2011 Symposium on Application Accelerators in High-Performance Computing

# Non-Serial Polyadic Dynamic Programming on a Data-Parallel Many-core Architecture

Maryam Moazeni, Majid Sarrafzadeh
Computer Science Department
University of California, Los Angeles
Los Angeles, CA
{mmoazeni, majid}@cs.ucla.edu

Alex A. T. Bui
Department of Radiological Sciences
University of California, Los Angeles
Los Angeles, CA
abui@mii.ucla.edu

*Abstract*— **Dynamic Programming (DP) is a method for efficiently solving a broad range of search and optimization problems. As a result, techniques for managing large-scale DP problems are often critical to the performance of many applications. DP algorithms are often hard to parallelize. In this paper, we address the challenge of exploiting fine grain parallelism on a family of DP algorithms known as non-serial polyadic. We use an abstract formulation of non-serial polyadic DP, derived from RNA secondary structure prediction and matrix parenthesization approaches that are well-known and important problems from this family. We present a load balancing algorithm that achieves the best overall performance with this type of workload on many-core architectures. A divide-and-conquer approach previously used on multi-core architectures is compared against an iterative version. To evaluate these approaches, the algorithm was implemented on three NVIDIA GPUs using CUDA. We achieved up to 10 GFLOP/s performance and up to 228x speedup over the single-threaded CPU implementation. Moreover, the iterative approach results in up to 3.92x speedup over the divide-and-conquer approach.**

**Keywords-component; GPGPU; Dynamic Programming; Many-core**

## I. INTRODUCTION

Dynamic programming is a method for efficiently solving a broad range of search and optimization problems that exhibit the characteristics of overlapping sub-problems. This technique is used in many application domains such as bioinformatics, VLSI design, scheduling, and inventory management. As a result, techniques for efficiently solving large-scale DP problems are often critical to the performance of many applications. Dynamic programming is one of the important classes of application recognized as one of the dwarfs in [3].

In general, dynamic programming has limited parallelism. However, due to its importance, parallel dynamic programming has become a classic problem and it is relatively well-studied on multi-core architectures. Optimizing dynamic programming for many-core architectures is different than multi-core architectures. A many-core implementation must distribute work among hundreds or thousands of threads. Many-core architectures are designed to hide the latency of memory accesses by means of multi-threading instead of using caches.

In this paper, we address the challenge of exploiting fine-grain parallelism of nonserial polyadic dynamic programming [4], which is a class of dynamic programming in which the sub-problem located on all levels depends on more than one previous level; and the functional equation contains more than one recursive term (more details on the differences among the four classes of dynamic programming can be found in [4]). We use an abstract formulation of non-serial polyadic DP, which was derived from RNA secondary structure prediction and matrix parenthesization. We present a load balancing algorithm that achieves the best overall performance with this type of workload on many-core architectures. Our optimization reasoning is based on performance data obtained via profiling and quantitative analysis. We compare a divide-and-conquer approach previously used on multi-core architectures with an iterative bottom-up approach. The divide-and-conquer approach was popular for multi-core implementations; however, as shown, the divide-and-conquer approach results in very poor load balancing with this workload. A dynamic programming workload is not pure SIMD (single instruction, multiple data); therefore, the load balancing algorithm is an important factor in achieving optimal performance. This is in contrast to multi-core implementations that only need to create tens of executions threads. Thus, workload imbalance was not a critical concern as much as data locality and cache performance.

The rest of the paper is organized as follows. After discussing related work in Section II, we present the abstract formulation of non-serial polyadic DP in Section III. In Section IV, we present the implementation on GPU. Section V, describes our load balancing algorithm. Section VI includes performance analysis and finally Section VII concludes the paper.

## II. RELATED WORK

There is no previous work that has carefully studied nonserial polyadic dynamic programming workload on GPUs. Throughout this paper, we compare our approach with the multi-core implementations of RNA secondary structure prediction presented in [1, 2], which entailed a 30x speedup. They presented a divide-and-conquer approach and a parallel pipeline for decomposing computations and improving cache performance in multi-core architectures. Our goal is to present the differences between techniques that can be used for accelerating DP on multi-core and many-core architectures.

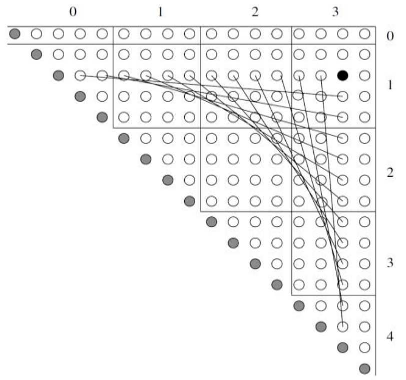

**Figure 1. The blocked transformed DP matrix [1].**

## III. PROBLEM FORMULATION

We use an abstract DP formulation that is based on a DP formulation for RNA secondary structure prediction and optimal matrix parenthesization from the nonserial polyadic family, previously used in [1]. In most applications, the computation in the formulation mainly involves floating point operations. The abstract formulation is as follows, where a(i) is an initial value:

$$m[i,j]=\begin{cases}\min_{i\le k<j}\{m[i,j],m[i,k]+m[k+1,j]\} & 0\le i<j<n \\ a(i) & i=j\end{cases} \quad (1)$$

