# A Parallelization of Non-Serial Polyadic Dynamic Programming on GPU

Tausif Diwan and Jitendra Tembhurne

Indian Institute of Information Technology, Nagpur, India

Parallelization of Non-Serial Polyadic Dynamic Programming (NPDP) on high-throughput manycore architectures, such as NVIDIA GPUs, suffers from load imbalance, *i.e.* non-optimal mapping between the sub-problems of NPDP and the processing elements of the GPU. NPDP exhibits non-uniformity in the number of subproblems as well as computational complexity across the phases. In NPDP parallelization, phases are computed sequentially whereas subproblems of each phase are computed concurrently. Therefore, it is essential to effectively map the subproblems of each phase to the processing elements while implementing thread level parallelism. We propose an adaptive Generalized Mapping Method (GMM) for NPDP parallelization that utilizes the GPU for efficient mapping of subproblems onto processing threads in each phase. Input-size and targeted GPU decide the computing power and the best mapping for each phase in NPDP parallelization. The performance of GMM is compared with different conventional parallelization approaches. For sufficiently large inputs, our technique outperforms the state-of-the-art conventional parallelization approach and achieves a significant speedup of a factor 30. We also summarize the general heuristics for achieving better gain in the NPDP parallelization.

*ACM CCS (2012) Classification:* Computing methodologies → Concurrent computing methodologies → Concurrent programming languages

Computing methodologies → Parallel computing methodologies → Parallel algorithms → Massively parallel algorithms

Computing methodologies → Parallel computing methodologies → Parallel algorithms → Vector / streaming algorithms

*Keywords*: dynamic programming, parallel computing, GPU, CUDA, NPDP

## 1. Introduction

NPDP is the most complex class of Dynamic Programming (DP) with dependencies across non-consecutive phases and more than one dependent term in the characteristics equation by [1]. These dependencies are easily identified in Matrix Chain Multiplication (MCM), optimal binary search tree, optimal triangulation problem, and Zukar's algorithm. Uniform reduction in the number of sub-problems and uniform increase in sub-problem computation leads to non-uniformity in the total computation across NPDP phases. In any DP algorithm, computation proceeds phase-wise. In NPDP, subproblems of each phase are dependent on the subproblems of the previous phases. After successful completion of a phase, computation of the subsequent phase is started. However, sub-problems of a phase are independent of each other and can be computed in parallel. The dependencies across the non-consecutive phases have limited the maximal utilization of the underlying high performance architecture in parallelizing the NPDP algorithms.

In NPDP parallelization on multi-core CPU using any parallel API such as OpenMP, optimal mapping and effective utilization of cores with equal workload can be achieved very easily by choosing an appropriate scheduling technique. Due to the limited number of processing cores on multi-core architectures, many-to-one mapping between the sub-problems and the processing threads does not affect the performance beyond a tolerance limit. In OpenMP, number of sub-problems assigned to a thread is defined by chunk-size parameter and a thread executing those chunks onto physical processing core is taken care of by scheduling policies of OpenMP, as illustrated in [2], [3], [4], [5]. However, on many-core GPUs, many-to-one mapping between the sub-problems of a phase and the processing threads fails to sustain the maximal utilization for each phase due to the huge threading capacity of the GPU. In this paper, mapping is used as an abbreviation for the "mapping between the sub-problems of a phase of NPDP and the processing threads of the GPU".

We propose a novel MCM parallelization approach using Compute Unified Device Architecture (CUDA) on NVIDIA Quadro K6000 GPU. We follow our predecessors in NPDP parallelization, *i.e.* we parallelize one phase of NPDP at a time. After the successful parallelization of one phase, parallel computation of the subsequent phase is started. In addition, we employ different computing power for each phase of parallelization for the optimal mapping and efficient utilization of the GPU resources. CUDA selects, at runtime, the appropriate mapping for each phase, depending upon the computing resources.

In the initial phases of NPDP computation, the number of sub-problems is very large; many-to-one mapping is convenient for this region. In middle phases, number of sub-problems is almost under the threading capacity of the GPU for large size input; one-to-one mapping is an effective approach in the direction of keeping all the SMs busy. In later phases, when the computations of one sub-problem are not limited by several comparisons, one-to-one mapping leads to under-utilization of the GPU resources. The large volume of data is also required for the subproblems computation in the later phases.

We broadly classify the NPDP parallelization in two categories viz. Conventional Mapping (CM) approaches and GMM. Under CM approaches, we implement, compare, and analyze MCM parallelization using four different conventional approaches suited for the same mapping throughout all the phases of NPDP. None of these approaches is suited for the entire NPDP computation. Proposed GMM chooses appropriate mapping for each region of phases in the direction of optimal utilization of hardware resources, *i.e.* to keep maximum SMs busy. For sufficiently large input, we achieve better speedup using GMM in comparisons with the best CM parallelization approach. The Amdahl's law state about the speedup dependency over the fraction of the code being parallelized. However, the speedup is computed in this paper as: $speedup = t_s / t_p$, where $t_s$ is time of serial execution and $t_p$ represents the time taken by the parallel execution. We generalize the MCM parallelization results for other NPDP problems.

## 2. Literature Review

Parallelization of DP on multi-core architectures and related issues has been widely discussed in [6], [7], [8]. This includes category-wise parallelization of DP, effective utilization of multi-core cache memory, dependence analysis, and dependency transformations. Parallelization of DP having more than $O(1)$ data dependency is studied in Galil and Park [9].

Later, it became the idea for the classification of DP on the basis of dependent terms in the recurrence equation. MCM parallelization is implemented for a parallel system by Lee and Hong [10]. A slight increase in the number of operations for computing MCM on a system of parallel processors is compensated by the optimal allocation of processing elements to a part of the MCM. The cache oblivious and cache aware of DP parallelization and related issues are discussed in [11]. The importance of effective utilization of cache memory has been nicely sketched. Multicore throughput becomes a bottleneck due to the huge computing demand originated from various scientific applications.

