thread block for the software-managed cache. This makes the hardware cache performance dependent on the interplay between the access pattern of different thread blocks and the cache policy, whereas the software-managed cache is immune to this problem.

Determine the cache address scheme. The data is located in the cache in different physical addresses than its global memory locations. Mapping between the old and the new location is required. Computing that mapping may be quite expensive as the address depends on the offset of the data in the cache and the cache policy, thus necessitating access to multiple cache policy lookup tables. Hence it may be beneficial to precompute it on a CPU as well. Fortunately, once constructed for a single thread block, the same mapping may be valid for all other thread blocks, thanks to access regularity.

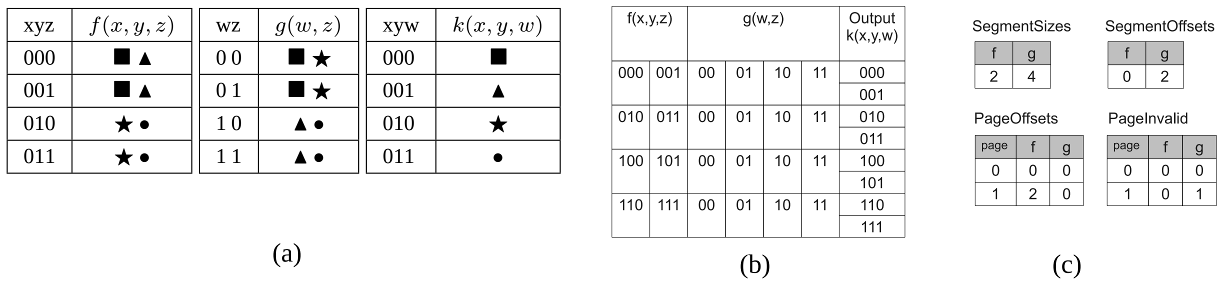
3.2 Software-managed caching for sum-product

Here we demonstrate the application of this "cache recipe" to the sum-product kernel, which forms the core of the inference computations in probabilistic networks. In general, sum-product computations arise in a wide variety of scientific applications, such as artificial intelligence, bioinformatics, statistics, image processing and digital communications. (See, e.g., Pakzad and Anantharam [4] for a comprehensive overview of sum-product.)

Consider the following expression:

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

This equation describes a function $\psi(x)$, which is computed by performing a series of tensor products followed by a summation. We skip the formal explanation here and focus on the access pattern of these computations as shown in the example below.



xyz	$f(x,y,z)$
000	■ ▲
001	■ ▲
010	★ ●
011	★ ●

wz	$g(w,z)$
0 0	■ ★
0 1	■ ★
1 0	▲ ●
1 1	▲ ●

xyw	$k(x,y,w)$
000	■
001	▲
010	★
011	●

(a)

f(x,y,z)		g(w,z)				Output k(x,y,w)
000	001	00	01	10	11	000
						001
010	011	00	01	10	11	010
						011
100	101	00	01	10	11	100
						101
110	111	00	01	10	11	110
						111

(b)

SegmentSizes

f	g
2	4

SegmentOffsets

f	g
0	2

PageOffsets

page	f	g
0	0	0
1	2	0

PageInvalid

page	f	g
0	0	0
1	0	1

(c)

Figure 2: Access pattern and cache structures for computing $k(x,y,w) = \sum_z f(x,y,z) \otimes g(w,z)$. (a) Input and output accesses for computing $k_{000}, k_{001}, k_{010}, k_{011}$. The symbols in the diagram represent which locations in the tables are accessed when computing the outputs with the same symbol. For example: to compute k_{000} (■) one reads $f_{000}, f_{001}, g_{000}$ and g_{001}, i.e. the input values which are also marked by ■. (b) Layout of the cache pages, one cache page per row. (c) Content of the cache policy tables assuming two cache pages accessed by a thread block, two threads per thread block.

Understanding the access pattern The individual functions $f(x,y,z)$, $g(x,w)$, and $h(y,z,w)$ in this application can be thought of as similar to multidimensional array accesses in C. For example, in the function $f(x,y,z)$ (with $x \in X, y \in Y, z \in Z$), the value $f_{x,y,z}$ is located in the memory at the offset $z + |Z| \times y + |Y| \times |Z| \times x$.

