2011 IEEE 17th International Conference on Parallel and Distributed Systems

Optimizing Dynamic Programming on Graphics Processing Units via Adaptive Thread-Level Parallelism

Chao-Chin Wu[1], Jenn-Yang Ke[2], Heshan Lin[3], and Wu-chun Feng[3]

[1]Department of Computer Science and Information Engineering, National Changhua University of Education
Changhua 500, Taiwan
ccwu@cc.ncue.edu.tw

[2]Department of Applied Mathematics, Tatung University, Taipei 104 Taiwan
jyke@ttu.edu.tw

[3]Department of Computer Science, Virginia Tech
Blacksburg, Virginia 24060 USA
{hlin2, wfeng}@vt.edu

***Abstract*—Dynamic programming (DP) is an important computational method for solving a wide variety of discrete optimization problems such as scheduling, string editing, packaging, and inventory management. In general, DP is classified into four categories based on the characteristics of the optimization equation. Because applications that are classified in the same category of DP have similar program behavior, the research community has sought to propose general solutions for parallelizing each category of DP. However, most existing studies focus on running DP on CPU-based parallel systems rather than on accelerating DP algorithms on the graphics processing unit (GPU).**

This paper presents the GPU acceleration of an important category of DP problems called *nonserial polyadic dynamic programming* (NPDP). In NPDP applications, the degree of parallelism varies significantly in different stages of computation, making it difficult to fully utilize the compute power of hundreds of processing cores in a GPU. To address this challenge, we propose a methodology that can adaptively adjust the thread-level parallelism in mapping a NPDP problem onto the GPU, thus providing sufficient and steady degrees of parallelism across different compute stages. We realize our approach in a real-world NPDP application – the *optimal matrix parenthesization* problem. Experimental results demonstrate our method can achieve a speedup of 13.40 over the previously published GPU algorithm.

Keywords-dynamic programming; GPU; parallel computing; parallelism; optimization

I. INTRODUCTION

Dynamic programming (DP) is a popular method used to solve complex problems, including scheduling, string editing, packaging, and inventory management [1]. Because it is believed that DP will remain important into the future for science and engineering, it is one of the Berkeley 13 dwarfs, where a dwarf is defined as an algorithmic method that captures a pattern of computation and communication for a class of applications [2]. The solution to a DP problem is usually expressed as a minimum (or maximum) of all possible alternate solutions. Dynamic programming can be classified into four categories based on the two criteria [1] below: (1) If solutions to problems at a phase depend only on solutions to problems at the previous level, the dynamic programming is called serial, otherwise it is called non-serial. (2) If the right-hand side of the optimization equation contains only one recursive term, the dynamic programming is called monadic. Otherwise, it is called polyadic. The categories of DP are then the four different combinations of the two above criteria: (1) serial-monadic, used in single-source shortest path and 0/1 knapsack problems, (2) non-serial-monadic, used in the longest common subsequence problem as well as the Smith-Waterman algorithm [3], (3) serial-polyadic, used in Floyd all-pairs, shortest-paths problem, and (4) non-serial-polyadic, used in the *optimal matrix parenthesization* problem [4, 5] and the Zuker algorithm [6, 7].

Recently, many efforts have investigated how to map the DP problems onto the emerging graphics processing unit (GPUs) [8-26]. Due to the wide variety of problems solved using DP, it is difficult to develop generic parallel algorithms for them on GPUs. In this work, we focus on the non-serial-polyadic DP problem. There are two distinct features that distinguish non-serial-polyadic DP from the other three DP problems [7]. First, the DP matrix for non-serial-polyadic DP is triangular, as opposed to being rectangular as in the other DP problems, where the DP matrix is used to show all data dependencies that occur during the computation. This property makes the optimization of memory accesses and load balancing difficult. Second, the data dependencies in non-serial-polyadic DP are dynamic, where data dependencies appear among nonconsecutive levels and the number of dependent elements varies for each subproblem.

In this paper, we investigate the mapping of non-serial-polyadic DP problems onto the GPU using the Optimal Matrix Parenthesization (OMP) problem as a case study. The OMP problem computes the optimal order of multiplying a chain of matrixes such that the total number of operations is the minimum. The state-of-the-art GPU implementation of the OMP problem [9] uses a static plan to map tasks onto GPU threads. More specifically, the computation is organized into different phases, each filling in one diagonal of the DP matrix, as shown in Figure 1. Each cell in the figure

1521-9097/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPADS.2011.92

represents a subproblem in the OMP problem, Cell(i, j) records the optimal cost for a single sub-problem of Matrix i through Matrix j. For each phase of the algorithm, each thread works to compute the optimal cost for a single sub-problem of length equal to the phase. Thus, the degree of parallelism decreases as the computation progresses. Also, as we will discuss in Section IV, the execution time of a sub-problem increases as the compute phase advances. Consequently, the fixed mapping algorithm can lead to significant underutilization of GPU resources in the later stages of computation.

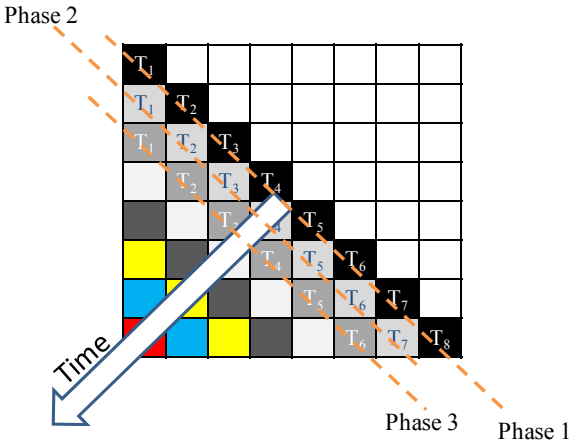

Figure 1. The execution flow of Solomon and Thulasiraman algorithm for solving the OMP problem with eight matrices. Each cell represents a subproblem, i.e., the solution of multiplying a subchain of matrices. T_i represents Thread i.