With the rapid evolution of GPUs, parallelization of general purpose applications on many-core GPUs has been accelerated tremendously and presented in [12], [13], [14], [15], [16]. The various issues enlightened by the research community are efficient utilization of hardware and software resources of GPUs, load balancing and optimal utilization of device cache, and multi-GPU parallelization.

The computational demand of NPDP has migrated from the conventional multi-core to throughput efficient many-core *i.e.* GPU. This section of literature survey covers the state of the art in the parallelization of NPDP problems on GPUs and related issues. Xiao *et al.* [17] proposed the behavior of various optimization techniques such as tiling, memory coalescing, and matrix realignment in the fine grained parallelism of Smith Waterman algorithm. To the best of our knowledge, this can be considered as a milestone in the fine grained parallelization of NPDP on GPU. In the direction of effective allotment of subproblems to GPU threads, three schemes viz. single thread, single block, and multiple blocks are employed for computing a subproblem of matrix chain product, which has been discussed in [18]. On the same line, inherent non-uniformity in the NPDP algorithms is targeted using the thread block analogy *i.e.* single block is employed for each subproblem and the number of threads in a block is the same as the number of comparisons required for computing the subproblem presented by Wu *et al.* [19]. In addition, two stage adaptive thread model for the efficient mapping is illustrated by employing different number of threads for different phases. This analogy leads to the creation of non-manageable thread blocks for the extreme phases. Though, this is very similar to our work in the sense that different computing power is employed to tackle the inherent non-uniformity in the NPDP problems, but it fails to sustain the effective mapping for all the phases of NPDP. Our work should be considered as an extension of this work in the context of the improvement in the effective mapping. To compensate the varying degree of parallelism across the NPDP phases, an attempt is made to enhance the parallelization gain with tiling and efficient shared memory usage, as illustrated in [20]. Moreover, Accelerated Massive Parallelism (AMP) in C++ is utilized for programming the GPUs in MCM parallelization [21] and significant gain is achieved.

Thread level task decomposition and level segmentation of parallel MCM are explored theoretically and experimentally, and parallelization efficacy is analyzed in [22]. Though MCM is a favorite instance in the NPDP category, other instances are also having the same interest because the computational matrix, complexity, and non-uniformity are consistent features for all the instances. In this series, parallelization of other NPDP algorithms such as optimal binary search tree and optimal triangulation has been parallelized in [23], [24], [25], [26].

Parallelization of MCM on manycore GPU is experimented in the recent literature that covers the aspects such as Multi-GPUs, optimal utilization of device memory, *etc.* This provides a motivation for performing and exploring parallelization of this DP category on the GPUs. In multi-GPUs parallelization for the Image Processing, MCM is realized with the help of two Kepler architectures and remarkable results are achieved [27]. Specifically for the robotics field, various algorithms belonging to the Differential Dynamic Programming (DDP) are parallelized and evaluated on the GPU and significant improvements are recorded in comparison with the multithreaded CPU [28]. The Pipelining approach is an experiment in the GPU implementation of parallel DP [29]. Many subproblems of the computational matrix of DP are partially computed in a pipeline fashion.

## 3. MCM and CUDA

### 3.1. MCM

There are numerous examples of NPDP, but MCM proposed by Godbole [30] is chosen as a classic example of it. Formally, MCM is defined as follows: a chain containing the dimensions of $n$ matrices, dimension of matrix $m_i$ is $(p_{i-1} \times p_i)$, $1 \leq i \leq n$, where $p[0...n]$ is the dimension vector of size $(n + 1)$. The NPDP formulation computes the optimal number of scalar multiplications needed for the actual multiplication. Let $m[i, j]$ denote the optimal number of multiplications for multiplying the sequence of matrices from $m_i$ to $m_j$. The $m[1, n]$ holds the minimum number of scalar multiplications for multiplying the sequence of $n$ matrices. The recursive formulation of MCM using DP technique is expressed as Eq. (1) and illustrated in [31], [32]. The skeleton and pseudo code corresponding to Eq. (1) are illustrated in Listing 1. After initializing $m[i, i] = 0$, $\forall i$: $1 \leq i \leq n$, there remains $(n - 1)$ phases to be computed. The time complexity of computation of one sub-problem is $\Theta(i - j)$, *i.e.* the number of comparisons needed for one sub-problem. Conclusively, MCM can be solved using DP approach with time and space complexity $O(n^3)$ and $O(n^2)$, respectively. However, various heuristics are also experimented to reduce the time complexity for this category of computation [33].

As we can identify the non-uniformity from this listing, visualization of this computation is also sketched in Figure 1. In this example, six matrices with random *p*[], *i.e.* dimension vector are considered for the MCM computation. The NPDP phases, subproblems in the phases, and computational flow are shown in Figure 1.

```
void NPDP(){//start
..........
for(r=2; r<=n; r++){/*(n-1)phases,
where n is the input size */
  for(i=0; i<=n-r; i++){/*Number of
subproblems in phase r */
  j = linear_function(i,r);
  Initialize NPDP[i][j] = max_int;
  for(k=i; k <= j-1; k++){/*Number of
comparisons needed*/
  .............
  }//finalization of NPDP[i][j]
  }//finalization of phase r
}//finalization of NPDP table
............
}//close
```

*Listing 1.* NPDP Problem Structure.