Figure 2(a) illustrates the memory accesses for computing $k_{000}, k_{001}, k_{010}, k_{011}$ in $k(x,y,w) = \sum_z f(x,y,z) \otimes g(w,z)$ for the case where X, Y, Z, and W are all of size 2. Observe that $g(w,z)$ is accessed in exactly the same manner when computing $k(x=0, \mathbf{y=0}, w=0)$ and $k(x=0, \mathbf{y=1}, w=0)$, whereas different locations of f are accessed for the same output, exhibiting no reuse at all.

As Figure 2(a) shows, the input data are reused and the access pattern is periodic. Furthermore, the data are accessed in segments since the summation variables are always grouped and iterated together (this is, in fact, a result of optimizing the locality of accesses as described in our previous work [5]). However we also see that the reuse pattern differs for each input function and depends on the specific variables in the function's scope.

Dividing into thread blocks. Each output location can be computed independently in a separate thread, but each thread is assigned multiple output locations to improve data reuse, as will be explained later.

The size of the thread block cannot be chosen arbitrarily. In the example in Figure 2(a), we see that the access pattern is correlated with the domain size of the function variables: for any group of power-of-two outputs the data used have the same offsets, but different base address. Here, computing the pairs of output k_{000}, k_{001} and k_{010}, k_{011} requires accessing two adjacent entries in f, starting from offset 0 for the first pair and offset 2 for the second one. So for the access pattern in different thread blocks to be the same, the set of outputs for every block should be aligned to a multiple of power-of-two. Hence, if we let one thread compute one output, the thread block size can be 2,4 or 8. For the case of 4, the first thread block would compute k_{000}, k_{001} ,k_{010}, and k_{011} [2].

Cache page. Each cache page is split into multiple *segments*, one per input function. The size of each segment depends on the amount of data accessed in each function by all the threads of one thread block. For example, each row in the table in Figure 2(b) represents one cache page (assuming one output per thread, two threads per block). The sizes of the segments as well as the total cache size are computed on a CPU for each set of input functions, as will be explained below. The results are stored as arrays in the GPU constant memory [3] (*SegmentSizes* and *SegmentOffsets* arrays in Figure 2(c)).

[2]These sizes are used only for the purpose of this example, but in practice much larger thread blocks should be used

[3]See the CUDA C Programming Guide for more details on the constant memory

Cache replacement policy. The amount of data accessed concurrently by all the threads, or, in other words, the size of one cache page, may exceed the available scratchpad memory size. Reducing this size by decreasing the number of threads per thread block may reduce the hardware utilization. Partial caching of the data, whereby one part of the function data can remain uncached, would require that an additional policy table be checked upon *every memory access* in order to determine whether the specific location is cached.

Our solution balances cache management overhead and cache hit rate. We mark some functions as not cached. These functions are accessed directly from the global memory, bypassing the cache. The choice of which functions to cache is precomputed on a CPU as follows: we add the functions in the order of the sizes of their cache segments, starting from the smallest and continuing for as long as cache space permits. Other replacement algorithms may of course be better suited for other computations.

Which data to fetch when switching to another cache page is also determined by the replacement policy. As mentioned above, multiple cache pages are accessed by the same thread block to exploit data reuse between the pages.(In Figure 2(b), the same data of function g is used in all four cache pages, which can be leveraged if all four pages are processed by the same thread block.) Upon switching to the next page, only the data of the functions which differ from the previous cache page are fetched. In the example above, if all four cache pages are processed by the same thread block, only the data of f would be replaced when moving to the next page.

A CPU is used to determine which functions are to be replaced when moving from one page to another; this information is used by the kernel each time the cache page is switched.

Examples of all the cache policy tables are presented in Figure 2(c). These policy tables correspond to the setup with two threads per thread block, two outputs per thread (hence, two pages per thread block). *SegmentSizes* and *SegmentOffsets* are used to determine how much data to fetch per cache page and where each function is placed in the cache. *PageOffsets* is used to determine the offset to the respective input function for each cache page. The *PageInvalid* table determines which function should be replaced when the page is switched. In this example both functions are to be fetched when the first cache page is accessed, but only f has to be fetched again. The data corresponding to the second page of this thread block is to be read from the global memory starting from offset 2.