To address the irregularity of non-serial-polyadic DP problems, we propose an algorithm that can adaptively map subproblems to the GPU thread hierarchy according to the available parallelism in different compute phases. More specifically, we first introduce a parallel algorithm that allows mapping a subproblem to a flexible number of threads, which greatly improves the available parallelism that can be exploited by massive GPU processing cores. We then optimize the program performance by adaptively controlling the number of threads mapping to a subproblem based on the subproblem size and the dynamic of hardware utilization. Together with the adaptive task mapping design, our study also contributes a parallel minimum algorithm optimized by efficiently utilizing registers and shared memory on a GPU as well as an optimized data layout for triangular DP matrix that allows memory coalescing on GPUs.

Experimental results demonstrate our proposed algorithm garners substantial performance improvement over the state-of-the-art GPU implementation of the OMP problem. The best performance improvement is as high as 13.40.

II. Background and Related Work

CUDA is a new language and development environment that allows execution of general-purpose applications with thousands of data-parallel threads on NVIDIA's GPUs [27]. CUDA-based GPUs are Single-Instruction, Multiple-Threads (SIMT) architecture, i.e., the same instruction is executed simultaneously on many data elements by different threads. The hardware model is comprised of several highly threaded streaming multiprocessors (SMs), where each SM consists of a set of streaming processors (SPs). In the CUDA programming model, the computing system consists of a host that is a traditional CPU, also called *host*, and one or more GPUs, also called *device*.

A CUDA program consists of one or more phases that are executed on either the host or a device. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called kernels, and their associated data structures. Each kernel is launched by the host and is organized as a grid, which is a one or two dimensional array of thread blocks which in turn are one, two, or three dimensional arrays of threads. A thread block will be executed on a single SM, and an SM may hold multiple blocks executing concurrently if resources allow. All the threads of a grid execute the same kernel. The SM executes threads in groups of 32 parallel threads called warps. The threads in the same warp start together at the same program address but they are free to branch and execute independently. Each thread has its own register and private local memory. The threads in the same thread block can communicate with each other by means of shared memory. Threads in different thread blocks can share their results only through global memory.

In order to have a maximum bandwidth for the global memory, memory accesses from a kernel have to be coalesced. Coalescing global memory accesses might be the most important optimization technique for the CUDA architecture. On devices of compute capability 1.3 such as Tesla C1060, global memory loads and stores by threads of a half warp are coalesced by the device into as few as one transaction when the pattern of accesses fits into a segment size of 32 bytes for 8-bit words, 64 bytes for 16-bit words, or 128 bytes for 32- and 64-bit words. Moreover, there are two other read-only memory spaces, i.e., constant memory and the texture memory, which are accessible by all threads.

There are two methods proposed that mapped the non-serial polyadic DP problems onto GPUs [9, 11]. Solomon and Thulasiraman recently implemented an Optimal Matrix Parenthesization algorithm onto a NVIDIA GPU [9], which will be detailed in the next section.

Rizk et al. proposed a CUDA implementation of the Zuker algorithm in the RNA folding package Unafold [11]. They split and reorder the computations to enable tiled computations and good data reuse between threads. As a result, the computation of the optimal cost for every sub-problem requires two sequential kernels where a single thread is needed for each sub-problem. The two methods mentioned above suffer from the following two problems. First, the number of threads created at part of phases cannot provide sufficient parallelism to fully utilize the massive parallel compute power of a GPU because each subproblem is executed by a thread. Second, finding the optimal cost for each subproblem is computed by only one thread, resulting in long execution time. To address these two problems, we propose an algorithm that can adjust the number of threads dynamically to fully utilize all the compute powers in a GPU.

The algorithm proposed in this paper is orthogonal to the approach proposed by Rizk et al.

III. THE OPTIMAL MATRIX PARENTHESIZATION PROBLEM

A. The problem definition

The optimal matrix parenthesization problem, also known as the matrix chain multiplication problem or the matrix chain products problem, is to find the most efficient way to multiply a given sequence of matrices together rather than actually to perform the multiplications. There are many options of multiplying them because matrix multiplication is associative.

Formally, the optimal matrix parenthesization problem is defined as follows. Given a sequence of N matrices, (M_1, M_2, M_3,..., M_N), where each M_i is a matrix with r_{i-1} rows and r_i columns. The optimal cost, C_{ij}, for a subsequence of matrices, (M_i, M_{i+1}, …, M_j), is computed by the following formula:

$$C_{i,j} = min_{i \le k < j}\{C_{i,k}+C_{k+1,j}+r_i \times r_k \times r_j\},$$

where $1 \le i, j, k \le N$ and $C_{ij} = 0$. The optimal cost of the sequence of N matrices is derived when $i = 1$ and $j = N$, i.e., $C_{1,N}$. The optimal matrix parenthesization problem has been studied for different architectures previously [4, 5].

