The Hierarchical Memory Machine Model for GPUs

Koji Nakano
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan
Email: nakano@cs.hiroshima-u.ac.jp

***Abstract*—The Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM) are theoretical parallel computing models that capture the essence of the shared memory access and the global memory access of GPUs. The main contribution of this paper is to introduce the Hierarchical Memory Machine (HMM), which consists of multiple DMMs and a single UMM. The HMM is a more practical parallel computing model which reflects the architecture of current GPUs. We present several fundamental algorithms on the HMM. First, we show that the sum of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l + \log n)$ time units using p threads on the HMM with width w and latency l, and prove that this computing time is optimal. We also show that the direct convolution of m and $m + n - 1$ numbers can be done in $O(\frac{n}{w} + \frac{mn}{dw} + \frac{nl}{p} + l + \log m)$ time units using p threads on the HMM with d DMMs, width w, and latency l. Finally, we prove that our implementation of the direct convolution is time optimal.**

***Keywords*-parallel computing models, memory machine models, convolution, GPU, CUDA**

I. Introduction

A. Background

The GPU (Graphics Processing Unit), is a specialized circuit designed to accelerate computation for building and manipulating images [1], [2], [3]. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers [1], [4], [5]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [6], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [7], since they have hundreds of processor cores and very high memory bandwidth.

CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [6]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-64 Kbytes. The global memory is implemented as an off-chip DRAM, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the bank conflict* of the shared memory access and *the coalescing* of the global memory access [2], [5], [7], [8]. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory banks at the same time, the access requests are processed in turn. Hence, to maximize the memory access performance, threads of CUDA should access distinct memory banks to avoid the bank conflicts of the memory accesses. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access when they access the global memory.

There are several previously published works that aim to present theoretical practical parallel computing models capturing the essence of parallel computers. Many researchers have been devoted to developing efficient parallel algorithms to find algorithmic techniques on such parallel computing models. For example, processors connected by interconnection networks such as hypercubes, meshes, trees, among others [9], bulk synchronous models [10], LogP models [11], reconfigurable models [12], among others. As far as we know, no sophisticated and simple parallel computing model for GPUs has been presented. Since GPUs are attractive parallel computing devices for many developers, it is challenging work to introduce a theoretical parallel computing model for GPUs.

B. Memory Machine Models

In our previous paper [13], we have introduced two models, *the Discrete Memory Machine (DMM)* and *the Unified Memory Machine (UMM)*, which reflect the essential features of the shared memory and the global memory of NVIDIA GPUs. The outline of the architectures of the DMM and the UMM is illustrated in Figure 1. In both architectures, *a sea of threads (Ts)* are connected to *the memory banks (MBs)* through *the memory management unit (MMU)*. Each thread is a Random Access Machine (RAM) [14], which can execute fundamental operations in a time unit. We do not discuss the architecture to implement the sea of threads, but we can imagine that it consists of a set of multi-core processors which can execute multiple threads in parallel. Threads are executed in SIMD [15] fashion, and the processors run on the same program and work on the different data.

MBs constitute a single address space of the memory. A single address space of the memory is mapped to the MBs in an interleaved way such that the word of data of address i is stored in the $(i \bmod w)$-th bank, where w is the number of MBs. The main difference of the two architectures is the connection of the address line between the MMU and the MBs, which can transfer an address value. In the DMM, the address lines connect the MBs and the MMU separately, while a single address line from the MMU is connected to the MBs in the UMM. Hence, in the UMM, the same address value is broadcast to every MB, and the same address of the MBs can be accessed in each time unit. On the other hand, different addresses of the MBs can be accessed in the DMM. Since the memory access of the UMM is more restricted than that of the DMM, the UMM is less powerful than the DMM.

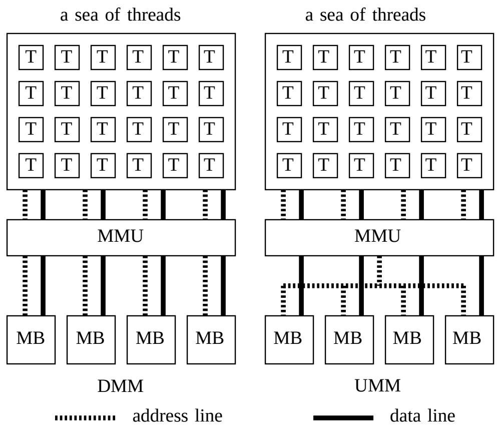

Figure 1. The architectures of the DMM and the UMM

The performance of algorithms of the PRAM is usually evaluated using two parameters: the size n of the input and the number p of processors. For example, it is well known that the sum of n numbers can be computed in $O(\frac{n}{p}+\log n)$ time on the PRAM [16]. We will use four parameters, the size n of the input, the number p of threads, the width w and the latency l of the memory when we evaluate the performance of algorithms on the DMM and on the UMM. The width w is the number of memory banks and the latency l is the number of time units to complete the memory access. These parameters are used when we evaluate the performance of algorithms on the DMM and the UMM. For example, we have shown in [17] that the prefix-sums of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units. In NVIDIA GPUs, the width w of global and shared memory is 32. Also, the latency l of the shared memory is 1 or 2 clock cycles while that of the global memory is several hundred clock cycles. In CUDA, a grid can have at most 65535 blocks with at most 1024 threads each [6].

Since the memory machines are promising as computing models for GPUs, we have published several efficient algorithms on the DMM and the UMM [17], [18], [19]. For example, in our previous paper [13], we have presented offline permutation algorithms on the DMM and the UMM. We also implemented the offline permutation algorithm on the DMM and showed that theoretical analysis of the performance on the DMM provides very good approximation of the CUDA C implementation of offline permutation algorithm [19]. This fact implies that the DMM is a good theoretical model for computation using the shared memory on GPUs.

C. Our Contribution: The Hierarchical Memory Machine Model and Fundamental Parallel Algorithms

The DMM and the UMM are good theoretical model of computation performed by a single streaming multiprocessor (SM) on the GPU. Algorithms on the DMM and the UMM correspond to the computation using the shared memory and the global memory of GPUs, respectively. However, since GPUs have multiple SMs, we need to develop a new parallel computing model that supports multiple SMs on the GPUs. The main contribution of this paper is to introduce the Hierarchical Memory Machine (HMM), which consists of multiple DMMs and a single UMM. The HMM is a more practical parallel computing model that reflects the architecture of GPUs. Figure 2 illustrates the architecture of the HMM. The HMM consists of d DMMs and a single UMM. Each DMM has w memory banks and the UMM also has w memory banks. We call the memory banks of each DMM *the shared memory* and those of the UMM *the global memory*. Each DMM can work independently and can perform the computation using its shared memory. Also, all threads of DMMs work as a single UMM and can access to the global memory. Since the shared memory and the global memory of NVIDIA GPUs have latency 1-2 clock cycles and several hundred clock cycles, we assume that those of the HMM is 1 and l. In other words, we use parameter l to represent the global memory access latency. Hence, the performance of algorithms on the HMM can be evaluated as a function of n (the size of a problem), d (the number of DMMs), p (the total number of threads), w (the width (i.e the number of memory banks) of the global memory and the shared memory), and l (the latency of a memory).