Cache address scheme. Each thread determines the data to be accessed by computing the offset using the local thread index and the cache segment offset for the function being accessed. For example, in Figure 2(b) the thread with index 1, which computes k_{001}, will access two values from each input function (for $z = 0$ and $z = 1$), with offsets 0 and 2 respectively. Since all the summation values are accessed sequentially, computing the offset is necessary only once per summation loop (provided proper loop unrolling).

3.2.1 Kernel for computing sum-product

The kernel that computes sum-product is presented in Figure 3. The identifiers starting with capital letters denote the data structures precomputed on the CPU, and the underlined names denote the cache policy tables in the constant memory. The remaining precomputed data are placed in the texture memory [4].

The kernel can be logically split into four parts: determining the input blocks to be processed by a given thread block (lines 3–8), cache prefetching via the *cachePrefetchPage* procedure (lines 26 – 41), computation loop (10–23), and writing back the result (23).

A few important points should be emphasized. First, despite the many conditional statements in the kernel, there is no divergence between the threads in a warp. This is because the outcome of the statement is independent of the thread identity and thus is the same for all threads in the warp. Second, all the threads in a warp always access the same location in the cache policy tables, which is ideal for the constant memory

[4]See the CUDA C Programming Guide for more details on the texture memory

```
Function SumProductKernel
Input: Input functions
outputPtr ← call computeOutputPtr(blockIdx)
for all input functions $f$ do
    inputPtrs[$f$] ← call computeInputPtrs(blockIdx, $f$)
end for
for all cache pages page do
    call cachePrefetchPage(page,inputPtrs)
    sum ← 0, counter ← 0
    for all summation values do
        product ← 1
        for all input functions $f$ do
            if SegmentSizes[$f$] $> 0$ then
                value ← call cacheFetch(ThreadOffsets[threadIdx][f] + counter)
            else
                offset ← inputPtrs[$f$] + PageOffsets[page][$f$] + ThreadOffsets[threadIdx][f] + counter
                value ← call memoryFetch(offset)
            end if
            product ← product × value
        end for
        sum ← sum + product, counter ← counter + 1
    end for
    outputPtr[page × ThreadBlockSize + threadIdx] ← sum
end for

Function cachePrefetchPage
Input: page number to fetch page, pointers to the the thread block inputs inputPtrs
for all input functions $f$ do
    if PageValid[page][$f$] is false AND SegmentSizes[f] $> 0$ then
        call __syncthreads()
        call parallelCopy(SegmentOffsets[f], inputPtrs[f]+ PageOffsets[page][f], SegmentSizes[f])
        call __syncthreads()
    end if
end for

Function parallelCopy
Input: cache offset $cOffset$, global memory pointer $gMem$, words to copy $size$
for $i$ = threadIdx to $size$ do
    $CACHE[cOffset + i] \leftarrow gMem[i]$
    $i \leftarrow i$ + ThreadBlockSize
end for
```

Figure 3: GPU kernel pseudocode

cache. Similarly, the data structures residing in the texture memory (e.g. *PageOffsets*) are small, and fit the small texture cache well. Finally, this procedure can be heavily unrolled, which allows for the overhead of accesses to the policy tables to be amortized over several iterations, reducing it even further.

The full GPU kernel code together with the CPU preparation procedures is available for download [1].

3.3 Algorithm for task tree scheduling

Here we present a fast algorithm for scheduling task dependency trees by assigning the tasks for execution to a CPU or a GPU.

The problem of task dependency graph scheduling on heterogeneous architectures has drawn a lot of attention (see, for example, an overview of the DAG scheduling [3]). All these algorithms try to decrease the running time by keeping all the processors busy, scheduling different graph nodes to different processors in parallel. Finding an optimal schedule that accounts for bandwidth constraints and processor heterogeneity is hard even for task trees (graphs without undirected cycles).