B. The Solomon and Thulasiraman algorithm

Solomon and Thulasiraman are the first and the only one that adopted GPUs to solve the optimal Matrix Parenthesization problem [9]. Their algorithm consists of N sequential phases and each phase will launch a kernel to solve the subproblems at a phase. At Phase P, the algorithm finds the minimum cost for each subsequence of P matrices, where $1 \le P \le N$. Therefore, (N-P+1) threads will be activated at phase P, where $1 \le P \le N$. In general, at phase P, Thread_{tid} is responsible for finding the optimal way to parenthesize the matrix subsequence (M_{tid+1}, M_{tid+2}, M_{tid+3}, …, M_{tid+P}), where $0 \le tid \le N$-P. For each matrix subsequence at Phase P, there are P-1 possible positions at which the subsequence can be split: cutting it right before M_k, where $(tid+1) \le k \le (tid+P-1)$ and the result matrix multiplication is:

$$(M_{tid+1} * M_{tid+2} * \ldots * M_{tid+k}) * (M_{tid+k+1} * \ldots * M_{tid+P}).$$

To calculate the cost of every possible splitting way, Thread_{tid} looks up the minimum cost of multiplying out each of the two subsequences, derived in previous phases, adds up the two costs and the cost of multiplying the two result matrices. Finally, Thread_{tid} takes the minimum over all costs of the possible splitting ways. This paper focuses on how to improve the Solomon and Thulasiraman algorithm.

C. Observation

We analyze the algorithm of the optimal matrix parenthesization problem proposed by Solomon and Thulasiraman as follows. At the first phase, the number of threads activated is equal to the length of the matrix sequence. When the computation proceeds to the next phase, the number of active threads is decreased by one. Eventually, there is only one active thread remaining at the final phase. For instance, if there are eight matrices in the sequence as shown in Figure 1, eight phases are required to solve this problem. The number of parallel threads decreases by one at every phase, from eight threads at Phase 1 down to only one thread in Phase 8.

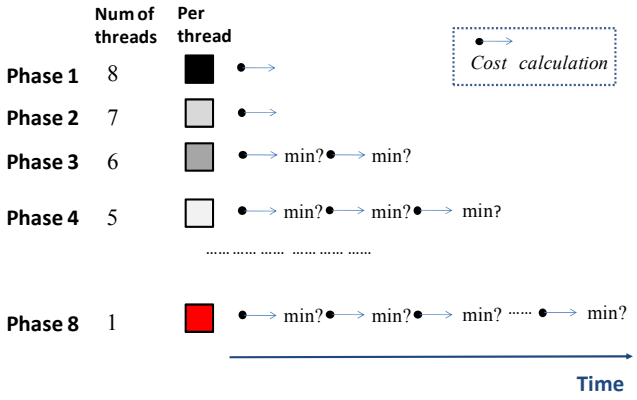

Figure 2. The way each thread in Figure 1 finds the minimum cost sequentially in Solomon and Thulasiraman algorithm.

Although the number of threads lowers by one at every phase, the subsequence length increases by one at every phase, resulting in more lookups and comparisons for generating an optimal cost for each subproblem. During each phase, each thread has to repeatedly perform the following two steps for each possible splitting ways as shown in Figure 2: (1) calculate the cost of multiplying the result matrices, and (2) replace the minimum cost with the newly generated cost for the possible splitting way if the new cost is less than the current minimum cost. At phase 1, there are eight parallel threads, each calculates one cost of the assigned subproblem. At phase 8, there is only one thread; the thread uses a sequential algorithm to find the minimum cost from seven possible costs. As only one single thread performs all of the work required for every sub-problem, the time taken per thread will increase when the number of threads decreases after we proceed to the next phase. According to the evaluation results reported in [9], the time taken to complete a phase increases gradually during the first half of the phases but it decreases during the latter half of the phases. For the first half of the phases the amount of work per thread increases faster than the problem space decreases, hence the increase in time to complete each phase. In the latter half of the algorithm, the reduced number of threads impacts the runtime in greater magnitude than the increased work per thread does. However, the runtime per phase in the latter half of the algorithm still notably remains significantly greater than that of the first half of the algorithm.

The algorithm proposed by Solomon and Thulasiraman suffers from the following problems. First, the maximum number of threads is limited by the matrix sequence length. Second, the number of threads decreases by one at every phase, resulting in decreased parallelism because of fewer and fewer threads in the latter phases. Third, the amount of work per thread increases whenever the algorithm proceeds to next phases. To address these problems, we propose an

algorithm in the next section to exploit more parallelism that can fully utilize all of the GPU compute power.

IV. Our Solution: Improving GPU Utilization with Adaptive Parallelism Exploitation

To address the problem of the irregular parallelism distribution among phases in the optimal matrix parenthesization problem, we propose an algorithm called Adaptive Thread Mapping (ATM) algorithm, which can dynamically and intelligently adjust the number of threads for different phases to fully utilize all of the GPU compute power.

A. *Exploiting thread parallelism with fine-grained task mapping*

As discussed in Section IV-C, mapping one subproblem to a thread can easily lead to resource underutilization when there is not sufficient number of subproblems. To address this issue, we first introduce an efficient algorithm that maps a subproblem to a thread block, which increases the parallelism that can be exploited. Specifically, a block of threads are used to calculate the costs of all the possible splitting ways, with one thread for calculating the cost of one possible splitting way, and find the minimum over all the possible costs in parallel. In other words, each thread in Solomon and Thulasiraman algorithm is split into multiple subthreads in the ATM. For clarity, 'subthread' is used for the ATM and 'thread' is used for Solomon and Thulasiraman algorithm hereafter. All the parallel subthreads in the same thread block firstly calculate the possible costs and then find the minimum over all possible costs in parallel.

For instance as shown in Figure 3, if there are eight matrices in the sequence, the number of thread blocks required in each phase decreases by one at every phase, resulting in eight thread blocks in Phase 1 while one thread block in Phase 8. The length of a matrix subsequence is increased by one when the algorithm proceeds to next phase, resulting in one more possible way of splitting to be considered. Consequently, the total number of subthreads in a thread block is increased by one when the algorithm proceeds to next stage. For instance, there is one subthread per thread block at Phase 1 while seven parallel subthreads per thread block at Phase 8, as shown in Figure 4. The total number of parallel subthreads is equal to the number of thread blocks multiplying the number of subthreads in a thread block. The maximum number of parallel subthreads in a phase is 16, which is occurred at Phase 5.

