Optimality of Fundamental Parallel Algorithms on
the Hierarchical Memory Machine, with GPU
implementation

Koji Nakano and Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

Abstract—The Hierarchical Memory Machine (HMM) is a
theoretical parallel computing model that captures the essence
of CUDA-enabled GPU architecture. It has multiple streaming
multiprocessors with a shared memory, and the global memory
that can be accessed by all threads. The HMM has several
parameters: the number d of streaming multiprocessors, the
number p of threads per streaming multiprocessor, the number w
of memory banks of each shared memory and the global memory,
shared memory latency l, and global memory latency L. The main
purpose of this paper is to discuss optimality of fundamental
parallel algorithms running on the HMM. We first show that
image convolution for an image with n × n pixels using a filter of
size (2v+1) × (2v+1) can be done in O(n2/d2v + n2/p + n2/w) time
units on the HMM. Further, we show that this parallel
implementation is time optimal by proving the lower bound of
the running time. We then go on to show that the product of two
n × n matrices can be computed in O(n2/d2v + n2/p + n2/w) time
units on the HMM if the capacity of the shared memory in each
streaming multiprocessor is O(m2). This implementation is also
time optimal. We further clarify the conditions for
image convolution and matrix multiplication to hide the memory
access latency overhead and to maximize the global memory
throughput and the parallelism. Finally, we provide experimental
results on GeForce GTX Titan to support our theoretical analysis.

Keywords—Image convolution, matrix multiplication, parallel
algorithms, memory machine models, GPU, CUDA

I. INTRODUCTION

The GPU (Graphics Processing Unit) is a specialized circuit
designed to accelerate computation for building and manipulating
images [1], [2]. Latest GPUs are designed for general pur-
pose computing and can perform computation in applications
typically handled by the CPU. Hence, GPUs have recently
attracted the attention of many application developers [1],
[3]. NVIDIA provides a parallel computing architecture called
CUDA (Compute Unified Device Architecture) [4], [5], 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 multicores processors [6], since
they have thousands 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 [4]. The shared
memory is an extremely fast on-chip memory with lower ca-
pacity, say, 16-48 KBytes. The global memory is implemented
as an off-chip DRAM, and thus, it 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 [5], [6], [7]. 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
shared memory access performance, threads of CUDA should
access distinct memory banks to avoid the bank conflicts of
the memory accesses. To maximize the throughput 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. Also, the latency of the global
memory access is several hundred clock cycles, while that of
the shared memory access is quite small [4]. Hence, we
should minimize the memory access to the global memory to
maximize the performance.

Many researchers have been devoted to implement many
useful operations in GPUs. However, most of parallel imple-
mentations in GPUs are evaluated by the running time of a
particular GPU. The performance of implementations depends
on many factors including compiler optimization, program-
ming skills, GPU model numbers, among others. Hence, the
running time on an actual GPU may not indicate the good-
ness of parallel algorithms and implementations. Theoretical
parallel computing models that capture the features of GPU
architecture should be used to evaluate the performance of
algorithms and implementations.

Recently, we have introduced three models, the Discrete
Memory Machine (DMM), the Unified Memory Machine
(UMM), and the Hierarchical Memory Machine (HMM) which
reflect the essential features of computation performed by
CUDA-enabled GPUs [8], [9], [10], [11]. The DMM is a theo-
retical parallel computing model of a streaming multiprocessor
in a CUDA-enabled GPU. It has a shared memory with w
memory banks, and w threads in a warp can access the shared
memory at the same time. The UMM is a model for the parallel computation using the global memory of a CUDA-enabled GPU. It also has a global memory with \( w \) memory banks. The difference of the shared memory of the DMM and the global memory of the UMM is the restriction of access to memory banks. The same address of memory banks of the global memory must be accessed in each time unit, while different addresses of memory banks of the shared memory can be accessed. The HMM is a hybrid of the DMM and the UMM, which can be used to design and evaluate parallel algorithms using multiple streaming multiprocessors in a GPU. The HMM has multiple DMMs each of which has a shared memory. It also has a global memory which can be addressed by threads in all DMMs. The reader should refer to Figure 1 illustrating the HMM with width 4 and 3 DMMs. We use parameters \( w, L, l \) to denote the number of memory banks, the global memory access latency, and the shared memory access latency. We also use parameters \( d \) and \( p \) to denote the number of DMMs in the HMM, and the number of threads in each DMM. Thus, the HMM has totally \( dp \) threads. By using the HMM, we can give a theoretical analysis of performance of algorithms developed for CUDA-enabled GPUs using these parameters, \( w, L, l, d, \) and \( p \). Actually, theoretical analysis of algorithms on the HMM approximates the performance of them on GPUs. For example, we have presented offline permutation algorithms for the DMM in [12] and for the HMM [13]. These offline permutation algorithms are implemented on CUDA-enabled GPUs. Experimental results showed that theoretical analysis approximates the actual performance on the GPU. We have also shown in [14] that parallel algorithms designed for the HMM are efficient on the GPU. Hence, the DMM, the UMM, and the DMM are promising parallel computing model for parallel computing using CUDA-enabled GPUs.