Our approach is different. We target an *acceleration schedule*, for the programming model where a GPU is considered *a co-processor*. In such an asymmetric setup, a GPU cannot operate on its own; the CPU must dedicate some of its time to GPU management. Hence the algorithm optimizes the runtime for the case where the CPU or GPU do not concurrently execute tasks. Indeed, an acceleration schedule may not be an optimal parallel schedule in cases other than the chain dependency graph. That is because the algorithm does not exploit the parallelism available in the graph itself: while one branch is processed on a CPU another could be processed on a GPU. Yet the acceleration model is very popular among GPU developers: it allows for a simple and easily implementable algorithm, while still enabling performance improvement over only-CPU or only-GPU implementations, and over the greedy algorithm combining both.

Because the assignment of a task in one tree branch does not influence the assignment of a task in another branch, we can apply a dynamic programming approach.

The input to the algorithm is a task dependency tree $T(V, E)$ with the nodes V and edges E. Every node $v \in V$ has the following attributes:

1. Kernel performance vector P_v with two entries for the expected kernel execution time for the respective task v on a CPU or a GPU respectively.
2. Transfer time matrix D_v with four entries for the time required to transfer the kernel input for all the combinations of source and destination: GPU$\rightarrow$ CPU, CPU$\rightarrow$GPU, GPU$\rightarrow$GPU, CPU$\rightarrow$CPU. Clearly, $D_v[CPU \rightarrow CPU] = D_v[GPU \rightarrow GPU] = 0$, as long as the source and destination are the same GPU.

For every node $v \in V$, the algorithm maintains the following variables:

1. Subtree processing time vector S_v of the subtree rooted at v, with two entries $S_v[CPU]$ and $S_v[GPU]$, each for the best processing time of that subtree assuming v is executed on a CPU or a GPU respectively.
2. Subtree scheduling decision vector O_v, containing the task assignment $O_v^d[CPU]$ and $O_v^d[GPU]$ for every immediate descendant (child) d of v corresponding to $S_v[CPU]$ and $S_v[GPU]$. This variable stores d's assignment which resulted in the best total elapsed time including the memory transfer from d to v were d executed on a CPU or a GPU. It is used in the backtracking step.
3. Scheduling decision A_v regarding where to execute the node.

The acceleration schedule algorithm presented in Figure 4 runs in two steps: in the forward traversal it traverses the tree from the leaves to the root. For each node v it computes the cost for all possible assignments of node v given the cost of computing its child nodes on a CPU and a GPU and the respective data transfer times (lines(11-21). When this step completes, every node holds the best costs of computing its subtree for both its schedules on a CPU or a GPU. The backtracking step then traverses the tree in the prefix DFS order from the root and determines the assignment for all the nodes, using the optimal scheduling decision for their respective parents and generating an optimal acceleration schedule.

3.3.1 Transfer and execution time predictions

The algorithm requires knowledge of the expected running time of a given task on a CPU and a GPU, and the times of the input data transfers. The latter is easy to estimate using the hardware bandwidth and the input data size known before the run.

The kernel time prediction is more complicated, and several approaches exist. The first approximation is to assume a constant device capacity and derive the running time from the total number of computations to be performed. More precise methods apply various machine learning techniques to predict the performance by using the profiles of the previous kernel invocations. In our work we applied a regression tree classifier which allowed runtime predictions to be derived from the traces of the kernel microbenchmarks.

```
Input: $T(V,E)$ - Task dependency tree, $R$ - postfix DFS traversal order of $T$
Output: Scheduling decisions $A_v$ for all the nodes $v \in V$.
                    Forward traversal
while $R$ is not empty do
    //get next tree node
    $v \leftarrow$ pop($R$)
    push $v \rightarrow \hat{R}$ // maintain prefix DFS order
    for all $device \in CPU, GPU$ do
        // set the cost of $v$ on $device$
        $S_v[device] \leftarrow P_v[device]$
        // compute the costs assuming $d$ is executed on a CPU (GPU) and $v$ on $device$
        for all $d \in$ child nodes of $v$ do
            $CPUCOST \leftarrow S_d[CPU] + D_v[CPU \rightarrow device]$
            $GPUCOST \leftarrow S_d[GPU] + D_v[GPU \rightarrow device]$
            // choose the best schedule for $d$ assuming $v$ is executed on $device$
            if $CPUCOST > GPUCOST$ then
                $O_v^d[device] \leftarrow$ GPU
                $S_v[device] \leftarrow S_v[device] + GPUCOST$
            else
                $O_v^d[device] \leftarrow$ CPU
                $S_v[device] \leftarrow S_v[device] + CPUCOST$
            end if
        end for
    end for
end while
                    Backtrack
$v \leftarrow$ pop($\hat{R}$)
// choose the device to compute the root node
if $S_v[CPU] > S_v[GPU]$ then
    $A_v \leftarrow$ GPU
else
    $A_v \leftarrow$ CPU
end if
// traverse in prefix DFS order
while $\hat{R}$ is not empty do
    for all $d \in$ child nodes of $v$ do
        // schedule $d$ on the device which led to the best cost for $v$
        $A_d \leftarrow O_v^d[A_v]$
    end for
    $v \leftarrow$ pop($\hat{R}$)
end while
```