Furthermore, we adopted a parallel minimum algorithm, similar to the parallel reduction algorithm [28], to accelerate the search of the minimum over all costs inside each thread block. For instance, at Phase 8, seven subthreads will run the parallel minimum algorithm together to find the minimum cost, as shown in Figure 4. However, we need to modify the parallel minimum algorithm because there are some different features in the ATM. First, because the number of subthreads per thread block increases by one when it proceeds to the next phase, we have to modify the algorithm to be able to deal with arbitrary lengths of matrix subsequences, especially for non-power-of-2 numbers. Second, there are multiple thread blocks at every phase and each thread block has to find its local minimum. That is, there are multiple independent minimums to be searched in each thread block at each phase. Therefore, each subthread in the same thread block will save the derived cost in shared memory before performing the parallel minimum algorithm with other subthreads in the same thread block, which can avoid inter-thread block synchronization and thus accelerate the parallel minimum algorithm. Third, the number of parallel subthreads might be larger than the system-defined maximum number of subthreads per thread block. To address the problem, we propose to solve several subproblems in parallel by a block of threads to improve resource utilization.

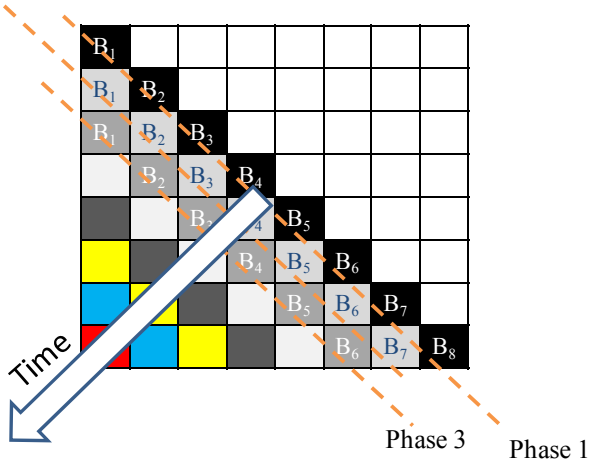

Figure 3. The execution flow of our algorithm. Each matrix cell, i.e., matrix subsequence, is solved by an independent thread block. B_i represents Thread Block i, which consists of many parallel subthreads.

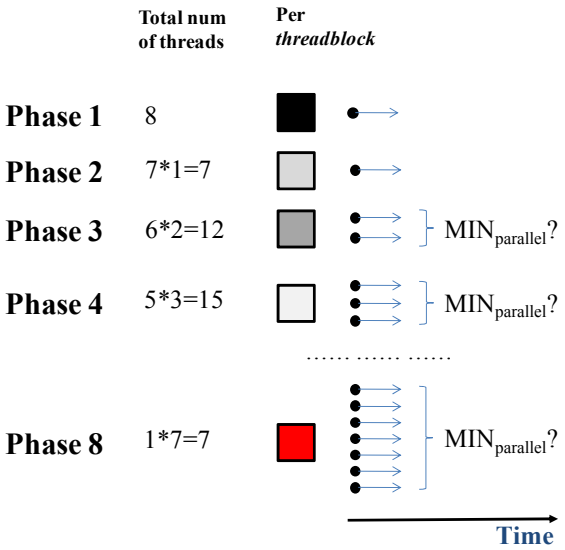

Figure 4. The way each thread block in Figure 3 uses multiple threads to find the minimum cost in parallel in our algorithm.

Figure 5 compares two different methods of executing the parallel minimum algorithm for finding the two minimum costs of two matrix subsequences by a thread block.

Assume the lengths of the two matrix subsequences are both 9 and the maximum number of parallel threads in a thread block is 4. The first method is to find the two minimum costs one at a time, as shown in Figure 5(a). To find a minimum, four threads are active at the first step; however, two and one thread are active at the second and the third steps, respectively. The utilization ratio is (4+2+1)/(4+4+4), i.e., 0.5833, and the total number of steps for solving the two subproblems is 6. On the other hand, the second method is to find the two minimum costs in parallel: two threads are assigned to find the minimum in parallel for a matrix subsequence, as shown in Figure 5(b). For the first three steps, four threads are active totally; for the last step, two threads are active. The utilization ratio for the second method is (4+4+4+2)/(4+4+4+4) , i.e., 0.875, and the total number of steps for solving the two subproblems is 4. Therefore, we will adopt the second method because of its better resource utilization ratio and smaller number of execution steps.

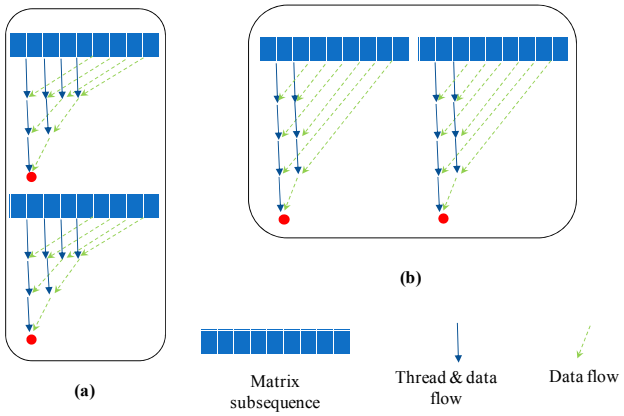

Figure 5. Two different methods of executing the parallel minimum algorithm for finding the minimum costs of two matrix subsequences.