Suppose that an image \( a \) of size \( n \times n \) and kernel \( b \) of size \((2v+1) \times (2v+1)\) are given. Image convolution is a task to compute the convolution of \( a \) and \( b \). It is one of the most important operations in the area of image processing, because it is used for various image processing applications [15]. For example, we can blur images by image convolution using a Gaussian kernel. Image convolution using Sobel kernels is used to detect edges in an image [16]. Image convolution can be implemented efficiently in GPUs using CUDA [17]. The first contribution of this paper is to present an optimal implementation for image convolution in the HMM. Our implementation runs \( O(\frac{n^2}{w} + \frac{n^2L}{dp} + \frac{n^2v^2}{dw} + \frac{n^2v^2}{dp}) \) time units on the HMM. We can think that four terms of the performance correspond to the global memory bandwidth, the global memory latency, the shared memory bandwidth, and the shared memory latency as shown in Table I. We also prove that any implementation of image convolution in the HMM takes \( \Omega(\frac{n^2}{w} + \frac{n^2L}{dp} + \frac{n^2v^2}{dw} + \frac{n^2v^2}{dp}) \) time units by exploring these four terms. Thus, our implementation for image convolution in the HMM is time optimal. We further show that, \( wL \leq dp \) and \( wL \leq p \) must be satisfied to hide the memory access latency. Also, the global memory throughput is maximized if \( v^2 \leq d \), and the parallelism is maximized otherwise.

Matrix multiplication is one of the most well-known applications which can be implemented in GPUs using CUDA very efficiently. Thus, it is introduced as the first example of a CUDA C program [4]. Our second contribution is to show time optimal parallel implementation for multiplying two matrices of size \( n \times n \) on the HMM. Our implementation runs \( O(\frac{n^3}{mwp} + \frac{n^3L}{mdp} + \frac{n^3}{dw} + \frac{n^3}{dp}) \) time units on the HMM if the capacity of each shared memory is \( O(m^2) \). Similarly to image convolution, the four terms of the performance correspond to four hardware limitations (Table I). By analyzing each of the four terms, we prove that any implementation of matrix multiplication in the HMM takes \( \Omega(\frac{n^3}{mwp} + \frac{n^3L}{mdp} + \frac{n^3}{dw} + \frac{n^3}{dp}) \) time units. Hence, our implementation is time optimal. In the same way as image convolution, we can see that \( wL \leq dp \).
and \( w l \leq p \) must be satisfied to hide the memory access latency. Further, the global memory throughput is maximized if \( m \leq d \), and the parallelism is maximized otherwise. We have implemented our image convolution and matrix multiplication algorithms using CUDA C. The experimental results on GeForce GTX Titan show that our theoretical analysis approximates the experimental results.

The rest of this paper is organized as follows. Section II introduces three memory machines, the Discrete Memory Machine (DMM), the Unified Memory Machine (UMM), and the Hierarchical Memory Machine (HMM), which are theoretical parallel computing models for CUDA-enabled GPUs. It also shows several fundamental memory access operations used later. In Section III, we show an implementation of image convolution in the HMM running \( O\left( \frac{n^2}{m} \right) \) time units. Section IV discusses the time lower bound for image convolution on the HMM, and shows the optimality of our implementation. It also shows conditions to hide the memory access latency and to maximize global memory throughput and parallelism. In Section V, we present an implementation of matrix multiplication in the HMM running \( O\left( \frac{n^2}{4m} + \frac{n^2}{4d} + \frac{n^4}{4dp} \right) \) time units. We also prove that our implementation of matrix multiplication is optimal in Section VI. Finally, Section VII provides experimental results on GeForce GTX Titan. Section VIII concludes our work.

## II. THE DMM, THE UMM, AND THE HMM

The main purpose of this section is to define three memory machine models: the Discrete Memory Machine (DMM), the Unified Memory Machine (UMM), and the Hierarchical Memory Machine (HMM), which capture the essence of parallel computing on CUDA-enabled GPUs.

We first define the Discrete Memory Machine (DMM) \cite{8,18} of width \( w \) and latency \( l \). Let \( B[i] = \{ j, j+w, j+2w, j+3w, \ldots \} \) \((0 \leq j \leq w-1)\) denote a set of addresses arranged in the \( i \)-th memory bank. In other words, each address \( i \) is in the \((i \mod w)\)-th memory bank. We assume that data with addresses in different banks can be accessed in a time unit, but no two data 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 (Memory Management Unit). Thus, it takes \( k + l - 1 \) time units to complete memory access requests to \( k \) distinct data in a particular bank.

We assume that \( p \) threads are partitioned into \( \frac{w}{p} \) groups of \( w \) threads called warps. More specifically, \( p \) threads \( T(0), T(1), \ldots, T(p-1) \) are partitioned into \( \frac{w}{p} \) warps \( W(0), W(1), \ldots, W(\frac{w}{p}-1) \) such that \( W(i) = \{ T(i \cdot w), T(i \cdot w + 1), \ldots, T((i+1)\cdot w-1) \} \) \((0 \leq i \leq \frac{w}{p}-1)\). 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, \( W(0), W(1), \ldots, W(\frac{w}{p}-1) \) are dispatched in a round-robin manner if at least one thread in a warp requests 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) \cite{8,11} of width \( w \) and latency \( L \). Let \( A[j] = \{ j \cdot w, j \cdot w + 1, \ldots, (j+1)\cdot w-1 \} \) denote the \( j \)-th address group. We assume that data 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 \( \frac{w}{p} \) groups of \( w \) threads called warps. We assume that each memory access request is completed when it reaches the last pipeline stage. Two warps \( W(0) \) and \( W(1) \) access to \((7, 5, 15, 0)\) and \((10, 11, 12, 9)\), respectively. In the DMM, memory access requests by \( W(0) \) are separated into two pipeline stages, because addresses 7 and 15 are in the same bank \( B(3) \). Those by \( W(1) \) occupy 1 stage, because all requests are in distinct banks. Thus, the memory requests occupy three stages, it takes \( 3+4-1 = 6 \) time units to complete the memory access. In the UMM, memory access requests by \( W(0) \) are destined for three address groups. Hence the memory requests occupy three stages. Similarly those by \( W(1) \) occupy two stages. Hence, it takes \( 5+6-1 = 10 \) time units to complete the memory access.