Figure 4: Acceleration scheduling algorithm pseudo-code

3.4 Application to inference in probabilistic networks

The general problem is:

$$\sum_{\mathbf{M}} \bigotimes_i f^i(\mathbf{X}^i), \quad \mathbf{M} \subseteq \bigcup_i \mathbf{X}^i, f^i \in \mathbf{F}, \tag{2}$$

where $\mathbf{M}$ is the set of summation variables, and $\mathbf{F}$ is the set of all input functions. One of the methods to compute this expression is by splitting the set of all functions into groups, called buckets, and process each bucket separately using the sum-product kernel described in Section 3.2.1. The algorithm for creating the buckets is outside of the scope of this chapter and can be found elsewhere [2].

For a given set of buckets, computation can be represented as a task dependency tree traversal from the leaves to the root, where each bucket is a tree node and the edges between the nodes represent the data dependencies.

We use the scheduling algorithm in section 3.3 to assign the computations to a CPU or a GPU, and then employ the GPU kernel described in section 3.2.1 to compute the results of the GPU-assigned nodes.

We partially relax the assumption of no concurrency between CPU and GPU execution by implementing CPU execution and GPU management in two different CPU threads. All the nodes scheduled on a CPU are computed by an OpenMP-based parallel CPU kernel that uses multiple CPU cores. The CPU execution thread keeps processing the nodes assigned to a CPU until until the node for a GPU is found. When that happens, the CPU execution thread analyzes the input to tune the kernel invocation parameters and then passes this information to the GPU management thread, which is responsible for transferring the data and invoking the GPU kernel. Meanwhile the CPU thread continues with the processing of the tasks in another task tree branch, until it runs out of CPU-assigned nodes because of data dependency, or there are no GPU-assigned nodes left to be prepared for execution.

4 Final evaluation

We conducted our experiments on both synthetic benchmarks and real probabilistic networks used for analyzing genetic data [6]. We performed three sets of experiments: (1) single kernel performance comparison on random inputs in order to test the software-managed caching on different GPU architectures; (2) the impact of CPU-GPU scheduling on the application performance; (3) complete application speedups over CPU-only execution.

4.1 Software-managed caching performance

Figure 5(a) and (b) presents the kernel performance on two different NVIDIA GPUs (GeForce GTX 285 and Tesla C2050) and the Intel E5540 hyper-threaded dual quad core machine on 5,000 random inputs. The inputs were generated by randomly selecting various input parameters: the number of variables per function, number of functions per input, variable domain size, data reuse per input, output size and number of summation variables per input. Each dot in the graph represents the performance of one run. Here and in the rest of the experiments in this subsection, the running times represent only the actual computing times without the data transfer. The input complexity FLOPs were computed by counting only the theoretical number of double-precision multiplications and summations required to compute the results. The CPU OpenMP implementation used up to 16 threads.

Table 1 summarizes the peak performance results for double-precision execution. Note that the peak speedup may reach **up to a factor of 50** for some inputs: for these inputs, a CPU performs poorly, but a GPU achieves the highest performance.

E5540 1 core	2xE5540 quad core, 16 HW threads	GTX 285	C2050
1.8	15.5	34	84

Table 1: Peak double-precision performance in GFLOPs/s achieved on different processors.