Although current GPUs have many features, the HMM mainly focuses on the memory access. It may be possible to incorporate all other features of GPUs and to introduce a more exact parallel computing model for GPUs. However, if all features of GPUs are incorporated in theoretical parallel models, they will be too complicated and need more parameters. The development of algorithms on such complicated models may have too much non-essential and

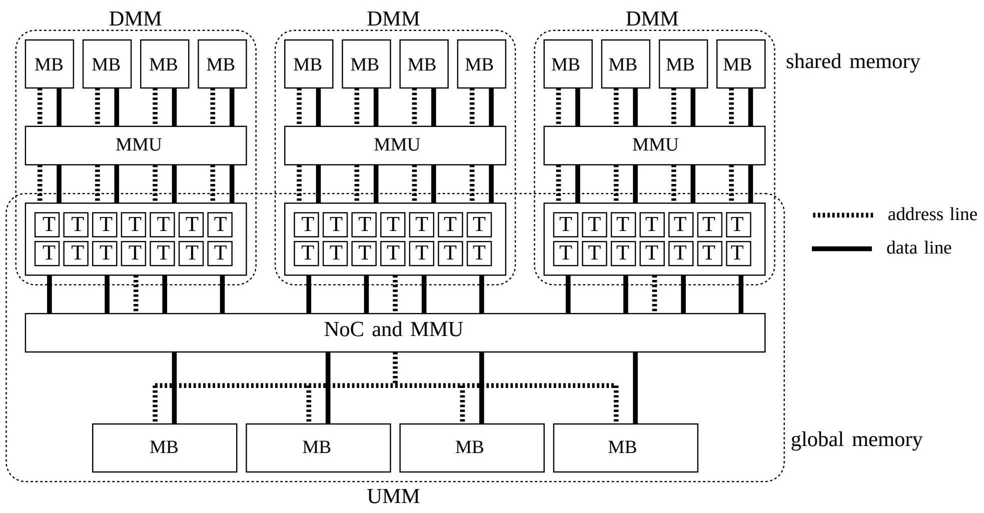

Figure 2. The architecture of Hierarchical Memory Machine Model(HMM)

tedious optimizations. Thus, we focus on just memory access on the current GPUs, and introduce the HMM.

The second contribution of this paper is to evaluate the performance of fundamental algorithms on the HMM and prove their time optimality. We show parallel implementations of two problems, *the sum* and *the convolution*. The sum is a problem to compute the sum $a[0]+a[1]+\cdots+a[n-1]$ for a given array a of size n. In the convolution problem two arrays x of length m and y of length $m+n-1$ are given. It requires to compute an array z of length n such that $z[i]=x[0]\cdot y[i]+x[1]\cdot y[i+1]+\cdots+x[m-1]\cdot y[i+m-1]$ for all i $(0\leq i\leq n-1)$. From the practical point of view, we assume that $m\ll n$. Clearly, naive sequential algorithms can compute the sum in $O(n)$ time and the convolution in $O(mn)$ time. Although an FFT-based convolution algorithm can compute the convolution in $o(mn)$ time, we focus on the direct convolution algorithm that compute each $z[i]$ independently, for the purpose of clarifying the computing power of the HMM.

It is well known that the sum of n numbers can be computed in $O(\frac{n}{p}+\log n)$ time using p processors on the PRAM [16]. Also, using this algorithm, it is not difficult to show that the direct convolution of m and $m+n-1$ numbers can be done in $O(\frac{mn}{p}+\log m)$ time using p processors on the PRAM. We can see the time optimality of these algorithms. Clearly, the sum involves $n-1$ additions. Since p processors can perform p additions in a time unit, $\Omega(\frac{n}{p})$ time is necessary to compute the sum using p processors. Also, any rooted binary tree of n leaves has a path from the root to a leaf with at least $\log n$ internal nodes. Since the computation of the sum is represented by a rooted binary tree of n leaves, it takes at least $\Omega(\log n)$ time to compute the sum on the PRAM. Thus, the computation of the sum has two lower bounds: $\Omega(\frac{n}{p})$-time *speed-up limitation*, and $\Omega(\log n)$-time *reduction limitation*. Since the direct convolution involves mn multiplications and p processors can perform p multiplications in a time unit, it has $\Omega(\frac{mn}{p})$-time speed-up limitation. Also, the computation of each $z[i]$ needs $m-1$ additions and thus, it has $\Omega(\log m)$-time reduction limitation.

Our second contribution is to show that the sum can be computed in $O(\frac{n}{w}+\frac{nl}{p}+l+\log n)$ time units on the HMM. We also show that the direct convolution can be done in $O(\frac{n}{w}+\frac{mn}{dw}+\frac{nl}{p}+l+\log m)$ time units on the HMM. Although optimal parallel algorithms for the sum and the direct convolution on the PRAM are trivial, those for the HMM are complicated and not trivial. We also prove that these implementations are time optimal. More specifically, the sum computation on the HMM has $\Omega(\frac{n}{w})$-time bandwidth limitation, $\Omega(\frac{nl}{p}+l)$-time latency limitation, and $\Omega(\log m)$-time reduction limitation. The direct convolution has $\Omega(\frac{n}{w})$-time bandwidth limitation, $\Omega(\frac{mn}{dw})$-time speed-up limitation, $\Omega(\frac{nl}{p}+l)$-time latency limitation, and $\Omega(\log m)$-time reduction limitation. Table I summarizes the computing time of the sum and the direct convolution on each models.

Table II summarizes the lower bound of the computing time for the sum and the direct convolution on each model. Each lower bound is the sum of the speed-up limitation, the bandwidth limitation, the latency limitation, and the bandwidth limitation. From this table, we can confirm that the computing time shown in Table I is time optimal.