Note that, the architecture of pipeline registers illustrated in Figure 2 are imaginary, and it is used only for evaluating the computing time. The actual architecture should involves a multistage interconnection network \cite{19,20} or sorting network \cite{21,22}, to route memory access requests.

Finally, we define the Hierarchical Memory Machine

### TABLE I.

<table>
<thead>
<tr>
<th>Image convolution</th>
<th>Sequential algorithm</th>
<th>Global memory bandwidth</th>
<th>Global memory latency</th>
<th>Shared memory bandwidth</th>
<th>Shared memory latency</th>
<th>Shared memory capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( O(n^2w^2) )</td>
<td>( O\left( \frac{n^2}{m} \right) )</td>
<td>( O\left( \frac{n^2}{d} \right) )</td>
<td>( O\left( \frac{n^2}{dp} \right) )</td>
<td>( O\left( \frac{n^2}{d} \right) )</td>
<td>( O(\frac{n^2}{d}) )</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>( O(n^3) )</td>
<td>( O\left( \frac{n^2}{m} \right) )</td>
<td>( O\left( \frac{n^2}{d} \right) )</td>
<td>( O\left( \frac{n^2}{dp} \right) )</td>
<td>( O\left( \frac{n^2}{d} \right) )</td>
<td>( O(\frac{n^2}{d}) )</td>
</tr>
</tbody>
</table>

\( n \times n \): the size of an image matrix, \((2n + 1) \times (2n + 1)\): the size of a kernel, \( L \): the global memory latency, \( l \): the shared memory latency, \( d \): the number of DMMs, \( p \): the number of threads in each DMM

---

628
(HMM). The HMM consists of $d$ DMMs and a single UMM as illustrated in Figure 1. 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. All DMMs work in parallel. Threads are partitioned into warps of $w$ threads, and each warp is dispatched for the memory access for the shared memory in turn. Further, each warp of $w$ threads in all DMMs can send memory access requests to the global memory. Figure 1 illustrates the architecture of the HMM with $d = 3$ DMMs and $w = 4$. 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 CUDA-enabled GPUs. Also, it makes sense to assume that the shared memory in each DMM can store up to $O(w^2)$ words of data. CUDA enabled-GPUs can store $12w^2$ words of data as follows. The capacity of the shared memory in a streaming multiprocessor of CUDA-enabled GPUs is up to 48Kbytes [4]. Since the number of memory banks and the number of threads in a warp is $w = 32$, an array of $w^2$ 32-bit integers occupies 4K bytes. Thus, the shared memory cannot store more than 12 such arrays. Despite this fact, we may use parameter $m$ to denote the capacity of the shared memory to see its effect for the computational power. We use $l$ and $L$ to denote the memory access latencies of the shared memory in a DMM and the global memory of the UMM. The memory access latency of the global memory of CUDA-enabled GPUs is several hundred clock cycles, while that of the shared memory of a streaming multiprocessor is several clock cycles [4]. Hence, it makes sense to assume that $l \ll L$.

For later reference, we will evaluate the time necessary to complete fundamental memory access operations. Suppose that $p (\geq w)$ threads in $\frac{w}{w}$ warps in a DMM of the HMM access $n$ elements in the shared memory, $\frac{n}{p}$ elements each in turn. If all memory access requests by $w$ threads in all warps are conflict-free, the first $p$ elements can be accessed in $\frac{w}{w} + l - 1$ time units. Since this memory access is repeated $\frac{p}{w}$ times, it takes $\frac{p}{w} \cdot (\frac{w}{w} + l - 1) = O(\frac{w}{w} + \frac{l}{p})$ time units to access $n$ elements. Hence, we have,

**Lemma 1:** If $n$ memory access requests by $p (\geq w)$ threads in a DMM with width $w$ and latency $l$ are conflict-free, then they can be completed in $O(\frac{n}{w} + \frac{l}{p})$ time units.

Suppose that the HMM has $d$ DMMs with $p$ threads each and they need to copy $n$ elements stored in consecutive addresses of the global memory to the shared memory of the DMMs such that each shared memory stores $\frac{n}{d}$ elements. Each of the $dp$ threads works for copying $\frac{n}{wd}$ elements. Clearly, memory access to the global memory is coalesced and that to the shared memory is conflict-free. Hence, the first $pd$ elements can be read from the global memory in $\frac{n}{wd} + L - 1$ time units. After that, they can be written in the shared memory in $\frac{n}{wd} + l - 1$ time units. Thus, the first $pd$ elements can be copied in $\frac{n}{wd} + L - 1 + (\frac{n}{wd} + l - 1) = O(\frac{n}{wd} + L)$ time units from $l \ll L$. Since this operation is repeated $\frac{n}{wd}$ times, the total running time is $\frac{n}{wd} \cdot O(\frac{n}{wd} + L) = O(\frac{n}{wd} + \frac{l}{wd})$ time units. Hence, we have,

**Lemma 2:** The task of copying $n$ elements in the global memory to the shared memory of $d$ DMMs with $p$ threads each takes $O(\frac{n}{wd} + \frac{l}{wd})$ time units of the HMM. Clearly, the task of copying in the opposite direction, that is, from the shared memories to the global memory can be done in the same time units.

Even if $n$ elements are not in consecutive addresses of the global memory and they are separated in several segments, the task of copy can be done in the same time units. Suppose that we have $n$ elements in the global memory to be copied into the shared memories of $d$ DMMs such that each shared memory stores $\frac{n}{d}$ elements each. We assume that $n$ elements are partitioned into at most $\frac{n}{w}$ segments each of which consitute elements in consecutive addresses. For example, a sub matrix of size $m \times m$ in a large submatrix of size $n \times n (m \leq n)$
corresponds to $m$ segments of $m$ elements each. We have the following lemma.