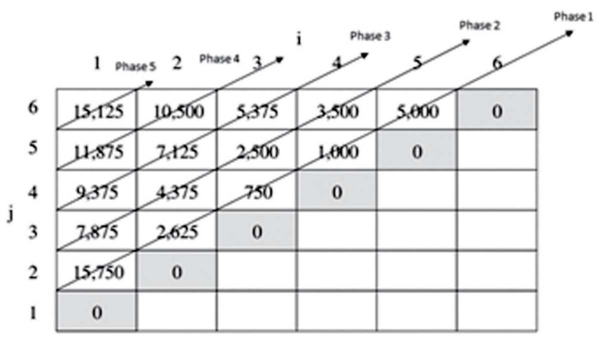

*Figure 1.* MCM Computation.

## 3.2. CUDA

The general purpose graphics processing unit (GPGPU) came into picture in early 2001 by projecting the computationally intensive parallel portions of various applications onto graphics processors. Shaders provide strong support for floating point operations and huge memory bandwidth makes GPUs quite popular for the parallelization of general purpose applications listed by Che *et al.* [34] and various scientific applications such as graphics rendering, bioinformatics applications [12], [13], fluid dynamics [35], [36], databases [37] and linear algebra [38], [39].

Parallelization of the aforementioned applications on GPU can be realized using parallel APIs such as CUDA, OpenCL, OpenGL, *etc*. Among all these, CUDA is quite robust, open source, and it provides an extension to familiar C/C++ libraries for adapting GPU. Conclusively, CUDA is a programming architecture and model for executing a parallel program over the graphics processors.

In GPU computing engine, streaming multiprocessor (SM) is a processing unit at broader level. The SM is further divided into streaming processors (SP). Thread slots, thread block slots, registers and shared memory are the various SM resources, all of them should be utilized effectively and efficiently to fully explore the capacity of the GPU. Kernel is a device function that executes on the GPU. The grid-size and block-size are the two important kernel parameters that define the number of blocks and number of threads in a block respectively, demonstrated in [40], [41], [42]. Selection of appropriate grid-size and block-size plays an important role in the task distribution of a kernel over the SMs. Various architectural specific parameters of a GPU are maximum no. of threads in a block, maximum no. of blocks in an SM, maximum no. of resident threads in an SM, and maximum no. of resident warps in an SM. The Quadro K6000 GPU architecture is il-

$$m[i,j] = \begin{cases} 0 & if\,(i=j) \\ \min_{i \le k < j}\left\{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\right\} & if\,(i<j) \end{cases} \tag{1}$$

lustrated in Figure 2. The nomenclatures related to the architectural parameters of a GPU and NPDP parameters used in this paper are summarized in Table 1.

*Table 1.* Nomenclatures.

| Symbols | Meanings |
|---|---|
| $n$ | Input size, *i.e.* number of matrices in MCM Parallelization |
| $l$ | Phase number of NPDP problem; $(2 \leq l \leq n)$, *i.e.* $(n - 1)$ phases |
| $sm$ | Number of SMs in the targeted GPU processor |
| $sp_l$ | Number of subproblems in phase l, *i.e.* $(n - l + 1)$ |
| $ws$ | Warp-size is 32; *i.e.* implementation specific |
| $mB$ | Maximum resident blocks in an SM |
| $mTB$ | Maximum threads in a block, *i.e.* 1024 for GPUs with computing capability $\geq 2.0$ |
| $mW$ | Maximum resident warps in an SM |
| $mTM$ | Maximum resident threads in SM, *i.e.* $min((mW \times ws), (mB \times mTB))$ |

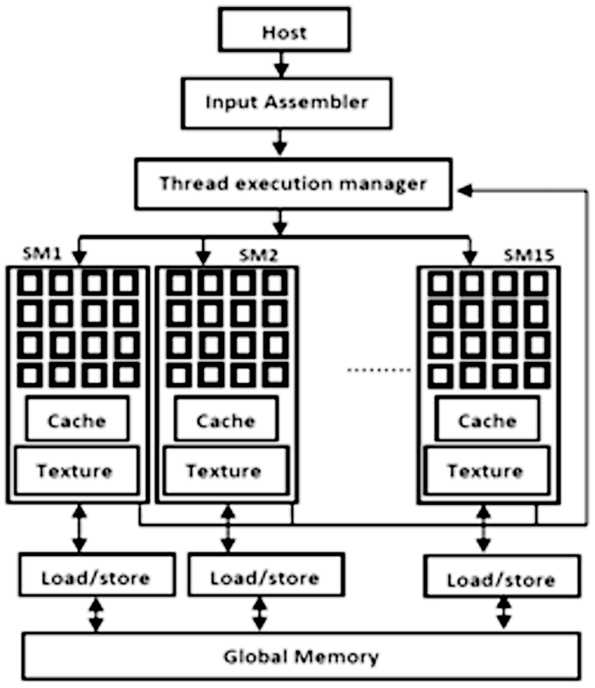

*Figure 2.* NVIDIA CUDA Quadro K6000.

## 4. Parallel Implementation

In CM approaches, same block-size is used throughout all the phases. We change grid-size to meet the appropriate mapping. Listing 2 illustrates the pseudo code for achieving a desired mapping in MCM parallelization.

```
__global__ void MCM(int *p, int n,
int r, int*m){  /*m[i,j] are the
subproblems of a phase */
 Step 1. map each i to thread-id or
block-id depending  upon the desired
mapping
 Step 2. for each such i, m[i,j] is
computed by a thread/block/multi-
blocks (using Equation(1))
 ..............
}/* kernel ends */
int main(){  ..........
  for(r=2; r<=n; r++)/* number of
  phases*/
  MCM<<<grid-size, block-size>>>(p,
  n, r, m);
  /*grid-size decides the CM
  approaches p,n,r,m indicates matrix
  dimensional vector, no. of
  matrices, phase no., and score
  matrix Respectively */
    ..........
}
```

*Listing 2.* Pseudo code of parallel MCM.