B. *Two-stage adaptive thread mapping*

Mapping a subproblem to a thread block can introduce considerable overhead because more than half of the streaming processors become idle during the course of finding the minimum in parallel. To optimize the GPU resource utilization, we need to adaptively choosing whether to map a subproblem to a thread or a thread block according to the available parallelism during a compute phase.

The ATM algorithm divides the whole execution of the optimal matrix parenthesization problem into two stages: the total number of threads divided by the number of streaming multiprocessors is less than or larger than the maximal number of threads supported in a thread block. For the first stage, each thread is split into multiple subthreads as many as possible if the limitation of the maximal number of threads in a thread block is satisfied. As for the second stage, several subproblems will be solved by a block of threads, where each thread in this stage is called superthread because each thread has to find the minimum among several costs before performing the parallel minimum algorithm for a subproblem.

Formally, assume the number of streaming multiprocessors on a GPU is N_{SM}, the maximal number of threads allowed in a thread block is MNT_B, and the total number of threads at Phase P is NT_P. If NT_P is larger than or equal to N_{SM}, we will create N_{SM} thread blocks. Otherwise, only NT_P thread blocks will be created. If the number of thread blocks created is N_B, each thread block should be allocated NT_P/N_B threads. However, if NT_P/N_B is larger than MNT_B, we have to merge NT_P/N_B threads in the same block into MNT_B superthreads to satisfy the limitation of the maximal number of threads in a thread block. Each superthread will find NT_P/N_B minimums one by one.

On the other hand, if NT_P/N_B is smaller than MNT_B, we will split NT_P/N_B threads in the same block into subthreads as many as possible as long as the limitation of the maximal number of threads in a thread block is satisfied. In general, $(MNT_B /(NT_P/N_B))$ subthreads will be created for each original thread in the same block. However, since there are $(P\text{-}1)$ possible costs to be computed in each thread at Phase P, $(P\text{-}1)$ threads will be created if $(MNT_B /(NT_P/N_B)) > (P\text{-}1)$, where $P > 1$. The number of subthreads to be split for each thread is denoted as ST_s. Since there are NT_P/N_B threads in the same thread block, (NT_P/N_B) minimums will be calculated by each single stream multiprocessor. The ST_s subthreads belonging to the same thread are grouped together by having consecutive *thread id*s. Therefore, there are (NT_P/N_B) groups of subthreads in each thread block. Each group of subthreads will perform the same parallel minimum program collaboratively to find a minimum. If ST_s is equal to $(P\text{-}1)$, each subthread calculates a cost and then participates in the parallel minimum algorithm. Otherwise, each subthread will be assigned $((P\text{-}1)/ST_s)$ matrix subsequences and perform a two-step operation as follows. Similarly, the ST_s subthreads belonging to the same thread are grouped together by having consecutive *thread id*s. First, each subthread will calculate the costs of the assigned $((P\text{-}1)/ST_s)$ subsequences and find the local minimum among the derived costs, where the local minimum will be stored in shared memory. Second, all the ST_s subthreads in the same group will participate in the same parallel minimum algorithm to find the minimum among the ST_s local minimums.

C. *Coalesced memory access*

Memory coalescing is one of the key technique to optimize the algorithm on nVIDIA GPUs, which maximizes global memory bandwidth by minimizing the number of bus transactions. In best cases, one transaction will be issued for a half-warp to access a contiguous region of global memory.

We give an example to show the original algorithm proposed by Solomon and Thulasiraman does not apply the memory coalescing optimization technique. The data structure of Type 0 shown in Figure 6 is used to store the minimum cost for every matrix subsequence in their algorithm, where the integer pair in each matrix cell of the data structure represents the *start id* and the *end id* of the matrix subsequence. For instance, "2, 6" in Row 5 and Column 2 in the data structure represents the cell is used to store the minimum of the matrix subsequence: $(M_2, M_3, M_4, M_5, M_6)$, denoted as $(M_2 .. M_6)$. At Phase 5, we have to calculate four

different subsequences: (M_1 .. M_5), (M_2 .. M_6), (M_3 .. M_7), and (M_4 .. M_8). We use the same color to specify which matrix subsequence subproblem needs which costs derived in previous phases. For instance, (M_1 .. M_5) needs the minimums of (M_1 .. M_1), (M_1 .. M_2), (M_1 .. M_3), (M_1 .. M_4), (M_2 .. M_5), (M_3 .. M_5), (M_4 .. M_5), and (M_5 .. M_5). Because each matrix subsequence is solved by a thread, totally four threads in the same warp are required to solve the subproblems in parallel. At the beginning, the four threads have to access in parallel the minimums of "1,1","2,2", "3,3", and "4,4", respectively. However, these four cells are not in a contiguous region. That is, these four cells cannot be fetched by a single bus transaction. Instead, four bus transactions are required. Similar behaviors also happen when the four threads accesses other minimums in parallel.

To apply the memory coalescing optimization technique, we transform the data structure to Type 1, as shown in Figure 6, where all the minimums in the same column are shifted upward to start from the first row. In this way, the minimums of "1,1","2,2", "3,3", and "4,4" are all stored in Row 0 and then can be accessed in parallel by a single bus transaction. For other every four minimums accessed in parallel are also in the same row although some do not start from Column 0. For example, the four threads need to access to "2,5","3,6", "4,7", and "5,8", respectively, at the same step; these four costs are stored in Row 3 and can be accessed by a single bus transaction although these data are stored from Column 1.