**Lemma 3:** The task of copying $n$ elements separated in at most $\frac{n}{w}$ segments in the global memory to the shared memory of $d$ DMMs with $p$ threads each takes $O\left(\frac{n}{w} + \frac{n^2}{wp}\right)$ time units of the HMM.

Due to the stringent page limitation, we omit the proof of Lemma 3. Please see [11] for the details and the proof. Similarly, the copy in the opposite direction can be done in the same time units.

III. IMAGE CONVOLUTION ON THE HMM

Let $a$ be an image of size $n \times n$, and $b$ be a kernel of size $(2v+1) \times (2v+1)$. The image convolution is to compute an image $c = a \otimes b$ of size $n \times n$ such that

$$c(i, j) = \sum_{s=-v}^{v} \sum_{t=-v}^{v} a(i+s, j+t) \cdot b(v+s, v+t).$$

For simplicity, we assume that $a(i, j) = 0$ if $(i, j)$ is out of range. It should be clear that, the values of all $c(i, j)$s can be computed in $O(n^2v^2)$ time in an obvious way.

Suppose that image $a$ of size $m \times m$ and kernel $b$ of size $(2v+1) \times (2v+1)$ are stored in the shared memory of the DMM. We first show a parallel implementation of image convolution in the DMM with $p$ threads. We partition output image $c$ into $\frac{m}{p}$ groups of $p$ pixels each in a raster scan order of $c$. Image $c$ is computed group by group in turn. We formally describe a parallel algorithm for the case that $p = n$ threads are used. Clearly, each row of $c$ corresponds to a group if this is the case.

**[Algorithm CONV-DMM]**

for $i \leftarrow 0$ to $n - 1$ do

for $j \leftarrow 0$ to $n - 1$ do in parallel

$T(j)$ performs $c(i, j) \leftarrow 0$

for $s \leftarrow -v$ to $v$ do

for $t \leftarrow -v$ to $v$ do

if $0 \leq i + s, j + t \leq n - 1$ then

$T(j)$ performs $c(i, j)$

$c(i, j) + a(i+s, j+t) \cdot b(v+s, v+t)$

The reader should have no difficulty to confirm that Algorithm CONV-DMM can be modified to perform the multiplication when $p \neq n$. If $p < n$ then each row of $c$ is computed in $\frac{p}{n}$ rounds. If $p > n$ then $\frac{n}{p}$ rows of $c$ are computed at the same time.

Let us evaluate the computing time of Algorithm CONV-DMM. For each pair $s$ and $t$, $p$ threads read $p$ elements in $c$, $p$ pixels in $a$ and one element in $b$ and write $p$ elements in $c$. Since these memory access operation is conflict-free, this procedure takes $O\left(\frac{2v}{w} + \frac{2v^2}{p}\right) = O\left(\frac{2v}{w} + l\right)$ time units from Lemma 1. To compute the value of $p$ elements in $c$, this operation is repeated $(2v+1)^2$ times. Thus, the value of $p$ elements in $c$ can be computed in $O\left(\frac{2v}{w} + v^2l\right)$ time units. Further, since this procedure is repeated $\frac{n^2}{p}$ times, the total computing time is $\frac{n^2}{p} \cdot O\left(\frac{2v}{w} + v^2l\right) = O\left(\frac{n^2v^2}{w} + \frac{n^2v^3}{p}\right)$. Thus, we have,

**Lemma 4:** Algorithm CONV-DMM computes the convolution of an $n \times n$ image and a kernel of size $(2v+1) \times (2v+1)$ in $O\left(\frac{n^2v^2}{w} + \frac{n^2v^3}{p}\right)$ time units using $p$ threads on the DMM with width $w$ and latency $l$.

Next, we will show a parallel algorithm for the HMM. Suppose that the global memory is storing an image $a$ of size $n \times n$ and kernel $b$ of size $(2v+1) \times (2v+1)$. The goal is to compute output image $c = a \otimes b$ in the global memory. Let $d$ be the number of available DMMs in the HMM. We partition $c$ into $s \times s$ subimages of size $w \times w$ each, where $s = \frac{n}{w}$. Each DMM computes $\frac{s^2}{d}$ subimages one by one. Let $c[i, j]$ ($0 \leq i, j \leq s-1$) be a subimage in $i$-th row and $j$-th column. Note that to compute a subimage of size $w \times w$ in $c$, a subimage of size $(w+2v) \times (w+2v)$ in $a$ is necessary. Let $a[i, j]$ be a subimage of necessary to compute $c[i, j]$. A DMM is assigned to $c[i, j]$ computes it as follows:

**[Algorithm CONV-HMM for subimage $c[i, j]$]**

Step 1: Copy $a[i, j]$ and $b$ from the global memory to the shared memory

Step 2: Compute $c[i, j] \leftarrow a[i, j] \otimes b$ using Algorithm CONV-DMM

Step 3: Copy $c[i, j]$ from the shared memory to the global memory

Algorithm CONV-HMM computes the resulting image $c$ by executing these three steps for subimages $c[i, j]$ for all $i$ and $j$ ($0 \leq i, j \leq s - 1$) using $d$ DMMs.