We implement four different CM approaches as follows:

(i) *CM Approach*-1: We choose the grid-size only once depending upon the block-size and input-size. The grid-size remains the same for all subsequent phases. This grid-size achieves one-to-one mapping for the first phase. Mapping may not remain the same for the subsequent phases as the total computational elements are fixed.

(ii) *CM Approach*-2: In this approach, grid-size is the same as the number of sub-problems for each and every phase *i.e.* we are trying to implement the sub-problem-block policy. One sub-problem is solved by one thread-block. It may be fruitful to employ a thread-block for a sub-problem belonging to the later phases, but employing a thread-block for a sub-problem belonging to the initial phase leads to unmanageable grid-size.

(iii) *CM Approach*-3: The grid-size is the same for each and every phase irrespective of the block-size and input-size. Appropriate grid-size is chosen for keeping maximal SMs busy for maximum phases. For the illustrations of this approach, we consider two grid-sizes, *i.e.* 32 and 64 with different input-sizes. The notable difference between this approach and Approach-1 is that the grid-size is independent of input-size in this approach, *i.e.* inadequate grid-size is no longer an issue in this approach for moderate input-size.

(iv) *CM Approach*-4: The processing elements, *i.e.* GPU threads are the same as the number of sub-problems in each phase. For the fulfilment of one-to-one mapping, the grid-size is computed depending upon the block-size and number of sub-problems in each phase. Due to this, we only achieve an approximation for one-to-one mapping as the number of sub-problems would mismatch mostly with the product of grid-size and block-size. The grid-size for different CM approaches is summarized in Table 2.

We observe in the previous sections that each CM approach has several advantages as well as several shortcomings. For effective and balanced load distribution of each and every phase

*Table 2.* Grid size strategies for different CM approaches.

| CM Approach | grid-size Computation |
|---|---|
| CM Approach-1 | Computed once to fulfill the one-to-one mapping for the very first-phase and remains the same for all subsequent phases. |
| CM Approach-2 | Computed for each phase, the same as the number of subproblems in a phase. |
| CM Approach-3 | Predefined irrespective of the block-size and input-size and remains the same across all the phases. |
| CM Approach-4 | Computed for each phase to fulfill the one-to-one mapping in each phase. |

of NPDP, we propose a GMM formulation for MCM parallelization. The GMM presents the cumulative advantages of all CM approaches. In GMM, sub-problems belonging to the initial phases are mapped to the processing threads by many-to-one mapping. If the number of sub-problems in a phase exceeds the threading capacity of the GPU, multiple sub-problems are mapped to a single thread. The computations for solving each sub-problem in these initial phases are limited by several comparisons only.

Many-to-one mapping exists only when the input-size is very large. The number of sub-problems decreases uniformly phase-by-phase and, eventually, it comes under the threading capacity of the GPU. We change the mapping from many-to-one to one-to-one. This one-to-one mapping region covers the maximum phases of NPDP. In one-to-one mapping, the number of threads generated in each kernel should be greater than or equal to the number of sub-problems in the respective phase. For sustaining one-to-one mapping, we either reduce the block-size and keep the number of blocks as required or reduce the grid-size by keeping the block-size unchanged. Ultimately, these blocks are further divided into warps and these warps are scheduled for execution by warp schedulers. Enough ready warps are required for each SM for tolerating the long latency operations of the running warp. Reducing grid-size below the number of SMs leads to several SMs becoming idle as blocks are directly mapped to SMs. Hence, in the central computational region of NPDP, the grid-size is kept unchanged. The block-size is reduced after each predefined interval of phases from its maximum supported block-size to an integral multiple of warp-size.

After a certain phase, the number of sub-problems is less than that of the processing threads. After this point, one-to-many mapping is chosen instead of one-to-one, *i.e.* one sub-problem is computed by one thread-block. We can select appropriate block-size. More specifically, we take an increasing integral multiple of warp-size as block-size. We call this mapping as one-to-many$_2$.

When we are not able to keep all the SMs busy even by choosing one-to-many$_2$ mapping, multiple blocks are employed for the computation of one sub-problem. We call this mapping as one-to-many$_1$. The part of sub-problem is

$$M_l = \left\{ \begin{array}{ll} one-to-many_1 & if\,(sp_l < sm) \\ one-to-many_2 & if\,(sm \le sp_l \le (k \times ws \times sm)) \\ one-to-one & if\,((k \times ws \times sm) < sp_l \le (mTM \times sm)) \\ many-to-one & if\,(sp_l > mTM \times sm) \end{array} \right\} \quad (2)$$

```
__global__ void MCM(int *p, int n,
int r, int*m)
{
    /*m[i,j] are the subproblems of a
phase*/
 Step 1. map each i to thread-id or
block-id depending upon the desired
mapping
 Step 2. for each such i, m[i,j] is
computed by a thread/block/multi-
blocks
(using Equation(1))
 ..............
}/* kernel ends */
void main()
 {//Compute n1 and n2
 for(r=2; r<n1; r++)/*Many-to-one
 Mapping */
   MCM<<<full threading
 capacity>>>(p, n, r, m);
 grid-size = sm // Number of SMs;
 for(r=n1; r<n2; r++)/*one-to-one
 Mapping */
 {/*block-size is ranging from
 maximum supported block-size to k
 times warp-size to fulfil the
 requirement of one-to-one mapping */
   MCM<<<grid-size, block-size>>>(p,
 n, r, m);
    ...............
 }
 block-size = warp-size;
 for(r=n2; r<n-sm; r++){/*One-to-ma-
 ny2 Mapping */
   MCM<<<n-r+1, block-size>>>(p, n,
 r, m);
 /* block-size is increasing integral
 multiple of warp-size */
 }
 for(r=n-sm; r<=n; r++)/*One-to-many1
 Mapping */
   MCM<<<grid-size, block-size>>>(p,
 n, r, m);
 }
```

*Listing 3.* Pseudo code of parallel MCM using GMM Approach.

solved by a thread-block and partial solutions from all the blocks are combined to generate an optimal solution for a sub-problem. In GMM, we express the mapping between the sub-problems of phase 1 and the processing elements of the GPU using CUDA by Eq. (2). Listing 3 presents the kernel parameters for different regions of NPDP adopted in the GMM approach.