This paper is organized as follows. In Section II, we review the DMM and the UMM presented in our previous papers [13], [20], that capture the essence of the shared memory access and the global memory access of GPUs. Section III introduces the HMM, which reflects the architecture

Table I
THE COMPUTING TIME OF THE SUM AND THE DIRECT CONVOLUTION ON EACH MODEL

	Sequential	PRAM	DMM and UMM	HMM
Sum	$O(n)$	$O(\frac{n}{p} + \log n)$	$O(\frac{n}{w} + \frac{nl}{p} + l \log n)$	$O(\frac{n}{w} + \frac{nl}{p} + l + \log n)$
Direct convolution	$O(mn)$	$O(\frac{mn}{p} + \log m)$	$O(\frac{mn}{w} + \frac{mnl}{p} + l \log m)$	$O(\frac{n}{w} + \frac{mn}{dw} + \frac{nl}{p} + l + \log m)$

p: #processors or #threads, w: width, l: latency, d: #DMMs

Table II
THE LOWER BOUND OF THE COMPUTING TIME FOR THE SUM AND THE DIRECT CONVOLUTION ON EACH MODEL

		PRAM	DMM and UMM	HMM
Sum	Speed-up limitation	$\Omega(\frac{n}{p})$	$\Omega(\frac{n}{p})$	$\Omega(\frac{n}{p})$
	Bandwidth limitation	$\Omega(\frac{n}{p})$	$\Omega(\frac{n}{w})$	$\Omega(\frac{n}{w})$
	Latency limitation	-	$\Omega(\frac{nl}{p} + l)$	$\Omega(\frac{nl}{p} + l)$
	Reduction limitation	$\Omega(\log n)$	$\Omega(l \log n)$	$\Omega(\log n)$
Direct convolution	Speed-up limitation	$\Omega(\frac{mn}{p})$	$\Omega(\frac{mn}{w})$	$\Omega(\frac{mn}{dw})$
	Bandwidth limitation	$\Omega(\frac{n}{p})$	$\Omega(\frac{n}{w})$	$\Omega(\frac{n}{w})$
	Latency limitation	-	$\Omega(\frac{mnl}{p} + l)$	$\Omega(\frac{nl}{p} + l)$
	Reduction limitation	$\Omega(\log m)$	$\Omega(l \log m)$	$\Omega(\log m)$

of current GPUs. In Section IV, we evaluate the performance of contiguous memory access on the memory machine models, which is a key ingredients of parallel algorithms on the memory machine models. Section V reviews sequential algorithms and PRAM algorithms for the sum and the direct convolution. In Sections VI and VII, we present parallel summing algorithms on the DMM/UMM, and the HMM, respectively. They also prove the time optimality of parallel summing algorithms. Sections VIII and IX show parallel convolution algorithms on the DMM/UMM, and the HMM, respectively and prove their optimality. Section X concludes our work.

II. PARALLEL MEMORY MACHINES: DMM AND UMM

The main purpose of this section is to define the Discrete Memory Machine (DMM) and the Unified Memory Machine (UMM) introduced in our previous paper [13]. The reader should refer to [13], [17] for the details of the DMM and the UMM.

We first define *the Discrete Memory Machine (DMM)* of width w and latency l. Let $m[i]$ ($i \geq 0$) denote a memory cell of address i in the memory. Let $B[j] = \{m[j], m[j+w], m[j+2w], m[j+3w], \ldots\}$ ($0 \leq j \leq w-1$) denote *the j-th bank* of the memory. Clearly, a memory cell $m[i]$ is in the $(i \bmod w)$-th memory bank. We assume that memory cells in different banks can be accessed in a time unit, but no two memory cells in the same bank can be accessed in a time unit. Also, we assume that l time units are necessary to complete an access request and continuous requests are processed in a pipeline fashion through the MMU. Thus, it takes $k + l - 1$ time units to complete memory access requests to k memory cells in a particular bank. However, we assume that multiple memory access requests destined for the same address in the same bank have no extra overhead. For example, if two or more threads read from the same address, it can be read at the same time. Also, if two or more threads write in the same address, one of them is arbitrary selected and succeeds in writing.

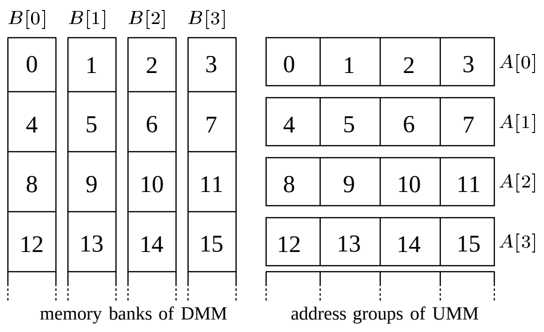

Figure 3. Banks and address groups for $w = 4$

We assume that p threads are partitioned into $\frac{p}{w}$ groups of w threads called *warps*. More specifically, p threads $T(0)$, $T(1)$, $\ldots$, $T(p-1)$ are partitioned into $\frac{p}{w}$ warps $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w + 1), \ldots, T((i+1) \cdot w - 1)\}$ ($0 \leq i \leq \frac{p}{w} - 1$). Warps are dispatched for memory access in turn, and w threads in a warp try to access the memory at the same time. In other words, $W(0), W(1), \ldots, W(\frac{p}{w}-1)$ are dispatched in a round-robin manner if at least one thread in a warp requests memory access. If no thread in a warp needs memory access, such warp is not dispatched for memory access. When $W(i)$ is dispatched, w threads in $W(i)$ send memory access requests, at most one request per thread, to the memory. We also assume that a thread cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread sends a memory access request, it must wait at least l time units to send a new memory access request.

We next define *the Unified Memory Machine* (*UMM* for short) of width w as follows. Let $A[j] = \{m[j \cdot w], m[j \cdot w + 1], \ldots, m[(j+1) \cdot w - 1]\}$ denote the j-th address group. We assume that memory cells in the same address group are processed at the same time. However, if they are in the different groups, one time unit is necessary for each of the groups. Also, similarly to the DMM, p threads are partitioned into warps and each warp accesses the memory in turn.

III. The Hierarchical Memory Machine Model (HMM)

This section is devoted to present the Hierarchical Memory Machine Model (HMM), a more realistic parallel machine model that capture the architecture of GPUs.

The HMM consists of d DMMs and a single UMM as illustrated in Figure 2. Each DMM has w memory banks and the UMM also has w memory banks. We call the memory banks of each DMM *the shared memory* and those of the UMM *the global memory*.

Each DMM works independently. Threads are partitioned into warps of w threads, and each warp are dispatched for the memory access in turn. We also assume that threads can access the global memory of the UMM. A warp of w threads in a DMM can send memory access requests to the global memory. Figure 2 illustrates the architecture of the HMM with $d = 3$ DMMs. Each DMM and the UMM has $w = 4$ memory banks. The shared memory of each DMM and the global memory of the UMM correspond to "the shared memory" of each streaming multiprocessor and "the global memory" of GPUs. Since "the shared memory" of existing GPUs has latency 1 or 2 clock cycles and [6], it is reasonable to assume that the memory access latency of the shared memory of the DMM is 1. Also, since the latency of "the global memory" in the GPUs is several hundred clock cycles [6], it makes sense to use parameter l for the global memory access of the HMM.