Let us evaluate the total computing time of Algorithm CONV-HMM. First, all DMMs compute $d$ subimages of $c$ in parallel. Since $a[i, j]$ has $(w+2v)^2$ pixels, totally $d(w+2v)^2$ pixels are copied from the global memory to the shared memories of $d$ DMMs in Step 1. From Lemma 3 and $v < w$, this copy operation takes $O\left(\frac{2v}{w} + \frac{2v^2}{p}\right) = O\left(dw + \frac{w^2}{p}\right)$ time units. After that, kernel $b$ of size $(2v+1) \times (2v+1)$ is copied from the global memory to the shared memory. Clearly, $b$ is smaller than subimage $a[i, j]$, it takes no more than $O\left(dw + \frac{w^2}{p}\right)$ time units. Hence, Step 1 takes $O\left(dw + \frac{w^2}{p}\right)$ time units. In Step 2, each DMM executes Algorithm CONV-DMM in parallel, which takes $O\left(\frac{w^2}{w} + \frac{w^2}{p}\right) = O\left(dw^2 + \frac{w^2}{p}\right)$ time units from Lemma 4. After that, each DMM copies $c[i, j]$ in parallel in Step 3. Since $c[i, j]$ has $w^2$ pixels, this copy operation takes $O\left(\frac{2v^2}{w} + \frac{2v^2}{p}\right) = O\left(dw^2 + \frac{w^2}{p}\right)$ time units from Lemma 3. Thus, $d$ DMMs computes $d$ subimages in $O\left(dw^2 + \frac{w^2}{w} + \frac{w^2}{p}\right)$ time units. Since this procedure is repeated $\frac{n^2}{p}$ times, Algorithm CONV-HMM runs $\frac{n^2}{p} \cdot O\left(dw^2 + \frac{w^2}{w} + \frac{w^2}{p}\right) = O\left(\frac{n^2w^2}{w} + \frac{n^2w^3}{wp}\right)$ time units. Thus, we have,

**Theorem 5:** Algorithm CONV-HMM runs $O\left(\frac{n^2w^2}{w} + \frac{n^2w^3}{wp}\right)$ time units using $d$ DMMs with $p$ threads each on the HMM with width $w$, global memory latency $L$, and shared memory latency $l$. 630
IV. LOWER BOUND OF IMAGE CONVOLUTION

We will prove the optimality of Algorithm CONV-HMM. More specifically, we will show four limitations, the global memory bandwidth limitation, the global memory latency limitation, the shared memory bandwidth limitation, and the shared memory latency limitation to implement image convolution on the HMM. Clearly, each of the \( n^2 \) pixels in the global memory must be accessed at least once and at most \( w \) pixels in the global memory can be accessed in a time unit. Hence, at least \( \Omega(\frac{n^2}{w}) \) time units are necessary (the global memory bandwidth limitation). The HMM has totally \( dp \) threads and each thread can send at most one memory access request to the global memory in \( L \) time units. Hence, \( dp \) threads can send at most \( tdp \) memory access requests in \( tL \) time units for any \( t \). Since \( n^2 \) memory access requests must be destined for the global memory, \( tdp \geq n^2 \) must be satisfied to send \( n^2 \) memory access requests by \( dp \) threads. Thus, \( tL \geq \Omega(\frac{n^2}{dp}) \) time units are necessary (the global memory latency limitation). For image convolution we need to compute \( a(i+s, j+t) \cdot b(v+s, v+t) \) for \( n^2v^2 \) pairs of \( a(i+s, j+t) \) and \( b(v+s, v+t) \). These elements must be read from the global memory or the shared memory. Since the HMM has \( d \) shared memories and one global memory with \( w \) memory banks each, at most \((d+1)w\) elements can be read at the same time. Thus, at least \( \frac{n^2v^2}{(d+1)w} = \Omega(\frac{n^2v^2}{w}) \) time units are necessary (the shared memory bandwidth limitation). Also, each of the \( dp \) threads in the HMM can send at most one memory access request to a shared memory in \( L \) time units or to the global memory in \( L \) time units. Since \( n^2v^2 \) memory access requests must be sent to the shared memories or the global memory and \( L \ll L \), at least \( \Omega(\frac{n^2v^2}{dp}) \) time units are necessary (the shared memory latency limitation). Thus, we have:

Theorem 6: Any parallel implementation of image convolution in the HMM with \( d \) DMMs with \( p \) threads each, with \( w \) global memory latency \( L \), and shared memory latency \( l \), takes at least \( \Omega(\frac{n^2v^2}{w} + \frac{n^2}{dp} + \frac{n^2v^2}{w} + \frac{n^2v^2}{dp}) \) time units.

From Theorem 6, Algorithm CONV-HMM shown for Theorem 5 is time optimal.

We show the conditions to hide the memory access latency.

The global memory bandwidth hides the global memory latency if \( \frac{n^2}{w} \geq \frac{n^2L}{dp} \) that is, \( wL \leq dp \) is satisfied. Strictly speaking, the condition must be \( wL = O(dp) \), because the computing time is evaluated using big-O notation. However, to avoid confusion, we simply write \( wL \leq dp \). Recall that the global memory is connected to \( L \)-stage pipeline with each stage having \( w \) registers. Hence, memory access requests are queued in \( wL \) pipeline registers. Also, at most one memory access request by each of \( dp \) threads in the HMM can be queued. Hence, it is possible to fill pipeline registers with memory access requests only if \( wL \leq dp \) is satisfied. The shared memory bandwidth hides the shared memory latency if \( \frac{n^2}{dp} \geq \frac{n^2v^2}{(d+1)w} \), that is, \( wL \leq p \). Recall that the shared memory in each DMM is connected to \( L \)-stage pipeline with each stage having \( w \) registers. At most one memory access request by each of \( p \) threads in a DMM, and memory requests are queued in \( wL \) pipeline registers. Hence, it is possible to fill pipeline registers with memory access requests if \( wL \leq p \) is satisfied.