## 5. Results and Discussion

The performance evaluation of parallel MCM using CM approaches and GMM is experimented on Quadro K6000 GPU. The GPU has 15 SMs, 192 SPs in each SM, with a global memory of 12 GB. The CUDA driver runtime version is 7.5 and 64-bit Ubuntu 16.04 is used with GNU GCC compiler 4.9.2. The dimensions of the matrices that are compatible to the multiplication are randomly generated for experimentation. Figure 3 and Figure 4 present the results of MCM parallelization using CM Approach-1 and Approach-2 respectively, with different block-sizes. In Approach-1, total computational elements are calculated for the very first phase and the remaining phases are also computed with the same computing power.

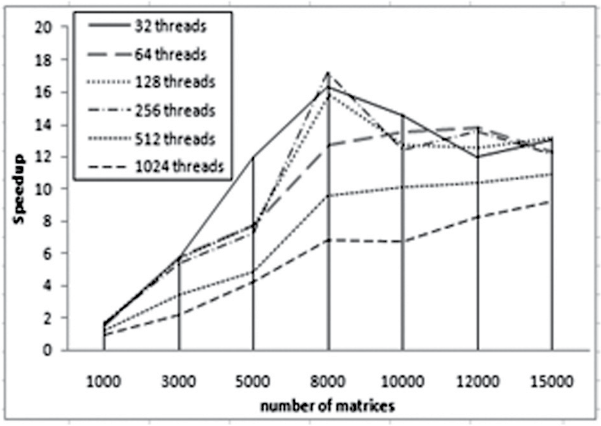

*Figure 3.* MCM parallelization with different block-size using Approach-1.

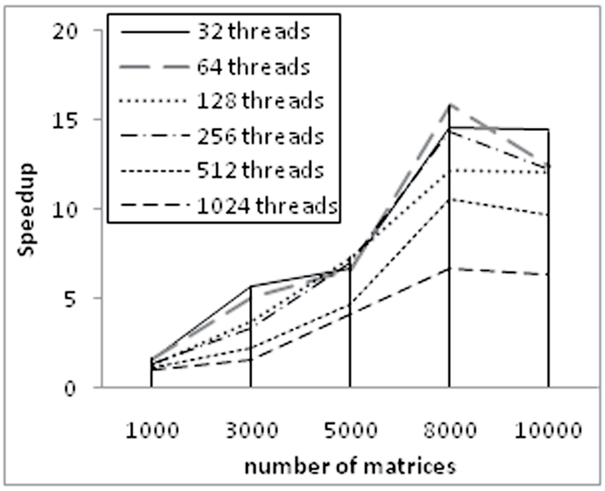

*Figure 4.* MCM parallelization with different block-size using Approach-2.

Mapping gets changed accordingly and eventually poor load balance arises among the GPU threads. In Approach-2, lesser speedup is obtained as compared to Approach-1. We conclude that block-subproblem policy performs worse for initial phases because of the two reasons: ample number of blocks and whole thread-block are actually not required for performing several comparisons of a sub-problem. Approach-2 performs the worst for later phases when the number of sub-problems is less than the number of SMs. A sub-problem with huge computation should not be catered by a single thread-block.

Figure 5 and Figure 6 highlight the results of CM Approach-3 with grid-size 32 and 64 respectively. Comparatively more speedup is obtained in Approach-3 as we fixed the grid-size throughout all the phases of NPDP. These grid-sizes produce enough blocks in each phase to keep all the SMs busy, but non-optimal mapping for several phases still persists. On the basis of peak speedup attained in these results, we conclude that managing larger grid is also a bottleneck for CUDA runtime.

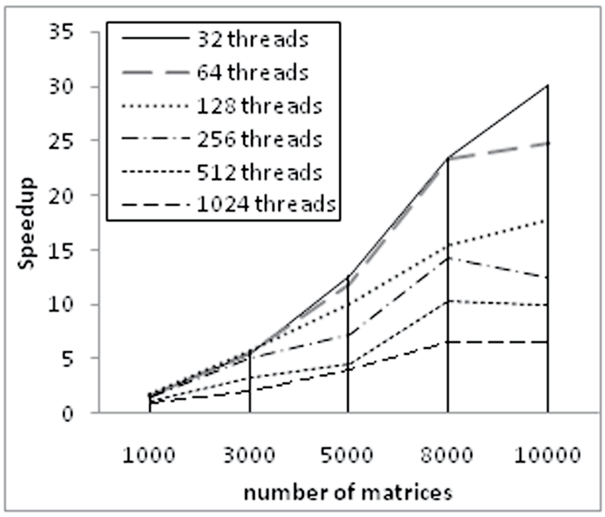

*Figure 5.* MCM parallelization using Approach-3 with block-size 32.

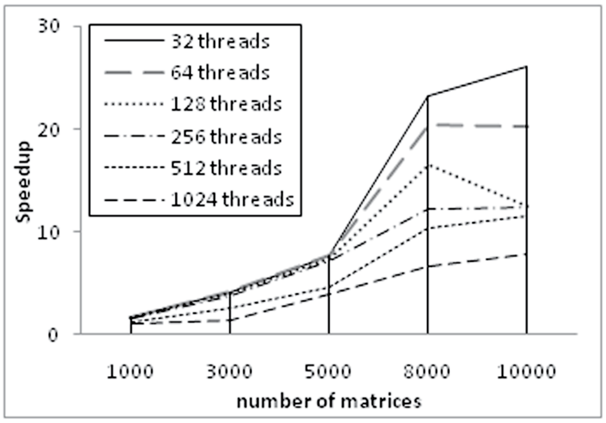

*Figure 6.* MCM parallelization using Approach-3 with block-size 64.