The global memory can handle memory access requests by warps in turn. The memory access requests by warps are processed by a pipeline fashion. Since memory access latency is l, it takes l time units to complete the memory access if all memory access requests by the warp is in the same address group. If they are separated in the k address groups, the memory access takes $l + k - 1$ time units. Figure 4 illustrates how memory access requests to the global memory with latency $l = 5$ and $w = 4$. In the figure, two warps $W(A)$ and $W(B)$ are trying to access the global memory. These two warps can be in the same DMM or different DMMs. We can consider that we have an imaginary l-stage pipeline with w registers each to store

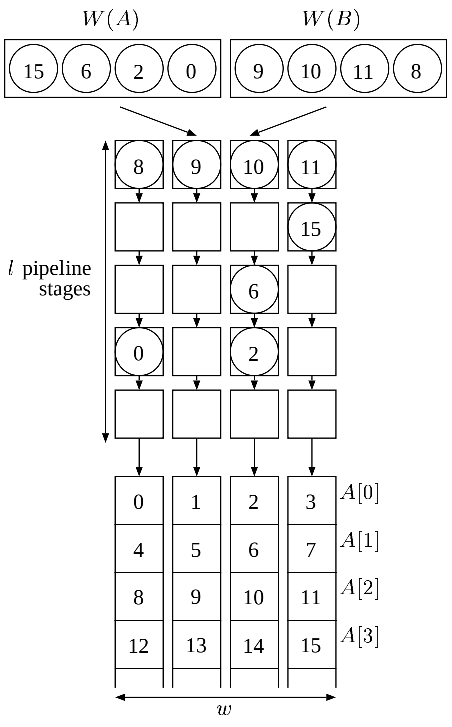

Figure 4. An example of global memory access of the HMM

memory access requests. Note that the l-stage pipeline is imaginary, actual implementation should use NoC (Network-on-Chip) technologies. One of the warps is dispatched for the global memory access and the memory access requests are queued. If the memory access requests by a warp are separated in k address groups, they occupy k pipeline stages. For example, the memory access requests of $W(A)$ are separated in 3 address groups, they use 3 stages. Those of $W(B)$ are in the same address group, they occupy 1 stage. Thus, the memory access requests by $W(A)$ and $W(B)$ in the Figure are completed in $5 + 3 + 1 - 1 = 8$ clock cycles.

Recall that algorithms on the DMM and the UMM are evaluated using four parameters, the size n of the input, the total number p of threads over all DMMs, the width w and the latency l of the memory. In addition, we the number d of DMMs when we evaluate the performance of algorithms on the HMM. For later reference, let $\mathrm{DMM}(0), \mathrm{DMM}(1), \ldots, \mathrm{DMM}(d-1)$ denote d DMM's in the HMM and $T_i(j)$ denote the j-th $(0 \leq j \leq p_i)$ thread of $\mathrm{DMM}(i)$, where p_i is the number of threads running on the $\mathrm{DMM}(i)$. Also, let $p = p_0 + p_1 + \cdots + p_{d-1}$.

For the reader's benefit, we will show how large the values of w, l, d and p of a typical existing GPU are. NVIDIA GeForce GTX580 [21] with CUDA compute capability 2.X [6] has $d = 16$ streaming multiprocessors with 32 cores each, and thus, it has 512 cores totally. Each warp has $w = 32$ threads and each streaming multiprocessor has the shared memory of size up to 48KB arranged in

$w = 32$ memory banks. Each streaming processor can load and run up to 48 (resident) warps with up to $p_i \leq 1536$ threads although it can run more (logical) warps by a time sharing manner. Hence, all streaming processors can run up to 768 warps with $p \leq 24576$ threads. Further, the latency l of the global memory is several hundred clock cycles and the size of the shared memory is up to 48KB while that of the global memory is 2GB. Therefore, GeForce GTX580 corresponds to the HMM with $d = 16$ DMMs with warp $w = 32$. Also, p can be up to 24576. Note that, parameters w, l, and d are fixed values of the HMM. However, each p_i and p are variable such that $p_i \leq 1536$ and $p \leq 24576$. However, since a warp of w threads are executed at the same time in a streaming multiprocessors, it makes sense to assume $p_i \geq w$. Thus, when we use all streaming multiprocessors (i.e. DMMs), it is reasonable to assume that $p = p_0 + p_1 + \cdots + p_{d-1} \geq dw$.

IV. CONTIGUOUS MEMORY ACCESS ON THE DMM AND THE UMM

The main purpose of this section is to review the contiguous memory access on the DMM and the UMM shown in [13], [17].

The contiguous memory access is a key technique for accelerating the computation. Suppose that an array a of size n ($\geq p$) is given. We use p threads to access all of n memory cells in a such that each thread accesses $\frac{n}{p}$ memory cells. Note that "accessing" can be "reading from" or "writing in." Let $a[i]$ ($0 \leq i \leq n-1$) denote the i-th memory cells in a. When $n \geq p$, *the contiguous access* can be performed as follows:

```
[Contiguous memory access]
for $t \leftarrow 0$ to $\frac{n}{p} - 1$ do
  for $i \leftarrow 0$ to $p - 1$ do in parallel
    $T(i)$ access $a[p \cdot t + i]$
```

Let us evaluate the computing time. First, we assume that $w \leq p \leq n$. For each t ($0 \leq t \leq \frac{n}{p} - 1$), p threads access p memory cells $a[pt], a[pt+1], \ldots, a[p(t+1)-1]$. This memory access is performed by $\frac{p}{w}$ warps in turn. More specifically, first, w threads in $W(0)$ access $a[pt], a[pt+1], \ldots, a[pt+w-1]$. After that, p threads in $W(1)$ access $a[pt+w], a[pt+w+1], \ldots, a[pt+2w-1]$, and the same operation is repeatedly performed. In general, p threads in $W(j)$ ($0 \leq j \leq \frac{p}{w} - 1$) access $a[pt+jw], a[pt+jw+1], \ldots, a[pt+(j+1)w-1]$. Since w memory cells accessed by a warp are in different banks, the access can be completed in l time units on the DMM. Also, these w memory cells are in the same address group, and thus, the access can be completed in l time units on the UMM. Recall that the memory access are processed in pipeline fashion such that w threads in each $W(j)$ send w memory access requests in one time unit. Hence, p threads in $\frac{p}{w}$ warps send p memory access requests in $\frac{p}{w}$ time units. After that, the last memory access requests by $W(\frac{p}{w} - 1)$ are completed in $l - 1$ time units. Thus, p threads access p memory cells $a[pt], a[pt+1], \ldots, a[p(t+1)-1]$ in $\frac{p}{w} + l - 1$ time units. Since this memory access is repeated $\frac{n}{p}$ times, the contiguous access can be done in $\frac{n}{p} \cdot (\frac{p}{w} + l - 1) = O(\frac{n}{w} + \frac{nl}{p})$ time units.

Next, let us consider the case that $p \leq w$. If this is the case p threads is in a single warp. This warp performs memory access $\frac{n}{p}$ times each of which takes l time units. Thus, the contiguous access can be done in $\frac{n}{p} \cdot l = O(\frac{nl}{p})$ time.