For our proposed ATM, it is more complicated to apply the memory coalescing technique because subthreads and superthreads have different access patterns. For subthreads, we cannot use Type 1 data structure to access memory coalesced because of the following reason. In this case, each thread is divided into multiple subthreads that are with consecutive thread *id*s and will use contiguous shared memory elements to find the minimum in parallel. Consequently, they are subthreads, rather than threads, in the same half warp. For instance, the eight minimums required for solving the matrix subsequence (M_1 .. M_5) will be accessed by four subthreads, as shown in Figure 7(a): "1,1","1,2", "1,3", and "1,4" will be accessed in parallel at the first step, and then "2,5","3,5", "4,5", and "5,5" will be accessed in parallel at the second step. However, "1,1","1,2", "1,3", and "1,4" are stored in the same column, not the same row. Similarly, "2,5","3,5", "4,5", and "5,5" are not stored in the same row. Therefore, Type 1 is not a suitable data structure for parallel subthreads to have coalesced memory access.

To take the performance advantage of memory coalescing, we transform the data structure to Type 2, as shown in Figure 7(b). Type 2 is the matrix transpose of Type 1. When parallel threads use Type 2 to access data, the minimums of "1,1","1,2", "1,3", and "1,4" are in the same row although the minimums of "2,5","3,5", "4,5", and "5,5" are still not in the same row. It is impossible to arrange the two groups of four minimums to let each group be in the same row. That is, we can only optimize half of the memory accesses by the memory coalescing technique for subthreads.

As for superthread, we should adopt Type 1 because superthreads have the similar behavior of memory access to that of threads in the algorithm proposed by Solomon and Thulasiraman. Because the phases in the second stage are much more time consuming, Type 1 should be better for the ATM algorithm.

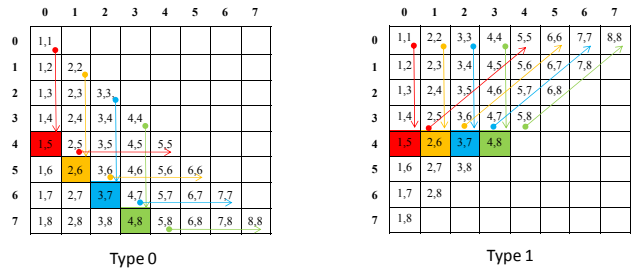

Figure 6. The first two types of data structure and how data elements are accessed in parallel by multiple threads in parallel. Different colored arrows represent the data required by different subproblems.

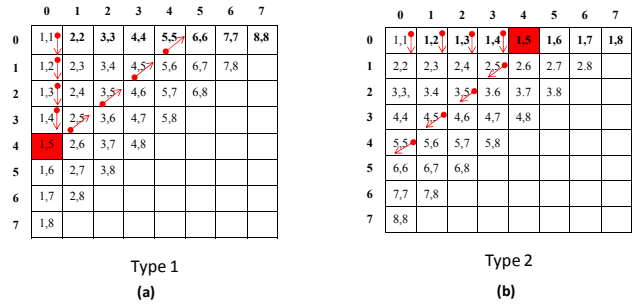

Figure 7. Types 1 and 2 of data structure as well as how data elemets are accessed in parallel by multiple subthreads in parallel. Only one subproblem is depicted for showing the corresponding data accesses.

V. Performance Evaluations

We use CUDA version 3.2 to implement different algorithms for solving the Optimal Matrix Parenthesization problem. The Dell Precision T5500 computer workstation is used as an experimental platform to evaluate our proposed algorithms, which consists of one Intel Xeon CPU and one nVIDIA Tesla C1060 GPU primarily. The compute capability of Tesla C1060 is 1.3. The Operating System installed is Linux and its version is openSUSE 11.2 32-bit. The main hardware configurations of Dell Precision T5500 are listed in Table I.

A. Optimization by the ATM

We compare the performances of the ATM and the Solomon and Thulasiraman's algorithm in Figures 8 and 9, where 'Solomon' and 'Solomon-MC' denote the Solomon and Thulasiraman's algorithm without and with memory colaescing. In the experiments, the maximum number of thread blocks used is 30 and the data structure for memory coalescing is type 1 as explained in Section V-C. In addition, to study the impact of from which phase we should start to split the threads, we conducted two different experiments that start to split threads. First, we start the thread splitting from Phase 512 in the ATM ('ATM-MC(512)' in the legend). For the length of the matrix sequence less than or equal to 512, the ATM-MC(512) has no chance to split threads but outperforms the Solomon and Solomon-MC. The perfor-

mance improvement is obtained by limiting the maximum number of thread blocks to 30, compared with the Solomon-MC that sets maximum number of thread blocks to 64.

TABLE I. THE HARDWARE CONCFIGURATIONS OF THE INTEL CPU AND THE NVIDIA TESLA C1060

Intel® Xeon® Processor E5504		nVIDIA C1060 Computing Processor	
# of Cores	4	# of GPUs	1
# of Threads	4	Thread Processors	240
Clock Speed	2 GHz	Clock Speed	1300MHz
Memory size	6GB	Memory size	4GB
Memory Types	DDR3 800	Memory Type	GDDR3
# of Memory Channels	3	Memory Clock	800MHz
Max Memory Bandwidth	19.2 GB/s	Max Memory Bandwidth	102.4 GB/s
Cache	4 MB	System interface	PCIe 2.0 x16

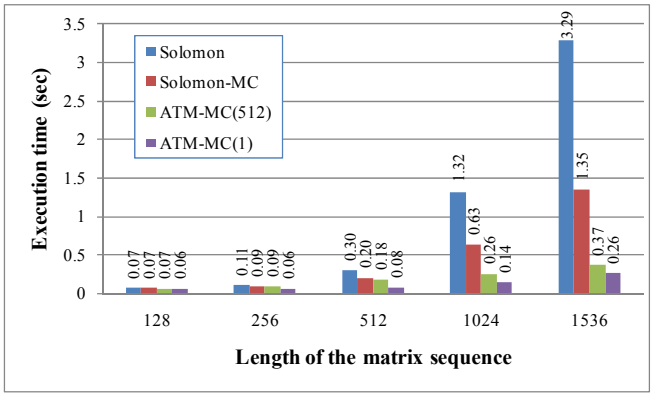

Figure 8. Performance comparision of the ATM for shorter matrix sequence lengths.