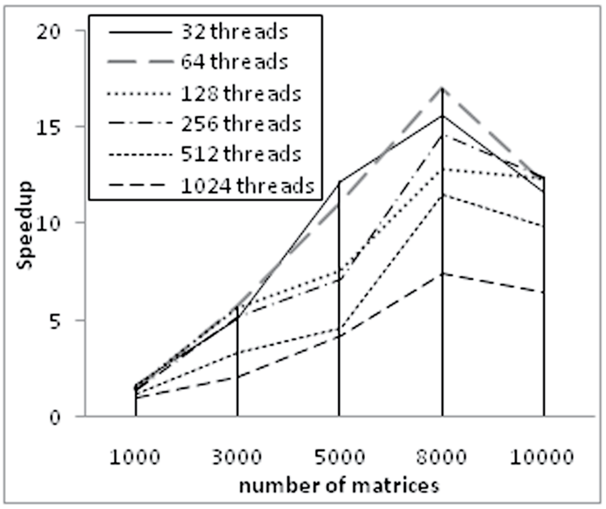

*Figure 7.* MCM parallelization with different block-size using Approach-4.

Figure 7 shows the outcome of MCM parallelization using CM Approach-4. This approach is suitable for the initial phase of computation whereas it leads to the worst mapping for later phases of NPDP. It leads to the idealization of more SMs for later phases. Achieving one-to-one mapping is not easy in case of fixed block-size strategy. This approach performs the worst in comparison with all three previous CM approaches.

We observe that the smaller block-size produces better results in NPDP parallelization. Large block-size degrades the performance because of under-utilization of SMs for later phases. For small input-size, we don't get significant difference in the achieved speedup. As we increase the input-size, notable difference in the speedup is seen. Approach-3 performs better as compared to other CM approaches because of steady and appropriate grid-size.

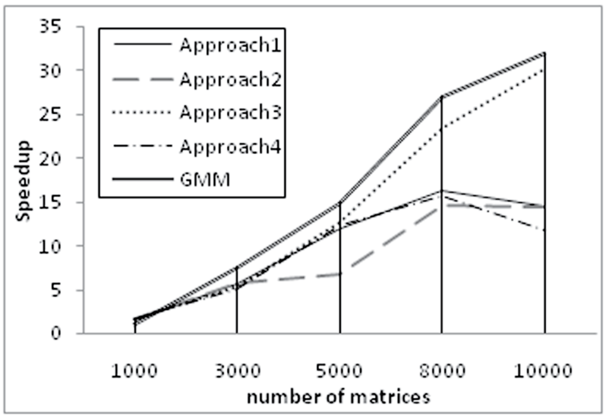

*Figure 8.* Comparative illustration among CM approaches and GMM.

The comparative study among the gains of CM approaches and GMM is shown in Figure 8. In GMM, desired grid-size and block-size are computed for each and every region of the NPDP to meet the necessary mapping expressed in Eq. (2). The significant gain in the speedup is obtained in GMM in comparison with the best CM approach, *i.e.* Approach-3. Partitioning the computational region of NPDP as per the Eq. (2) and selecting the appropriate computing power for each phase lead to better mapping for the entire NPDP computation and maximum utilization of SMs in all the phases. In comparisons of these results with earlier state-of-art by Wu *et al.* [19], the maximum achieved speedup was 13.40×, whereas speedup using GMM approach is more than 30×.

## 6. Conclusions

We propose, formulate, and analyze a comparative investigation among CM approaches and the proposed GMM in the GPU parallelization of NPDP. MCM is chosen for the implementation of all the CM approaches and GMM. GMM outperforms the state of the art by employing pre-calculated computing resources for each and every phase that in turn achieves an optimal mapping for the respective phase.

Among the outcomes of the CM approaches, approach 3 performs much better for sufficiently large size input, as illustrated in the results section. The enhanced speedup is due to the adequate number of threads and the number of blocks in the parallelization of each phase. Sufficient number of blocks would lead to keeping maximum SMs busy. The probable bottleneck of this approach is the non-suitability for the initial phases of NPDP. We also conclude that generation of maximal computing elements does not necessarily draw the efficient mapping in GPU parallelization. Sufficient number of blocks with sufficient threads in each block would cater the subproblems well. Sufficient number of blocks means all the SMs should be busy as blocks are directly mapped to SMs. Sufficient threads in a block means that enough warps should be there for each SM.

Efficient mapping plays a vital role in the maximal utilization of the GPU. Performance degradation due to the non-uniformity in the computations across the phases of NPDP is counter balanced by the suitable mapping. There may be little deviations in the results of NPDP parallelization if we use other advanced GPU or multiple GPUs. As the computational pattern and parallelization strategy is the same for the other problems of NPDP, GMM approach is applicable for them, too. Parallelization of other NPDP instances and related issues shall be fueled with this work. We also abridged the literature gap by providing this work on NPDP parallelization. Effective tiled mapping for NPDP parallelization is our ongoing work.

## References

[1] G. Tan *et al.*, "Improving Performance of Dynamic Programming via Parallelism and Locality on Multicore Architectures", *IEEE Transaction on Parallel Distributed Systems*, vol. 20, no. 2, pp. 261–274, 2009.
http://dx.doi.org/10.1109/TPDS.2008.78

[2] R. Chandra *et al.*, "Parallel Programming in OpenMP", Morgan Kaufmann Publishers, USA, 2001.