Finally, let us consider the case that $p > n$. If this is the case n threads out of p threads are used. Since we have $\frac{n}{w}$ warps, the memory access takes $\frac{n}{w} + l - 1$ time units.

Therefore, we have,

Lemma 1: The contiguous memory access to an array of size n can be done in $O(\frac{n}{w} + \frac{nl}{p} + l)$ time using p threads on the DMM and the UMM with width w and latency l.

We can generalize the contiguous memory access such that memory access are performed for several arrays. Let $a_0, a_1, \ldots, a_{k-1}$ be k arrays of size $n_0, n_1, \ldots, n_{k-1}$ each. Suppose that p threads access each of $a_0, a_1, \ldots, a_{k-1}$ in turn. The readers should not have difficulty to confirm that if $k \leq \frac{n}{w}$, the contiguous memory access to all k arrays can be done in $O(\frac{n}{w} + \frac{nl}{p} + l)$ time units. Thus, we have,

Theorem 2: The contiguous memory access to at most $\frac{n}{w}$ arrays of total size n can be done in $O(\frac{n}{w} + \frac{nl}{p} + l)$ time using p threads on the DMM and the UMM with width w and latency l.

V. SEQUENTIAL AND PARALLEL ALGORITHMS FOR THE SUM AND THE DIRECT CONVOLUTION

The main purpose of this section is to define the sum and the direct convolution and to show optimal sequential and parallel algorithms.

Let a be an array of size n. The sum problem is a problem to compute the sum $a[0] + a[1] + \cdots + a[n-1]$. It should be clear that a sequential algorithm can compute the sum in $O(n)$ time.

Let x and y be two arrays of size m and $n+m-1$. Also, let z be an array of size n. The convolution of x and y is a problem to compute z such that $z[i] = x[0] \cdot y[i] + x[1] \cdot y[i+1] + \cdots + x[m-1] \cdot y[i+m-1]$ for all i ($0 \leq i \leq n-1$). In this paper, we assume that $m \ll n$. The convolution can be computed by evaluating each $z[i]$ independently. We call such convolution *the direct convolution*, which takes $O(mn)$ time. From theoretical point of view, the convolution can be done faster using an FFT based technique. However, it has a large constant factor in computing time, especially for small m. Also, the main goal of this paper is to clarify the computing power of the HMM. Thus, we focus on the direct convolution, which is much simpler than the FFT-based convolution.

For the readers benefit, we will review that the PRAM can compute the sum of n numbers in $O(\frac{n}{p} + \log n)$ time using

p processors. Please see [22], [23] for the details. We first show that the sum can be computed in $O(\log n)$ time using $\frac{n}{2}$ processors. The sum can be computed by repeating the computation of pair-wise sums as illustrated in Figure 5. Let a be an array of $n = 2^m$ numbers. The details are spelled out as follows:

```
[Parallel summing algorithm]
for $t \leftarrow m-1$ downto 0 do
  for $i \leftarrow 0$ to $2^t - 1$ do in parallel
    $a[i] \leftarrow a[i] + a[i + 2^t]$
```

The reader should have no difficulty to confirm that the sum can be computed in $O(\log n)$ time.

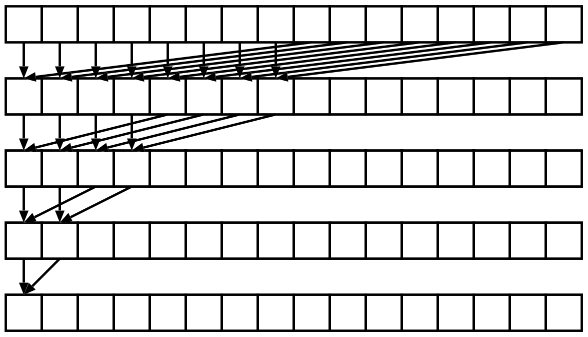

Figure 5. Illustrating the summing algorithm for n numbers

Suppose that we have p $(< n)$ processors. If this is the case, we partition the input into p groups of $\frac{n}{p}$ numbers each and compute the sum of each group in $O(\frac{n}{p})$ time. After that, the sum of the sums can be computed using the parallel algorithm illustrated in Figure 5 in $O(\log p)$ time. Thus, from $O(\frac{n}{p} + \log p) = O(\frac{n}{p} + \log n)$, we have,

Lemma 3: The sum of n numbers can be computed in $O(\frac{n}{p} + \log n)$ time using p processors on the PRAM.

Next, we will show that the direct convolution can be done in $O(\frac{mn}{p})$ time using p processors on the PRAM. First, assume that $p \geq n$. We partition p processors into n groups of $\frac{p}{n}$ processors each. Each $z[i]$ $(0 \leq i \leq n-1)$ is computed using $\frac{p}{n}$ processors. The computation of $z[i]$ involves m multiplications. This can be done in $\frac{m}{\frac{p}{n}} = \frac{mn}{p}$ time. After that the sum of m numbers can be computed in $O(\frac{m}{\frac{p}{n}} + \log m) = O(\frac{mn}{p} + \log m)$ time using $\frac{p}{n}$ processors. Thus, the direct convolution can be done in $O(\frac{mn}{p} + \log m)$ time.

If $p < n$, then we partition array z into p groups of $\frac{n}{p}$ elements in z. The values of $\frac{n}{p}$ elements are computed using a single processor. Since each z can be computed in $O(m)$ time using a single processor, $\frac{n}{p}$ elements can be computed in $O(\frac{mn}{p})$ time. Thus, we have,

Lemma 4: The direct convolution of m and $m+n-1$ numbers can be done in $O(\frac{mn}{p} + \log m)$ time using p processors on the PRAM.

As we have discussed, we can prove that algorithm for Lemmas 3 and 4 are time optimal. The sum algorithm on the PRAM has $\Omega(\frac{n}{p})$-time speed-up limitation and $\Omega(\log n)$-time reduction limitation. Also, the direct convolution has $\Omega(\frac{mn}{p})$-time speed-up limitation and $\Omega(\log m)$-time reduction limitation.

VI. An optimal parallel algorithm for computing the sum on the DMM and the UMM

The main purpose of this section is to review an optimal parallel algorithm for computing the sum on the memory machine models [17].

The sum can be computed by the PRAM algorithms shown for Lemma 3. We assume that p threads is used to compute the sum. For each t $(0 \leq t \leq m-1)$, 2^t operations "$a[i] \leftarrow a[i] + a[i + 2^t]$" are performed. These operation involve the following memory access operations:

- reading from $a[0], a[1], \ldots, a[2^t - 1]$,
- reading from $a[2^t], a[2^t + 1], \ldots, a[2 \cdot 2^t - 1]$, and
- writing in $a[0], a[1], \ldots, a[2^t - 1]$,

Since these memory access operations are contiguous, they can be done in $O(\frac{2^t}{w} + \frac{2^t l}{p} + l)$ time using p threads both on the DMM and on the UMM with width w and latency l from Theorem 2. Thus, the total computing time is $\sum_{t=0}^{m-1} O(\frac{2^t}{w} + \frac{2^t l}{p} + l) = O(\frac{2^m}{w} + \frac{2^m l}{p} + lm) = O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ and we have,

Lemma 5: The sum of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units using p threads on the DMM and on the UMM with width w and latency l.