Suppose that both \( wL \leq dp \) and \( wL \leq p \) are satisfied. If this is the case, Algorithm CONV-HMM runs \( O(\frac{n^2}{w} + \frac{n^2v^2}{dp}) \) time units. Intuitively, \( O(\frac{n^2}{w}) \) and \( O(\frac{n^2v^2}{dp}) \) correspond to the global memory access and the shared memory access, respectively. If \( \frac{n^2}{w} \geq \frac{n^2v^2}{dp} \), that is \( v^2 \leq d \), the time for global memory access dominates that for the shared memory access. Hence, we can think that global memory access throughput in Algorithm CONV-HMM is maximized if \( v^2 \leq d \), because Algorithm CONV-HMM runs \( O(\frac{n^2}{w}) \) time units. On the other hand, if \( v^2 > d \), Algorithm CONV-HMM runs \( O(\frac{n^2v^2}{dp}) \). Recall that sequential algorithm for image convolution runs in \( O(n^2v^2) \) time. Also, \( w \) threads in a warp out of \( p \) threads in a DMM work in each time unit, and thus \( wd \) threads in the HMM work in parallel. Hence, the acceleration ratio can be up to \( wd \), and the running time of \( O(\frac{n^2v^2}{dp}) \) attains optimal speed-up. Thus, we can say that parallelism is maximized if \( v^2 > d \).

V. MATRIX MULTIPLICATION ON THE HMM

In this section, we first show Algorithm MUL-DMM that computes the product of two matrices on the DMM. After that, we present Algorithm MUL-HMM that computes the product of two matrices on the HMM.

Let us start with the computation of the product of two matrices \( a \) and \( b \) of size \( n \times n \) \((n \geq w)\) each stored in the shared memory of a DMM. The goal is to compute a matrix \( c \) of the same size such that \( c = a \times b \). In other words,

\[
c(i,j) = \sum_{k=0}^{m-1} a(i,k) \cdot b(k,j)
\]

is computed for all \( i \) and \( j \) \((0 \leq i, j \leq n-1)\).

Suppose \( w \leq p \leq n^2 \) threads are used for computing the product. For simplicity, we assume that \( n \) and \( p \) are multiples of \( w \), and \( n^2 \) is a multiple of \( p \). We partition output matrix \( c \) into \( \frac{n^2}{w} \) groups of \( p \) elements each in a raster scan order of \( c \). Output matrix \( c \) is computed by group by group in turn. We formally describe a parallel algorithm for the case that \( p = n \). Clearly, each row of \( c \) corresponds to a group if this is the case.

[Algorithm MUL-DMM]

for \( i \leftarrow 0 \) to \( n-1 \) do

for \( j \leftarrow 0 \) to \( n-1 \) do in parallel

\( T(j) \) performs \( c(i,j) \leftarrow 0 \)

for \( k \leftarrow 0 \) to \( n-1 \) do

\( T(j) \) performs \( c(i,j) \leftarrow c(i,j) + a(i,k) \cdot b(k,j) \)

The reader should have no difficulty to modify Algorithm MUL-DMM to compute the product when \( p \neq n \). If \( p < n \) then each row of \( c \) is computed in \( \frac{n}{p} \) rounds. If \( p > n \) then \( \frac{n}{p} \) rows of \( c \) is computed at the same time.

Let us evaluate the computing time. For each value of \( k \), \( p \) threads read \( p \) elements in \( c \), \( \max(\frac{n}{p}, 1) \) elements in \( a \) and \( p \) element in \( b \), and write \( p \) elements in \( c \). Since these memory
access operation is conflict-free, this procedure takes $O(\frac{p}{w} + l)$
time units from Lemma 1. To compute the value of $p$ elements
in $c$, this operation is repeated $n$ times. Thus, the value of $p$
elements in $c$ can be computed in $O(\frac{p}{w} + nl)$ time units.
Since this procedure is repeated $\frac{m}{w}$ times, the total computing
time is $\frac{n^2}{p} \cdot O(\frac{p}{w} + nl) = O(\frac{n^2}{w} + \frac{n^2l}{p})$ time units. Thus, we have,

**Lemma 7:** Algorithm MUL-DMM computes the product of
two matrices of size $n \times n$ each in $O(\frac{n^2}{w} + \frac{n^2l}{p})$ time units
using $p$ threads on the DMM with width $w$ and latency $l$.

Next, we will show an algorithm for matrix multiplication
on the HMM. We assume that two matrices $A$ and $B$ of size
$n \times n$ ($n \geq w$) each are stored in the global memory of
the HMM. The goal is to store the product of $A$ and $B$ in
matrix $C$ of the global memory. Let $m$ be a parameter such
that $w \leq m \leq n$, and let $s = \frac{n}{m}$. We partition matrix $C$
into $s^2$ submatrices of size $m \times m$ each. Let $d$ ($\leq s^2$)
be the number of DMMs in the HMM. Each DMM is assigned
$s^2$ submatrices of $C$, and is responsible for computing the resulting
values of them. Let $a[i, j], b[i, j]$, and $c[i, j]$ ($0 \leq i, j \leq s - 1$)
denote submatrices of size $m \times m$ in $i$-th row and $j$-th column of $A$, $B$, and $C$,
respectively. Clearly, $c[i, j]$ can be computed by

$$c[i, j] = \sum_{k=0}^{s-1} a[i, k] \times b[k, j],$$

where $a[i, k] \times b[k, j]$ denotes the product of two matrices.
Thus, a DMM assigned to $c[i, j]$ can compute $c[i, j]$ as follows:

**Algorithm MUL-HMM for $c[i, j]$**

1. Copy $a[i, k]$ and $b[k, j]$ from the global memory to
2. the shared memory
3. Compute $c[i, j] = c[i, j] + a[i, k] \times b[k, j]$ by
4. Algorithm MUL-DMM
5. Copy $c[i, j]$ from the shared memory to the global memory

We assume that $c[i, j]$ in the shared memory is initialized by
zero. Since $c[i, j]$ stores $\sum_{k=0}^{s-1} a[i, k] \times b[k, j]$ when Algorithm
MUL-HMM terminates, the product of two matrices is
computed correctly. Since the HMM has $d$ DMMs, $d \times s^2$ submatrices can
be computed in parallel, and this parallel computation is repeated
$\frac{m}{w}$ times.