[3] R. Blikberg and T. Sorevik, "Load Balancing and OpenMP Implementation of Nested Parallelism", Parallel Computing, vol. 31, no. 10–12, pp. 984–998, 2005. http://dx.doi.org/10.1016/j.parco.2005.03.018

[4] B. Chapman *et al.*, "Using OpenMP: Portable Shared Memory Parallel Programming", The MIT Press, USA, 2007.

[5] OpenMP specifications. (2017). Retrieved (14 April, 2018) from http://www.openmp.org/specs.

[6] Z. Galil and K. Park, "Dynamic Programming with Convexity, Concavity and Sparsity", *Theoretical Computer Science*, vol. 92, no. 1, pp. 49–76, 1999. http://dx.doi.org/10.1016/0304-3975(92)90135-3

[7] P. G. Bradford, "Efficient Parallel Dynamic Programming", Indiana University, Technical Report 352, 1992.

[8] S. Huang *et al.*, "Parallel Dynamic Programming", *IEEE Transaction on Parallel and Distributed Systems*, vol. 5, no. 3, pp. 326–328, http://dx.doi.org/1994. 10.1109/71.277784

[9] Z. Galil and K. Park, "Parallel Algorithms for Dynamic Programming Recurrences with more than O(1) Dependency", *Journal of Parallel and Distributed Computing*, vol. 21, no. 2, pp. 213–222, 1994. http://dx.doi.org/10.1006/jpdc.1994.1053

[10] H. Lee and S. J. Hong, "Processor Allocation and Task Scheduling of Matrix Chain Products on Parallel Systems", *IEEE Transactions on Parallel Distributed Systems*, vol. 14, no. 4, pp. 394–407, 2003. http://dx.doi.org/10.1109/TPDS.2003.1195411

[11] R. Chowdhury and V. Ramachandran, "Cache-efficient Dynamic Programming Algorithms for Multicores", Department of Computer Sciences, UT-Austin, Technical Report, TR-08-16, 2008.

[12] G. Tan *et al.*, "Locality and Parallelism Optimization for Dynamic Programming Algorithm in Bioinformatics", in *Proc. of the ACM/IEEE Conference on Supercomputing*, 2006, pp. 11–17. http://dx.doi.org/10.1109/SC.2006.41

[13] S. Tang *et al.*, "Easy PDP: An Efficient Parallel Dynamic Programming Runtime System for Computational Biology", *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 5, pp. 862–872, 2011. http://dx.doi.org/10.1109/TPDS.2011.218

[14] S. Ryoo *et al.*, "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA", in *Proc. of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, 2008, pp. 73–82. http://dx.doi.org/10.1145/1345206.1345220

[15] S. Solomon and P. Thulasiraman, "Performance Study of Mapping Irregular Computations on GPUs", in *Proc. of the IEEE Int. Symposium on Parallel and Distributed Processing*, 2010, pp. 1–8. http://dx.doi.org/10.1109/IPDPSW.2010.5470770

[16] D. Strnad and N. Guid. "Parallel Alpha-beta Algorithm on the GPU", *CIT. Journal of Computing and Information Technology*, vol. 19, no. 4, pp. 269–274, 2011. http://dx.doi.org/10.2498/cit.1002029

[17] S. Xiao *et al.*, "On the Robust Mapping of Dynamic Programming onto a Graphics Processing Unit", in *Proc. of the 15th Int. Conference on Parallel and Distributed Systems (ICPADS-2009)*, 2009, pp. 26–33. http://dx.doi.org/10.1109/ICPADS.2009.110

[18] K. Nishida *et al.*, "Accelerating the Dynamic Programming for Matrix Chain Product on the GPU", in *Proc. of the 2nd Int. Conference on Networking and Computing (ICNC-2011)*, 2011, pp. 320–326. http://dx.doi.org/10.1109/ICNC.2011.62

[19] C. C. Wu *et al.*, "Optimizing Dynamic Programming on Graphics Processing Units via Adaptive Thread-Level Parallelism", in *Proc. of the 17th Int. Conference on Parallel and Distributed Systems (ICPADS-2011)*, 2011, pp. 96–103. http://dx.doi.org/10.1109/ICPADS.2011.92

[20] C. C. Wu *et al.*, "Optimizing Dynamic Programming on Graphics Processing Units Via Data Reuse and Data Prefetch with Inter-Block Barrier Synchronization", in *Proc. of the IEEE Int. Conference on Parallel and Distributed Systems*, 2012, pp. 45–52. http://dx.doi.org/10.1109/ICPADS.2012.17

[21] K. Shyamala *et al.*, "Design and Implementation of GPU-based Matrix Chain Multiplication using C++ AMP", in *Proc. of the 2nd Int. Conf. on Electrical, Computer and Communication Technologies (ICECCT)*, 2017, pp. 1–6. http://dx.doi.org/10.1109/ICECCT.2017.8117870

[22] B. B. Mabrouk *et al.*, "Theoretical and Experimental Study of a Parallel Algorithm Solving the Matrix Chain Product Problem" in *Proc. of the Int. Conference on Parallel and Distributed Processing Techniques and Applications*, 2017, pp. 341–347. https://csce.ucmss.com/cr/books/2017/LFS/CSREA2017/PDP6155.pdf.

[23] B. Han and L. Yongquan, "Research on Optimization and Parallelization of Optimal Binary Search Tree Using Dynamic Programming", in *Proc. of the 2nd Int. Conference on Electronic*

*and Mechanical Engineering and Information Technology*, 2012.
https://doi.org/10.2991/emeit.2012.45

[24] Y. Ito and K. Nakano, "A GPU Implementation of Dynamic Programming for the Optimal Polygon Triangulation", *IEICE Transactions on Information and Systems*, vol. E96.D, no. 12, pp. 2596–2603, 2013.
https://doi.org/10.1587/transinf.E96.D.2596

[25] J. F. Myoupo and V. K. Tchendji, "Parallel Dynamic Programming for Solving the Optimal Search Binary Tree Problem on CGM", *Int. J. of High Performance Computing and Networking*, vol. 7, no. 4, pp. 269–280, 2014.
http://dx.doi.org/10.1504/IJHPCN.2014.062729

[26] P. Ganapathi, "Automatic Discovery of Efficient Divide-&-Conquer Algorithms for Dynamic Programming Problems", PhD Thesis, Stony Brook University, 2016.