As we have shown in our previous paper [17], the summing algorithm for Lemma 5 is optimal. The computing time is the sum of the three lower bounds, $\Omega(\frac{n}{w})$-time bandwidth limitation, $\Omega(\frac{nl}{p})$-time latency limitation, and $\Omega(l \log n)$-time reduction limitation. Let us briefly explain these limitations. Since n numbers in w memory banks must be read at least once, $\Omega(\frac{n}{w})$ time is necessary. Also, each thread can read at most one number in l time units. Thus, p threads can read $\frac{pt}{l}$ numbers in t time units. Since $\frac{pt}{l} \geq n$ must be satisfied, we have $t \geq \frac{nl}{p}$. We use similar discussion for $\Omega(\log n)$-time reduction limitation on the PRAM. Since each internal node of the rooted tree takes l time units to compute the sum, at least $\Omega(l \log n)$ time is necessary to compute the sum.

VII. An optimal parallel algorithm for computing the sum on the HMM

This section is devoted to show an optimal summing algorithm on the HMM. We assume that an input array a of n numbers are stored in the global memory.

The summing algorithm for Lemma 5 can be used to compute the sum using the global memory on the HMM. In other words, the sum of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units using p threads on the HMM width w and latency l. We first show a straightforward algorithm that uses only p_0 threads on $\mathrm{DMM}(0)$ of the HMM. The resulting algorithm runs in $O(\frac{n}{w}+\frac{nl}{p_0}+\log n)$ time units. Thus, the computing time is improved when $p=p_0$.

We can think that the input array a is a 2-dimensional array with p_0 columns and $\frac{n}{p_0}$ rows. First, each of the p_0 threads in the $\mathrm{DMM}(0)$ is assigned to a column of a and computes the column-wise sum. Let $b[0], b[1], \ldots, b[p_0-1]$ denote the p column-sums thus obtained. After that, the sum of b is computed by the algorithm for Lemma 5 on the $\mathrm{DMM}(0)$. The details of the algorithm are spelled out as follows:

```
[Straightforward Summing algorithm on the HMM]
for $t \leftarrow 0$ to $\frac{n}{p_0}-1$ do
  for $i \leftarrow 0$ to $p_0-1$ do in parallel
    $b[i] \leftarrow b[i]+a[t][i]$
$\mathrm{DMM}(0)$ computes the sum $b[0]+b[1]+\cdots+b[p_0-1]$
```

Let us evaluate the computing time. The readers should have no difficulty to confirm that the computation of the column-wise sums performs the contiguous read. Thus, the column-wise sums can be computed in $O(\frac{n}{w}+\frac{nl}{p_0})$ time units from Theorem 2. After that, the sum of them can be computed in $O(\frac{p_0}{w}+\frac{p_0\cdot 1}{p_0}+1\cdot\log p_0)=O(\frac{p_0}{w}+\log p_0)$ time units from Lemma 5. Therefore, the total computing time is $O(\frac{n}{w}+\frac{nl}{p_0}+l)+O(\frac{p_0}{w}+\log p_0)=O(\frac{n}{w}+\frac{nl}{p_0}+l+\log p_0)=O(\frac{n}{w}+\frac{nl}{p_0}+l+\log n)$, and we have,

Lemma 6: The sum of n numbers can be computed in $O(\frac{n}{w}+\frac{nl}{p_0}+l+\log n)$ time units using p_0 threads in the $\mathrm{DMM}(0)$ on the UMM with width w and latency l.

If $p_0 \geq wl$ then the latency limitation $\frac{nl}{p_0}$ can be hidden by the bandwidth limitation $\frac{n}{w}$. However, from the practical point of view, $p_0 \leq \frac{wl}{d}$ must be satisfied. To minimize the latency overhead, we set $p_0=\frac{wl}{d}$. If this is the case, the computing time is $O(\frac{n}{w}+\frac{nl}{\frac{wl}{d}}+l+\log n)=O(\frac{nd}{w}+l+\log n)$. In other words, the computing time has $O(\frac{nd}{w})$ latency overhead, which is much larger than $O(\frac{n}{w})$ bandwidth limitation.

We need to use all DMMs to hide the latency overhead. We will show a summing algorithm using all DMMs. Let p be the total number of threads and each DMM has $\frac{p}{d}$ threads. Recall that we assumed that $p \geq dw$, that is, each DMM has at least one warp. Again, suppose that a 2-dimensional array a with p columns and $\frac{n}{p}$ rows is storing the n input numbers. We assign one thread to each column and compute the column-wise sum. After that, the sum of the column-wise sums are computed in each DMM. Finally, the sum in each DMM is copied to the global memory and the sum of the sums are computed using $\mathrm{DMM}(0)$.

Let $b_i[j]$ $(0 \leq i \leq d-1, 0 \leq j \leq \frac{p}{d}-1)$ be the local register of $T_i(j)$ to store the column-wise sum. Also, let c be an array of size d in the global memory used to store the sum computed by every DMM. The details of the summing algorithm are spelled out as follows.

```
[Summing algorithm on the HMM]
for $t \leftarrow 0$ to $\frac{n}{p}-1$ do
  for $i \leftarrow 0$ to $d-1$ do in parallel
    for $j \leftarrow 0$ to $\frac{p}{d}-1$ do in parallel
    $b_i[j] \leftarrow b_i[j]+a[t][i\cdot\frac{p}{d}+j]$
Each $\mathrm{DMM}(i)$ computes the sum $b_i[0]+b_i[1]+\cdots$
$+b_i[\frac{p}{d}-1]$ and store it in $c[i]$.
$\mathrm{DMM}(0)$ compute the sum $c[0]+c[1]+\cdots+c[d-1]$.
```

Let us evaluate the computing time. We assume that n is large enough such that $n \geq d^2$. Also, we assume $n \geq p$. Since the computation of the column-wise sum performs the contiguous memory access, it takes $O(\frac{n}{w}+\frac{nl}{p}+l)$ time units from Theorem 2. After that, each sum $b_i[0]+b_i[1]+\cdots+b_i[\frac{p}{d}-1]$ is computed by using $\frac{p}{d}$ threads the summing algorithm for Lemma 5 on $\mathrm{DMM}(i)$. Since the latency is 1, it takes $O(\frac{p}{dw}+\log\frac{p}{d}) \leq O(\frac{n}{w}+\log n)$ time units. Finally, the sum $c[0]+c[1]+\cdots+c[d-1]$ is computed using $p_0=\frac{p}{d}$ threads by the algorithm for Lemma 6 in $O(\frac{d}{w}+\frac{dl}{\frac{p}{d}}+l+\log d)=O(\frac{d}{w}+\frac{d^2l}{p}+l+\log d) \leq O(\frac{n}{w}+\frac{nl}{p}+l+\log n)$. Thus, we have,

Theorem 7: The sum of n numbers can be computed in $O(\frac{n}{w}+\frac{nl}{p}+l+\log n)$ time units using p threads on the HMM with width w and latency l whenever $n \geq d^2$ and $n \geq p$.

we can remove the condition $n \geq d^2$ by computing the sum $c[0]+c[1]+\cdots+c[d-1]$ by a recursive manner. Due to the stringent page limitation we omit the explanation.