Observe the performance variability which characterizes the kernel: it is due to the internal properties of the computations. More specifically, some of the inputs have more parallelism available and require less cache space, thereby enabling higher performance gains on a GPU and on a CPU alike. That is why the dynamic selection of execution policies is imperative – some inputs clearly perform better on a CPU than on a GPU.

4.1.1 Cache performance comparison

Figure 5(c) presents the relative speedup of the software-managed cache over the version that uses only global memory and the version that uses texture memory for the input on NVIDIA GeForce GTX 285. We see that the hardware texture cache improves the kernel performance but falls short of the performance achieved by the software-managed caching techniques. Overall, the lack of proper hardware cache on this and earlier architectures made software-caching *the only way* to achieve high performance.

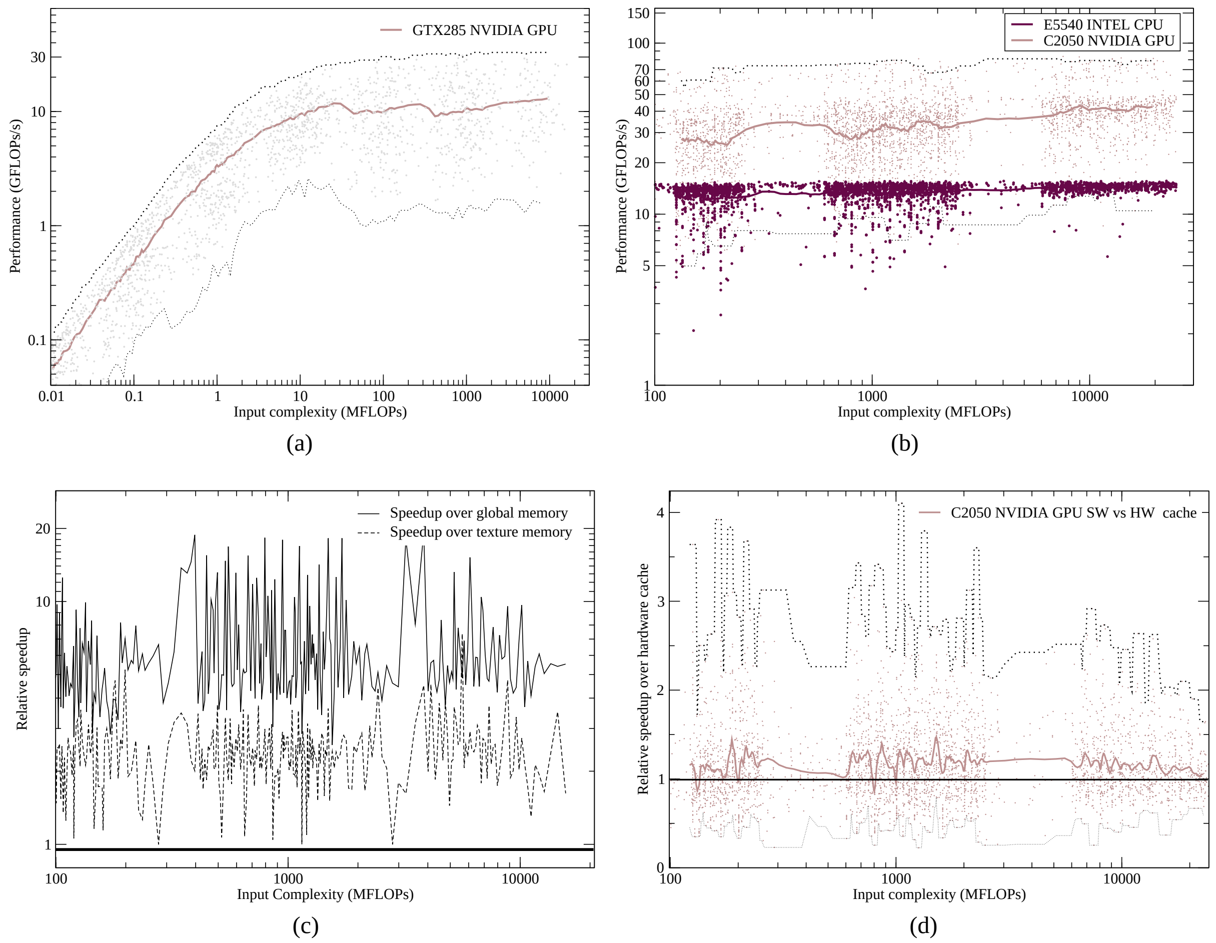


Figure 5: Random-input performance of the double-precision kernel execution. Each dot represents the performance of one run. The dotted lines represent the peak and the lowest performance measured. The continuous line is the average performance. (a) NVIDIA GeForce GTX 285 GPU. (b) NVIDIA Tesla C2050 GPU and OpenMP parallel execution on dual 4-core 8-thread Intel E5540 2.53GHz CPU. (c) Relative speedup of the software-manged cache over pure global memory and texture memory runs in GeForce GTX 285. (d) Relative speedup of the hardware and software-managed cache combined over hardware-only cache in Tesla C2050.

Figure 5(d) shows the hardware/software cache comparison on a Tesla C2050 GPU featuring a hardware cache. The experiments with the software-managed cache were performed using a 48K/16K scratchpad/L1 partition; therefore, the figure reflects the combined performance of the hardware cache and the software-managed cache.