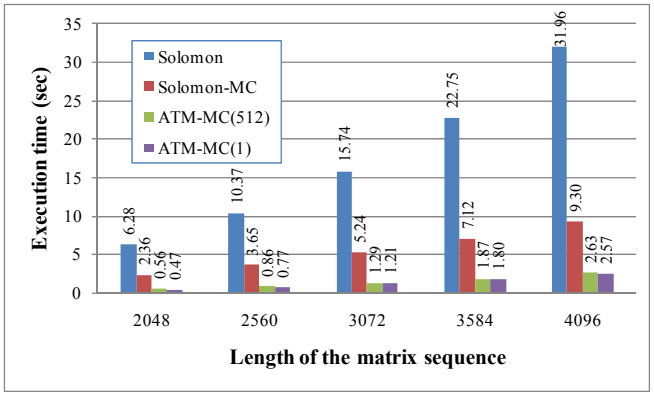

Figure 9. Performance comparision of the ATM for longer matrix sequence lengths.

Second, we start the thread splitting from Phase 1 in the ATM ('ATM-MC(1)' in the legend). The ATM-MC(1) outperforms the ATM-MC(512) for all different lengths of the matrix sequences. For those problems with the sequence lengths less or equal to 512, the ATM-MC(1) parallelizes all the findings of the minimums, compared with the ATM-MC(512). Therefore, the ATM-MC(1) can reduce the execution significantly for the small problem sizes. On the other hand, the difference between the ATM-MC(1) and the ATM-MC(512) is only for the first 512 phases. When the problem size increases, the ratio of 512 to the sequence length becomes smaller, resulting in less performance improvement. Based on the above discussion, it is better to split threads from the very beginning. The ATM-MC(1) can provide the speedup from 1.12 to 5.16 over the Solomon-MC, or from 1.18 to 13.40 over the original Solomon and Thulasiraman's algorithm.

B. *Impact of when to start splitting*

In this subsection, we investigate on the impact of when to start splitting threads on performance for different matrix sequence sizes in the ATM algorithm, as shown in Table II. For the smaller matrix sequence sizes, it is better to split threads from Phase 128 although the performance differences are small. For the larger sizes, it is better to split threads from Phase 256 or 1024. Because each of the latter phases is more time consuming, if threads are divided too late, the execution time increases significantly. Interestingly, if threads are split from Phase 1, the execution time will not be increased. Therefore, we can always get the best performance no matter how large the matrix sequence size is if we start to split threads from Phase 1.

TABLE II. THE IMPACT OF WHEN TO START SPLITING THREADS ON EXECUTION TIMES FOR DIFFERENT MATRIX SEQUENCE SIZES IN THE SPLIT-LIMIT ALGORITHM

Starting Phase of Thread Splitting \ Matrix Sequence Size	128	256	512	1024	1536	2048	2560	3072	3584	4096
1	0.06	0.07	0.08	0.14	0.26	0.47	0.77	1.21	1.80	2.57
32	0.06	0.07	0.08	0.14	0.26	0.47	0.78	1.22	1.80	2.57
64	0.06	0.07	0.08	0.14	0.26	0.47	0.77	1.21	1.80	2.57
128	0.07	0.07	0.09	0.15	0.27	0.47	0.78	1.21	1.80	2.57
256		0.09	0.11	0.17	0.29	0.49	0.79	1.23	1.82	2.58
512			0.19	0.26	0.37	0.56	0.86	1.29	1.87	2.63
1024				0.59	0.72	0.88	1.15	1.55	2.10	2.83
1536					1.27	1.46	1.68	2.03	2.54	3.21
2048						2.25	2.50	2.78	3.21	3.81
2560							3.51	3.83	4.16	4.69
3072								5.07	5.46	5.85
3584									6.92	7.37
4096										9.06

VI. CONCLUSION

This paper focused on how to use CUDA to optimize the implementation of the category of NPDP models on a NVIDIA GPU, where the optimal matrix parenthesization problem is taken as an example. To fully utilize the compute powers of a GPU, we propose to exploit more thread parallelisms. Instead of one thread for solving each subproblem, our algorithm splits a thread into multiple subthreads to execute the parallel minimum algorithm for finding the minimum for each subproblem. However, creating too many subthreads for finding one minimum will lead to too many idle SPs during the execution of the parallel minimum algorithm. We have to adjust the number of subthreads for each subproblem intelligently such that not only SPs are busy as many as possible but also subproblems are solved simultaneously as many as possible. Therefore, our algorithm de-

fines a threshold helping determine when to solve multiple subproblems by a block of thread.

Because our algorithm adjusts the number of threads at each phase dynamically, the memory access pattern varies in different phases. Consequently, we just cannot adopt a single form of data layout to achieve coalesced memory access for all phases. Two different forms of data layout are proposed and investigated to see which is more suitable for our algorithm. Experimental results demonstrate our algorithm outperforms the previous algorithm significantly, with the best speedup of 13.40.

In the future, we will study how to optimize the algorithm further by different techniques. Moreover, we will investigate how to implement the algorithm on multi-GPU platform and how to optimize the algorithm for AMD and NVIDIA GPUs written in the OpenCL language.