Let us evaluate the computing time. First, $d$ DMMs copy $d$
submatrices of $A$ and $B$, respectively, from the global memory
to the shared memory. Since each submatrix has $m^2$ elements,
this copy operation can be done in $O(\frac{m^2}{w} + \frac{m^2L}{p}) =
O(\frac{m^2}{w} + \frac{m^2L}{p})$ time units from Lemma 3. After that, Algorithm
MUL-DMM is executed by $d$ DMMs in parallel. This
matrix multiplication takes $O(\frac{n^2}{w} + \frac{n^2L}{p})$ time units
from Lemma 7. These copy and matrix multiplication operations
are repeated $s$ times, $c[i, j]$ can be computed in $s \cdot O(\frac{n^2}{w} +
\frac{m^2L}{p} + \frac{m^2}{w} + \frac{m^2L}{p})$ time units. Finally, $d \times s^2$ copies are
generated to the global memory in $O(\frac{m^2}{w} + \frac{m^2L}{p})$ time units. Hence, $d$
DMMs computes $d$ submatrices in $s \cdot O(\frac{m^2}{w} + \frac{m^2L}{p} + \frac{m^2}{w} + \frac{m^2L}{p})$
time units. Since the HMM computes $s^2$ submatrices, the total
computing time is $\frac{n^2}{p} \cdot s \cdot O(\frac{m^2}{w} + \frac{m^2L}{p} + \frac{m^2}{w} + \frac{m^2L}{p}) =
O(\frac{m^2}{w} + \frac{m^2L}{p} + \frac{m^2}{w} + \frac{m^2L}{p})$. Thus, we have,

**Theorem 8:** Algorithm MUL-HMM computes the products
of two matrices of size $m \times m$ ($w \leq m \leq n$) in $O(\frac{n^2}{w} +
\frac{m^2L}{p} + \frac{m^2}{w} + \frac{m^2L}{p})$ time units using $d$ ($\leq s^2$) DMMs
with $p$ ($\geq w$) threads each on the HMM with width $w$, global memory
latency $L$ and shared memory latency $l$.

Note that, in Algorithm MUL-HMM, the shared memory
of each DMM must store $O(m^2)$ elements.

**VI. LOWER BOUND OF MATRIX MULTIPLICATION**

Let us discuss the lower bound for Theorem 8. We first show
that, at least $\Omega(\frac{n^2}{w})$ read operations to the global memory are
necessary if the capacity of each shared memory is $O(m^2)$. We
can assume that each element in $C$ is computed in one of the
DMM, and evaluate the number of memory access destined for
the global memory; it may be possible that two or more DMMs
computes a particular element of $C$ partially, and the resulting
values are combined. However, such cooperative computation
does not decrease the total number of read operations to the
global memory necessary to compute a particular element.

It should be clear that, to compute each element $c[i, j]$, the
DMM computing $c[i, j]$ must read $n$ elements in the $i$-th row
of $A$ and those in $j$-th column of $B$. Hence, the DMM must read
$2n$ elements. Since the shared memory of the DMM can store
$O(m^2)$ elements, it can compute at most $O(m^2)$ elements in $c$
at the same time. Since we use big-$O$ notation and ignore the
constant factor, we can assume that DMM can compute at most
$m^2$ elements of $C$ at the same time. Suppose that, a DMM must
read $a$ rows of $A$ and $b$ rows of $B$ to compute $m^2$ elements
of $C$. Clearly, $m^2 \leq ab$ must be satisfied. Since the DMM
reads $a+b$ elements in the global memory, $a+b$ must be
minimized. From $a+b \geq 2\sqrt{ab} \geq 2m$, we have $a = b = m$
to minimize $a+b$. If a DMM computes a submatrix of size
$m \times m$, $a = b = m$ is satisfied. Hence, Algorithm MUL-
HMM performs the minimum memory read operations to the
global memory. Also, to compute $s^2$ submatrices, $s^2 \times 2n =
2m^2$ elements in the global memory must be read. Since at most
$w$ elements can be read from the global memory in a time unit,
at least $\Omega(\frac{n^2}{w})$ time units are necessary (the global memory
bandwidth limitation). Further, the HMM has totally $pd$ threads,
and each thread can send one memory access request to the
global memory in $L$ time units. Hence, to read $2m$ elements in the global memory, at least $\Omega(\frac{n^2}{w})$ time units are
necessary (the global memory latency limitation).