VIII. Optimal convolution on the DMM and the UMM

The main purpose of this section is to evaluate the performance of the convolution on the DMM and the UMM.

Let us start with a straightforward algorithm. We use $p=mn$ threads and each thread is used to compute $x[j]\cdot y[i+j]$ $(0 \leq i \leq n-1, 0 \leq j \leq m)$. For this purpose, mn threads performs read operations in $O(\frac{mn}{w}+\frac{mnl}{mn})=O(\frac{mn}{w}+l)$ time units. After that, every $z[i]$ $(0 \leq i \leq n-1)$ is computed using m threads using the algorithm for Lemma 5. Since the sums $z[0], z[1], \ldots, z[n-1]$ can be computed at the same time, we can reduce the latency overhead. Let us evaluate the computing time necessary to compute all $z[i]$'s at the same time. When we execute the algorithm for Lemma 5 using m threads to compute $z[i]$, the algorithm has $\log m$ stages. In other words, the optimal algorithm for computing the sum is executed for $t=\log m-1, \log m-2, \ldots, 0$. The reader should no difficulty to confirm that the contiguous memory access for $O(n2^t)$ numbers are performed in each Stage t. Thus, Stage t takes $O(\frac{n2^t}{w}+\frac{n2^tl}{mn}+l)=O(\frac{n2^t}{w}+l)$ time units. Therefore, all $z[i]$'s can be computed in $\sum_{t=0}^{\log m-1} O(\frac{n2^t}{w}+l)=O(\frac{mn}{w}+l\log m)$ time units.

Next, suppose that we have $p=n$ threads and each thread $T(i)$ is used to compute $z[i]$ as follows.

```
[Convolution on the DMM/UMM using $n$ threads]
for $t \leftarrow 0$ to $m-1$ do
  for $i \leftarrow 0$ to $n-1$ do in parallel
    $T(i)$ performs $z[i] \leftarrow z[i] + x[t] \cdot y[i+t]$
```

Let us evaluate the computing time. In each t, every thread read $x[t]$. Also, $T(0), T(1), \ldots, T(n-1)$ read $y[t], y[t+1], \ldots, y[t+n-1]$. Thus, the contiguous access is performed. Each t takes $O(\frac{n}{w} + \frac{nl}{n} + l) = O(\frac{n}{w} + l)$ time units from Theorem 2. Hence, the total computing time is $O(O(\frac{n}{w} + l) \cdot m = O(\frac{mn}{w} + lm)$ time.

Suppose that we have p threads such that $n < p < nm$. We partition p threads into n groups, each of which is used to compute $z[i]$. In other words, we partition the computation of $z[i]$ into $s = \frac{p}{n}$ blocks $z[i,0], z[i,1], \ldots, z[i,s-1]$ such that $z[i,j] = x[j \cdot \frac{m}{s}] \cdot y[i + j \cdot \frac{m}{s}] + x[j \cdot \frac{m}{s} + 1] \cdot y[i + j \cdot \frac{m}{s} + 1] + \cdots + x[(j+1) \cdot \frac{m}{s} - 1] \cdot y[i + (j+1)\frac{m}{s} - 1]$. After computing every $z[i,j]$ using p threads, we compute $z[i] = z[i,0] + z[i,1] + \cdots + z[i,s-1]$ using $\frac{n}{p}$ thread each. The details of the computation of each $z[i,j]$ are as follows.

```
[Convolution on the DMM/UMM using $p$ threads]
for $t \leftarrow 0$ to $\frac{m}{s} - 1$ do
  for $i \leftarrow 0$ to $n-1$ do in parallel
    for $j \leftarrow 0$ to $s-1$ do in parallel
      $z[i,j] \leftarrow z[i,j] + x[j \cdot \frac{m}{s} + t] \cdot y[i + j \cdot \frac{m}{s} + t]$
```

As before, for each t, the consecutive access for $ns = p$ numbers is performed. It takes $O(\frac{p}{w} + \frac{pl}{p} + l) = O(\frac{p}{w} + l)$ time units. Thus, the total computing time is $O(\frac{p}{w} + l) \cdot \frac{m}{s} = O(\frac{mp}{sw} + \frac{ml}{s}) = O(\frac{mn}{w} + \frac{mnl}{p})$ time units.

Theorem 8: The direct convolution of m and $m+n-1$ numbers can be done in $O(\frac{mn}{w} + \frac{mnl}{p} + l \log m)$ time units using p threads on the DMM and on the UMM with width w and latency l.

Let us discuss the lower bound of the computing time. The multiplication $x[t] \cdot y[i+t]$ must be computed by a thread. Since a warp of w threads is activated in turn, at most w multiplications are performed in a time unit. Hence, it takes at least $\Omega(\frac{mn}{w})$ time units (speed-up limitation). Also, p thread can read $\frac{pt}{l}$ numbers in t time units, $\frac{pt}{l} \geq mn$ must be satisfied. Thus, we have $t \geq \frac{mnl}{p}$ (latency limitation). Also, since the sum of m numbers must be computed for each $z[i]$, we have $\Omega(l \log m)$ time lower bound (reduction limitation). Thus, the algorithm for Theorem 8 is optimal.

IX. Optimal convolution on the HMM

This section is devoted to show optimal direct convolution on the HMM. We assume that two arrays x and y are stored in the global memory of the HMM. The goal of the convolution on the HMM is to compute z and store it in the global memory.

Let us design an algorithm for computing z using the HMM. The algorithm consists of three steps.

Step 1: Copy x and y from the global memory to the shared memory.

Step 2: Compute z in the shared memory.

Step 3: Copy z from the shared memory to the global memory.

We first show an implementation of the algorithm using p threads on d DMMs. We partition the input z into d groups $z_0, z_1, \ldots, z_{d-1}$ of length $\frac{n}{d}$ each such that $z_i = \langle z[i \cdot \frac{n}{d}], z[i \cdot \frac{n}{d} + 1], z[(i+1) \cdot \frac{n}{d} - 1] \rangle$. We use each $\mathrm{DMM}(i)$ with $\frac{p}{d}$ threads to compute $\frac{n}{d}$ z's in z_i. We assume that $\frac{p}{d} \geq w$ to ensure that each DMM has at least one warp of w threads.