ACKNOWLEDGMENT

The authors would like to thank the National Science Council, Taiwan, for financially supporting this research under Contract No. NSC99-2918-I-018-001.

REFERENCES

[1] A. Grama, A. Gupta, G. Karypis, V. Kumar, Introduction to Parallel Computing, Second Edition, Addison-Wesley, 2003.

[2] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick, The landscape of parallel computing research: a view from Berkeley, Technical Report No. UCB/EECS-2006-183, 2006.

[3] T. Smith, M. Waterman, Identication of common molecular subsequences Journal of Molecular Biology 147(1) (1981) 195-197.

[4] M. Hafeez, M. Younus, An effective solution for matrix parenthesization problem through parallelisation, International Journal of Computers 1(1) (2007) 1-9.

[5] H. Lee, J. Kim, S.J. Hong, S. Lee, Processor allocation and task scheduling of matrix chain products on parallel systems, IEEE Transactions on Parallel and Distributed Systems 14(4) (2003) 394-407.

[6] R.B. Lyngso, M. Zuker, Fast evaluation of internal loops in RNA secondary structure prediction, Bioinformatics 15(6) (1999) 440-445.

[7] G. Tan, N. Sun, G.R. Gao, Improving performance of dynamic programming via parallelism and locality on multicore architectures, IEEE Transactions on Parallel and Distributed Systems 20 (2) (2009) 261-274.

[8] V. Boyer, D. El Baz, M. Elkihel, Dense dynamic programming on multi GPU, in: Proceedings of the 19th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 2011, pp. 545 - 551.

[9] S. Solomon, P. Thulasiraman, Performance study of mapping irregular computations on GPUs, in: IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, 2010, pp. 1 – 8.

[10] N. Deshmukh, H. Rivaz, P. Stolka, H.-J. Kang, G. Hager, M. Alaf, E. Boctor, Real-time GPU-based analytic minimization/dynamic programming elastography, In: Proceedings of the 2nd International Workshop on High-Performance Medical Image Computing for Image-Assisted Clinical Intervention and Decision-Making, 2010.

[11] G. Rizk, D. Lavenier, GPU accelerated RNA folding algorithm, in: the 9th International Conference on Computational Science: Part I, 2009, pp. 1014 – 1023.

[12] P. Steffen, R. Giegerich, M. Giraud, GPU parallelization of algebraic dynamic programming, in: Proc. of the 8th International Conference on Parallel Processing and Applied Mathematics: Part II, 2009, pp. 290-299.

[13] C.-H. Sin, C.-M. Cheng, S.-H. Lai, S.-Y. Yang, Geodesic tree-based dynamic programming for fast stereo reconstruction, in: IEEE 12th International Conference on Computer Vision Workshops, 2009, pp. 801 – 807.

[14] J. Congote, J. Barandiaran, I. Barandiaran, O. Ruiz, Realtime dense stereo matching with dynamic programming in CUDA, in: XIX Spanish Congress of Graphical Informatics, 2009, pp. 231-234.

[15] Y. Liu, B. Schmidt, D. L. Maskell, CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD Abstractions, BMC Research Notes, 3(1) (2010) 93.

[16] K. Dohi, K. Benkridt, C. Ling, T. Hamada, Y. Shibata, Highly efficient mapping of the Smith-Waterman algorithm on CUDA-compatible GPUs, in: Proc. 21st IEEE International Conference on Application-specific Systems Architectures and Processors, 2010, pp. 29 – 36.

[17] D. Razmyslovich, G. Marcus, M. Gipp, M. Zapatka, A. Szillus, Implementation of Smith-Waterman algorithm in OpenCL for GPUs, in: Second International Workshop on High Performance Computational Systems Biology, 2010, pp. 48 – 56.

[18] S. Xiao, A.M. Aji, W. Feng, On the robust mapping of dynamic programming onto a graphics processing unit, In: 15th International Conference on Parallel and Distributed Systems, 2009, pp. 26 – 33.

[19] L. Ligowski, W. Rudnicki, An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases, in: IEEE International Symposium on Parallel and Distributed Processing, 2009.

[20] C. Ling, K. Benkrid, T. Hamada, A parameterisable and scalable Smith-Waterman algorithm implementation on CUDA-compatible GPUs, in: IEEE 7th Symposium on Application Specific Processors, 2009, pp. 94–100.

[21] G.M. Striemer and A. Akoglu, Sequence alignment with GPU: performance and design challenges, in IPDPS, 2009.

[22] S.A. Manavski, G. Valle, CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment, BMC Bioinformatics 9(Suppl 2) (2008) S10.

[23] Y. Munekawa, F. Ino, K. Hagihara, Design and implementation of the Smith-Waterman algorithm on the CUDA-compatible GPU, in: 8th IEEE International Conference on BioInformatics and BioEngineering, 2008, pp. 1-6.

[24] W. Liu, B. Schmidt, G. Voss, W. Muller-Wittig, Streaming algorithms for biological sequence alignment on GPUs, IEEE Transactions on Parallel and Distributed Systems 18 (2007), 1270-1281.

[25] Y. Liu, W. Huang, J. Johnson, S. Vaidya, GPU accelerated Smith-Waterman, in: Proc. Int'l Conf. Computational Science, 2006, pp. 188-195.

[26] J. Nickolls, W.J. Dally, The GPU computing era, IEEE Micro 30(2) (2010) 56–69.

[27] The CUDA Zone, http://www.nvidia.com/object/cuda_home_new.html

[28] M. Harris, S. Sengupta, J.D. Owens, Parallel prefix sum (scan) with CUDA, GPU Gems 3 (2007)