Next, let us discuss the limitation of shared memory access.
As we have discussed, at most $(d+1)w$ elements in $d$ shared
memory and one global memory can be read. Since at least $n^3$
read operations to $a$ (or $b$) must be performed to compute $c$,
at least $\frac{n^3}{(d+1)w} = \Omega(\frac{n^3}{d+1})$ time units are necessary
(the shared memory bandwidth limitation). Further, each of the $dp$
threads in the HMM can send at most one memory access request
to a shared memory in $L$ time units or to the global memory in $L$
time units. Since $n^3$ memory access requests must be sent to the
shared memories or the global memory and $l \ll L$, at least
\( \Omega \left( \frac{n^2 \mu \lambda}{dp} \right) \) time units are necessary (the shared memory latency limitation). Thus, we have,

**Theorem 9:** Any parallel implementation of straightforward matrix multiplication using \( d \) DMMs with \( p \) threads each in the HMM with width \( w \), global memory latency \( L \), and shared memory latency \( l \), takes at least \( \Omega \left( \frac{n^2}{mw} + \frac{n}{Ldp} + \frac{n}{lw} + \frac{n^3}{dp} \right) \) time units.

Hence, Algorithm MUL-HMM for Theorem 8 is time optimal. Note that, Theorem 9 is not the lower bound for matrix multiplication. It just shows the lower bound for implementations of the \( O(n^3) \)-time straightforward matrix multiplication algorithm. It is well-known that the product of two matrices of size \( n \times n \) can be computed in less than \( o(n^3) \) time by Strassen’s algorithm [23], [24], [25]. Hence, it is possible that matrix multiplication can be computed faster than the lower bound shown in Theorem 9, if the Strassen’s algorithm is implemented in the HMM. However, Strassen’s algorithm is complicated and has a large constant factor in the computing time, and it is not practically fast.

Similarly to image convolution, we discuss the conditions for hiding memory access latency. To hide the global memory latency, \( \frac{\mu}{nmw} \geq \frac{\mu L}{dp} \) that is, \( wL \leq dp \) must be satisfied. Similarly, to hide the shared memory latency, \( \frac{\lambda}{nw} \geq \frac{\lambda l}{dp} \) that is, \( wL \leq dp \) must be satisfied. Note that these conditions are the same as those for image convolution.

If both \( wL \leq dp \) and \( wL \leq dp \) are satisfied, Algorithm MUL-HMM runs \( O \left( \frac{n^3}{mw} + \frac{n^2}{dw} \right) \) time units. Intuitively, \( O \left( \frac{n^3}{mw} \right) \) and \( O \left( \frac{n^2}{dw} \right) \) correspond to the global memory access and the shared memory access, respectively. If \( \frac{\mu}{nw} \geq \frac{\mu L}{dp} \), that is \( m \leq d \), the time for global memory access dominates that for the shared memory access. Hence, we can think that the global memory access throughput in Algorithm MUL-HMM is maximized if \( m \leq d \), because Algorithm MUL-HMM runs \( O \left( \frac{n^3}{mw} \right) \) time units. On the other hand, if \( m > d \), Algorithm MUL-HMM runs \( O \left( \frac{n^2}{dw} \right) \). Recall that sequential algorithm runs in \( O(n^3) \) time. Also, \( w \) threads in a warp out of \( p \) DMM work in each thread unit and thus \( wL \) threads work in parallel. Hence, the acceleration ratio can be up to \( w \), and the running time of \( O \left( \frac{n^2}{dw} \right) \) attains optimal speed-up. Thus, we can say that the parallelism is maximized if \( m > d \).

**VII. EXPERIMENTAL RESULTS**

We have implemented our image convolution and matrix multiplication using CUDA C, and evaluated the performance on GeForce GTX TITAN. An algorithm for a DMM is implemented as a CUDA block. Hence, our implementation invokes \( d \) CUDA blocks to implement an algorithm on the HMM with \( d \) DMMs.

Figure 3 shows the computing time of image convolution of a \( 1024 \times 1024 \) image with respect to a \( 7 \times 7 \) kernel. Each pixel is a 4-byte float number. From the table, the computing time is almost the same for \( p = 512 \) and \( p = 1024 \) when \( d \geq 32 \). Hence, we can think that the global memory access latency is hidden when \( p = 512 \) and \( d \geq 32 \). Therefore, the number \( d \) of CUDA blocks must be at least 32 and the number \( p \) of threads per CUDA block must be at least 512 to maximize the performance. Figure 4 shows the computing time of image convolution when \( p = 1024 \). Clearly, the computing time is longer for larger \( v \). For each value of \( v \), the computing time is almost the same when \( d \geq 64 \). This is because the streaming multiprocessors on the GPU execute CUDA blocks in turn, if \( d \) is large.

Figure 5 shows the computing time for matrix multiplication of two \( 1024 \times 1024 \) float matrices. Similarly to image convolution, we can see that the computing time is almost the same for \( p = 256, 512 \) and \( 1024 \) when \( d \geq 32 \), because the global memory access latency is hidden. Figure 6 shows the computing time when \( p = 1024 \). Since the size of matrix is \( 1024 \times 1024 \), the number of submatrix is 256 when \( m = 64 \). Hence, the computing time is evaluated only for \( d \leq 256 \) CUDA blocks. We can see that the computing time is shorter for larger \( m \). However, the computing for \( m = 32 \) is not so different from that for \( m = 64 \), because the size of
submatrix is too large when \( m = 64 \). Each submatrix occupies \( 64 \times 64 \times 4 = 64\text{Kbytes} \) in the shared memory, and we need to store three matrices. Since the size of the shared memory is up to \( 48\text{Kbyte} \) [4], there is no extra space in the shared memory.

VIII. CONCLUSION

We have shown two parallel implementations for image convolution and matrix multiplications on the HMM, which is a theoretical parallel computing model for CUDA-enabled GPUs. We have proved that the running time of our implementations are time optimal. We also clarified the conditions for hiding memory access latency, and for maximizing global memory access throughput and parallelism. Finally, we have provided experimental results on GeForce GTX Titan to support our theoretical analysis.

REFERENCES