To test the hardware cache performance we modified the original kernel by removing all the cache-related logic, including the thread synchronizations, and configured a 16K/48K scratchpad/L1 partition. We see that the software-managed cache performance is not stable and can be both worse and better than the hardware-only cache, with the speedup being about 25% on average. Despite the low average-case speedup, there are two important observations that justify application of software caching. First, the software cache enables much higher peak performance. Only 0.4% of the hardware cache-based runs exceeded the 60 GFLOPs/s performance threshold, and none reached 70 GFLOP/s, versus 5.5% and 2% of the software-managed cache kernel runs, respectively. Second, for a given input, the best of both worlds can be achieved by analyzing the input properties and selecting the expected-to-be-optimal kernel configuration on the fly. Specifically, we observed that the hardware cache usually performs better for inputs with fewer summation variables, where the software-managed cache access overhead cannot be amortized over multiple memory accesses. Thus,

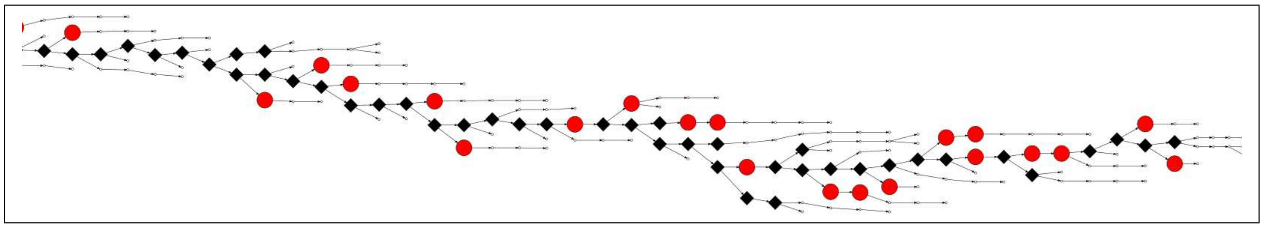


Figure 6: Part of the 268-node task tree with the hybrid-greedy and hybrid communication-aware schedule. Nodes marked with diamonds denote the tasks assigned to a GPU by both schedules. Circles denote only the tasks assigned to a GPU according to the communication-aware schedule. All unmarked nodes are assigned to a CPU.

Tasks in tree	Runtime (seconds)				Nodes mapped to GPU	
	Hybrid communication-aware	Hybrid-greedy	CPU-only	GPU-only	Hybrid communication-aware	Hybrid-greedy
390	**110**	126	213	211	25	14
529	**86**	86	119	174	41	28
268	**55**	55	408	67	86	62
595	**21**	25	174	33	139	111
1194	**35**	44	364	60	301	230
505	**126**	140	494	250	46	21

Table 2: Comparative performance analysis of the dynamic schedule for several real genetic analysis inputs. The optimal acceleration schedule is the one produced by the algorithm described here. The hybrid-greedy schedule considers only single kernel performance.

we dynamically select the best expected kernel configuration as a part of the complete application. As we will show in the following sections, the use of software caching was critical for attaining high performance exceeding that of the CPU implementation.