For simplicity, we show how $\frac{p}{d}$ threads $T_0(0)$, $T_0(1)$, $\ldots$, $T_0(\frac{p}{d} - 1)$ on $\mathrm{DMM}(0)$ compute z_0. For the purpose of computing z_0, the values of $x[0], x[1], \ldots, x[m-1]$ and $y[0], y[i+1], \ldots, y[m + \frac{n}{d} - 1]$ are necessary. In Step 1, $\frac{p}{d}$ threads in $\mathrm{DMM}(0)$ copy these values to the shared memory. In Step 2, $\frac{p}{d}$ threads compute the convolution of $x[0], x[1], \ldots, x[m-1]$ and $y[0], y[1], \ldots, y[m + \frac{n}{d} - 1]$ on the shared memory. Finally, Step 3 copies the resulting values of z_0 to the global memory.

Let us evaluate the computing time. In Step 1, each DMM read m numbers in x and $m + \frac{n}{d} - 1$ numbers in y. Thus, totally, all d DMMs read $(m + m + \frac{n}{d} - 1) \cdot d \leq 2md + n$ numbers from the global memory. Since the memory access are contiguous, Step 1 takes $O(\frac{2md+n}{w} + \frac{2md+nl}{p} + l) = O(\frac{md+n}{w} + \frac{(md+n)l}{p} + l)$ time units from Theorem 2.

In Step2, the convolution of m numbers and $m + \frac{n}{d} - 1$ numbers are performed on $\mathrm{DMM}(0)$. Recall that DMMs on the HMM have memory access latency $l = 1$. Thus, from theorem 8 and from $\frac{p}{d} \geq w$, the convolution can be done in $O(\frac{m\frac{n}{d}}{w} + \frac{m\frac{n}{d}}{p} + \log m) = O(\frac{mn}{dw} + \log m)$ time units. Finally, Step 3 performs the copy of z to the global memory. It just copies n numbers from the shared memory to the global memory, the computing time is no more than Step 1. Hence, the total computing time is $O(\frac{md+n}{w} + \frac{(md+n)l}{p} + l) + O(\frac{mn}{dw} + \log m) = O(\frac{md+n}{w} + \frac{mn}{dw} + \frac{(md+n)l}{p} + l + \log m)$. Thus, we have,

Theorem 9: The convolution of two sequences of length m and $n+m-1$ can be computed in $O(\frac{md+n}{w} + \frac{mn}{dw} + \frac{(md+n)l}{p} + l + \log m)$ time units using p threads on the HMM with d DMMs, width w, and latency l.

From Theorem 9, we have the following corollary whenever $m \ll n$:

Corollary 10: The convolution of two sequences of length m and $n+m-1$ can be computed in $O(\frac{n}{w} + \frac{mn}{dw} + \frac{nl}{p} + l + \log m)$ time units using p threads on the HMM with d DMMs, width w, and latency l whenever $m \leq \frac{n}{d}$.

Let us discuss the optimality of Corollary 10. The multiplication $x[t] \cdot y[i+t]$ must be computed by a thread. Since array x of size $m+n-1$ in the w memory banks must be read at least once Hence, it takes at least $\Omega(\frac{m+n-1}{w}) = \Omega(\frac{n}{w})$ time units (bandwidth limitation). Also, p thread can read $\frac{pt}{l}$ numbers in t time units, $\frac{pt}{l} \geq n$ must be satisfied. Thus, we have $t \geq \frac{nl}{p}$. Further, since each $x[i]$ is read at least once, we need l time units. Hence, we have $\Omega(\frac{nl}{p} + l)$ time lower bound (latency limitation). The HMM has d DMMs, each of which can perform w multiplications. Thus, the HMM can perform at most dw multiplications in a time unit. Since we need mn multiplications, it takes at least $\Omega(\frac{mn}{dw})$ time units (speed-up limitation). Similarly to the PRAM reduction limitation, we have $\Omega(\log m)$ time lower bound (Reduction limitation). Thus, the algorithm for Corollary 10 is optimal.

X. CONCLUSION

The main contribution of this paper is to introduce the Hierarchical Memory Machine (HMM), which consists of multiple DMMs and a single UMM. The HMM is a more practical parallel computing model which reflects the architecture of GPUs. First, we presented that the sum of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l + \log n)$ time units using p threads on the HMM with width w and latency l, and proved that this computing time is optimal. We also showed that the direct convolution of m and $m + n - 1$ numbers can be done in $O(\frac{n}{w} + \frac{mn}{dw} + \frac{nl}{p} + l + \log m)$ time units using p threads on the HMM with d DMMs, width w and latency l. Finally, we proved that our implementation of the direct convolution is time optimal.

REFERENCES

[1] W. W. Hwu, *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.

[2] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing euclidean distance map with efficient memory access," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 68–76.

[3] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proc. of International Conference on Networking and Computing*, Dec. 2011, pp. 153–159.

[4] K. Ogawa, Y. Ito, and K. Nakano, "Efficient canny edge detection using a gpu," in *Proc. of International Conference on Networking and Computing*, Nov. 2010, pp. 279–280.

[5] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the optial poygon triangulation on the GPU," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*, Sept. 2012, pp. 1–15.

[6] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 4.2," 2012.

[7] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, pp. 260–276, July 2011.

[8] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 3.1," 2010.

[9] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1991.

[10] R. H. Bisseling, *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press, 2004.

[11] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, "LogP: towards a realistic model of parallel computation," in *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1993, pp. 1–12.

[12] R. Vaidyanathan and J. L. Trahan, *Dynamic Reconfiguration: Architectures and Algorithms*. Kluwer Academic/Plenum Publishers, 2004.

[13] K. Nakano, "Simple memory machine models for GPUs," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, May 2012, pp. 788–797.

[14] A. V. Aho, J. D. Ullman, and J. E. Hopcroft, *Data Structures and Algorithms*. Addison Wesley, 1983.

[15] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, 1972.

[16] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[17] K. Nakano, "An optimal parallel prefix-sums algorithm on the memory machine models for GPUs," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*, Sept. 2012, pp. 99–113.

[18] ——, "Efficient implementations of the approximate string matching on the memory machine models," in *Proc. of International Conference on Networking and Computing*, Dec. 2012, pp. 233–239.

[19] A. Kasagi, K. Nakano, and Y. Ito, "An implementation of conflict-free off-line permutation on the GPU," in *Proc. of International Conference on Networking and Computing*, 2012, pp. 226–232.

[20] K. Nakano, "Asynchronous memory machine models with barrier syncronization," in *Proc. of International Conference on Networking and Computing*, Dec. 2012, pp. 58–67.

[21] NVIDIA Corporation, "NVIDIA GeForce GTX580 GPU datasheet," 2011.

[22] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing*. Addison Wesley, 2003.

[23] M. J. Quinn, *Parallel Computing: Theory and Practice*. McGraw-Hill, 1994.