[27] J. Ke *et al.*, "Optimized GPU implementation for Dynamic Programming in Image Data Processing", in *Proc. of the 35th Int. Performance Computing and Communications Conference (IPCCC)*, 2016, pp. 1–7.
http://dx.doi.org/10.1109/PCCC.2016.7820646

[28] B. Plancher and S. Kuindersma, "A Performance Analysis of Parallel Differential Dynamic Programming on a GPU", in *Proc. of the Int. Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 2018.
https://agile.seas.harvard.edu/files/agile/files/gpu-ddp.pdf.

[29] M. Miyazaki, and S. Matsumae, "A Pipeline Implementation for Dynamic Programming on GPU", in *Proc. of the 6th Int. Symp. on Computing and Networking Workshops (CANDARW)*, 2018, pp. 305–309.
http://dx.doi.org/10.1109/CANDARW.2018.00063

[30] S. S. Godbole, "On Efficient Computation of Matrix Chain Products", *IEEE Transactions on Computers*, vol. C-22, no. 9, pp. 864–866, 1973.
http://dx.doi.org/10.1109/TC.1973.5009182

[31] T. H. Cormen *et al.*, "Introduction to Algorithm" 2nd ed., PHI Learning Private Limited, New York, 2008.

[32] J. Leung, "Handbook of scheduling: algorithms, models, and performance analysis", Chapman & Hall/CRC, New York, 2004.

[33] B. Suvarna and T. Maruthi Padmaja, "Enhanced Matrix Chain Multiplication", *J. of Cyber Security and Mobility*, vol. 7, no. 4, pp. 409–420, 2018.
http://dx.doi.org/10.13052/jcsm2245-1439.743

[34] S. Che *et al.*, "A Performance Study of General Purpose Applications on Graphics Processors using CUDA", *J. of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370–1380, 2008.
https://doi.org/10.1016/j.jpdc.2008.05.014

[35] A. C. Crespo *et al.*, "GPUs, a New Tool of Acceleration in CFD: Efficiency and Reliability on Smoothed Particle Hydrodynamics Methods", PLoS ONE, vol. 6(6): e20685, 2011.
https://doi.org/10.1371/journal.pone.0020685

[36] K. E. Niemeyer and C. J. Sung, "Recent Progress and Challenges in Exploiting Graphics Processors in Computational Fluid Dynamics", *J. of Supercomputer*, vol. 67, no. 2, pp. 528–564, 2014.
https://doi.org/10.1007/s11227-013-1015-7

[37] P. Bakkum and K. Skadron, "Accelerating SQL Database Operations on a GPU with CUDA", in *Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 94–103.
http://dx.doi.org/10.1145/1735688.1735706

[38] Ph. Tillet *et al.*, "Towards Performance-Portable, Scalable, and Convenient Linear Algebra". in *Proc. of the 5th USENIX Workshop on Hot Topics in Parallelism*, 2013.
https://pdfs.semanticscholar.org/f6f2/216c4172748e8ca7c423d447e5804174e1df.pdf.

[39] J. D. Carvalho Maia, "GPU Linear Algebra Libraries and GPGPU Programming for Accelerating MOPAC Semiempirical Quantum Chemistry Calculations", *J. of Chemical Theory and Computation*, vol. 8, no. 9, pp. 3072–3081, 2012.
http://dx.doi.org/10.1021/ct3004645

[40] NVIDIA Corporation, "CUDA Programming Guide 4.2", 2012.
https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[41] D. B. Kirk and W. Wen-mei Hwu, "Programming Massively Parallel Processors", Elsevier Inc., USA, 2010.

[42] E. Kandrot and J. Sanders, "CUDA by Example: An Introduction to General-purpose GPU Programming", Addison-Wesley, New York, 2011.

*Received*: January 2019
*Revised:* March 2019
*Accepted:* April 2019

*Contact addresses*:
Tausif Diwan
Indian Institute of Information Technology
Nagpur
India
e-mail: tausif.diwan@cse.iiitn.ac.in

Tausif Diwan received the M.Tech. and PhD degrees from the Department of Computer Science and Engineering, VNIT Nagpur, India in 2011 and 2017, respectively. From June 2012 to July 2019, he was associated with the Department of Computer Science and Engineering, RCOEM Nagpur, India as an Assistant Professor. He joined IIIT Nagpur as an Assistant Professor in the Department of Computer Science and Engineering in July 2019. His research area includes parallel computing, analysis of algorithm, machine learning, and deep learning.

Jitendra Tembhurne
Indian Institute of Information Technology
Nagpur
India
e-mail: jitendra.tembhurne@cse.iiitn.ac.in

Jitendra Tembhurne received his B.E. degree in Computer Technology from KITS, Ramtek, Nagpur, India in 2003. He received the M.E. degree in Computer Science and Engineering from MGMCoE, Nanded, India in 2011 and PhD degree in Computer Science and Engineering from VNIT, Nagpur, India in 2017. In 2005, he joined the Department of Computer Technology, KITS, Ramtek, Nagpur, as a Lecturer. In 2010, he was promoted to the position of Assistant Professor. He joined the Department of Computer Science and Engineering, Indian Institute of Information Technology, Nagpur, India, as an Assistant Professor in 2018. His research interests include deep learning, parallel computing, algorithms on multi-core and many-core architectures, data mining, and security.