4.2 Influence of CPU-GPU scheduling

Another set of experiments examined the impact of the CPU-GPU schedule on the performance of the entire multi-kernel application. The experiments used task dependency trees from probabilistic networks for genetic linkage analysis, and were invoked on a 4-core Intel Core 2, 2.33GHz CPU with the NVIDIA GeForce GTX 285 GPU. The performance results for a few representative inputs are shown in Table 2.

We observe that the best CPU-only multi-threaded version or GPU-only version can be each up to a factor of two slower than the combined CPU-GPU execution using the hybrid communication-aware schedule produced by our algorithm. Observe

that this schedule (the column marked in bold) usually results in more nodes being mapped to a GPU. Figure 6 shows a part of the task tree, with the diamonds denoting the nodes scheduled on a GPU by both the hybrid-greedy and hybrid communication-aware schedules and the circles denoting those scheduled by the communication-aware schedule only. Observe that the latter effectively reschedules the “islands” of CPU-scheduled nodes to a GPU.

We found, however, that for the task trees having a set of dominating complex tasks for which GPU performance substantially exceeds CPU performance and the I/O to CPU ratio is low, mapping kernels with larger input sizes is sufficient for achieving the best performance. Still, even then, the dynamic schedule that combines CPU and GPU execution remains superior to the static schedules that use only one or the other.

4.3 Performance on large probabilistic networks

The last set of experiments evaluated the speedups of the entire application with all the optimizations. The experiments were carried out on real-life probabilistic networks used for the analysis of genetic diseases. Table 3 summarizes the results of execution on 11 different networks. The table shows speedup over the

Network	E5540 4 core	E5540 1 core	2xE5540 8 cores	GTX 285	C2050 (h/w cache)	C2050 (s/w cache)
BN1	97s	0.26	1.9	2.3	1.1	3.8
BN2	27s	0.29	2.0	–	1.9	2.28
BN3	3s	0.37	1.8	1.3	1.4	1.3
BN4	21s	0.25	2.0	0.5	1.4	1.8
BN5	844s	0.25	1.5	–	1.7	3.3
BN6	66s	0.26	1.3	–	3.1	6.0
BN7	316s	0.31	2.1	–	1.7	5.4
BN8	74s	0.25	1.9	2.9	2.4	5.4
BN9	17s	0.25	1.8	–	1.6	3.3
BN10	72s	0.25	2.1	–	3.9	6.5
BN11	91s	0.25	2.1	–	1.9	4.3

Table 3: Complete application performance on large inputs. The first column shows the absolute runtime in seconds, while the rest show the relative speedup over the quad-core execution. Some inputs could not be computed on GeForce GTX 285 due to insufficient memory.

quad-core only execution on a single chip E5540 CPU. Using the single-chip performance as the baseline allows for a chip-to-chip comparison with GPUs. We also provided the results of the dual-CPU performance with 16 concurrent hardware threads; hence the speedup may exceed 2.

The kernel that uses the software-managed cache outperforms the parallel single CPU version by up to a factor of 6.5. Furthermore, using the hardware-only cache results in an up to three-fold performance drop, often yielding slower execution than the eight-core parallel CPU version. Observe that the software-managed cache is faster than the hardware cache in all but one case. This is the result of the automatic selection of the purely-hardware and combined hardware-software cache configurations, depending on the specific input, as described above.

References

[1] Sum-product GPU kernel. `http://sites.google.com/site/silbersteinmark/Home`.

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

[3] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *J. Parallel Distrib. Comput.*, 59(3):381–422, 1999.

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

[5] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on GPUs through software-managed cache. In *22nd ACM International Conference on Supercomputing*, pages 309–318, June 2008.

[6] M. Silberstein, A. Tzemach, N. Dovgolevskiy, M. Fishelson, A. Schuster, and D. Geiger. On-line system for faster linkage analysis via parallel execution on thousands of personal computers. *American Journal of Human Genetics*, 78(6):922–935, 2006.