The data dependence in this DP exists between non-consecutive stages, which make the data access pattern non-uniform. The non-uniform data access pattern makes this problem harder to optimize for parallelization. Therefore, we also use the data transformation that was used in [1] to eliminate cross block references, improving data locality (Figure 1). Assume (i, j) is the original coordinate in the original domain D = {(i, j) | 0 ≤ i ≤ j < n }, where n = |D| is the original problem size, (i', j') is the new coordinate in the transformed domain D' = {(i', j') | 0 ≤ i' ≤ j' < n' }, where n' = n + 1 = |D'| is the new problem size. The iteration domain transformation is defined as follows:

$$(i^{'},j^{'})=f(i,j)\colon i^{'}=i, j^{'}=j+1$$

Therefore, Equation 1 is rewritten as the new Equation 2 in the transformed domain, where a(i) is the known initial value. The values on the new diagonal can be any value. In the new domain, the values on the new diagonal do not contribute to the computation.

$$m[i',j']=\begin{cases}\min_{i'+1\le k'<j'}\{m[i',j'],m[i',k']+m[k',j']\} & 0\le i'<j'<n' \\ a(i) & i'=j'\end{cases} \quad (2)$$

### A. Parallelism

To exploit fine-grain parallelism we can use the blocking technique to decompose the computations. By using the transformed DP formulation, the cross block reference is eliminated in the blocked matrix.

The blocked algorithm can be observed as comprising many matrix block operations. Let matrices $A=(a_{ij})_{sxs}$, $B=(b_{ij})_{sxs}$, $C=(c_{ij})_{sxs}$,, the tensor operations $\otimes$ and $\oplus$ for the blocked matrix is defined as follows:

*Definition* 1. $\forall a_{ij}\in A, b_{ij}\in B, c_{ij}\in C,\ 1\le i,\ j\le n,$

*if* $c_{ij}=\min_{k=1}^{n}\{c_{ij},a_{ij}+b_{ij}\}$, *then* $C=A\otimes B$

*Definition* 2. $\forall a_{ij}\in A, b_{ij}\in B, c_{ij}\in C,\ 1\le i,\ j\le n,$

*if* $c_{ij}=\min\{a_{ij},b_{ij}\}$, *then* $C=A\oplus B$

The formulation to compute any matrix sub-blocks[1] A (i, j) is as follows:

$$\begin{aligned}A(i,j)&=\oplus_{k=i}^{j}(A(i,k)\otimes A(k,j))\\&=(\oplus_{k=i+1}^{j-1}(A(i,k)\otimes A(k,j)))\\&\quad\oplus(A(i,i)\otimes A(i,j))\oplus(A(i,j)\otimes A(j,j))\end{aligned} \quad (3)$$

In this equation, the calculation is divided into two parts. The first part uses the rectangular blocks in the same row and column. We refer to this part of the computation as the rectangular computation:

$$(\oplus_{k=i+1}^{j-1}(A(i,k)\otimes A(k,j)) \quad (4)$$

The second part of the computation depends on triangular blocks and itself. We refer to this computation as the triangular computation:

$$(A(i,i)\otimes A(i,j))\oplus(A(i,j)\otimes A(j,j)) \quad (5)$$

These computations contribute to the partial result of the block A(i, j). For each k in Equation 4 above, all the blocks A(i, j) in the same diagonal can be computed in parallel. In Equation 5, the two $\otimes$ operations depend on the result of A(i, j) from Equation 4; this computation can be performed in parallel for all the blocks A(i, j) in the same diagonal.

For rectangular computation, each element in one block is mapped to one thread. According to Definition 1, there is no dependency between elements in a block for $\otimes$ operation, so all threads can be computed in parallel, similar to a dense matrix multiplication.

## IV. NONSERIAL POLYADIC DP ON GPU

There are two diverse forms of computation in this dynamic programming formulation, the rectangular computation and the triangular computation, which makes the load balancing of this workload interesting on many-core architectures.

The rectangular computation workload and data access pattern is very similar to a dense matrix multiplication and the same optimizations can be applied. The triangular computation, on the other hand, has very limited parallelism because of its data dependency between two consecutive entries: parallelism can only be exploited along the diagonal. However, triangular computations can be broken down into rectangular computations and triangular computations of a smaller size to expose more parallelism. Therefore, a good load balancing algorithm is to reduce the proportion of triangular computations.

The rectangular computation of each block is dependent on blocks in the same row and column. Therefore, the execution of blocks proceeds along the diagonal. Because the computations for each block in the same diagonal do not have any interdependencies, they can be computed in parallel. In the multicore implementation [1], on the order of 10s of threads are used (i.e., 16, 32, and 64). Therefore, in contrast to a many-core implementation, it is not possible to

[1] In the rest of this paper, matrix sub-blocks are referred as blocks.

take advantage of the parallelism that exists for the computation of blocks in the same diagonal. However, a single GPU kernel can distribute work among thousands or tens of thousands of threads, which can exploit parallelism at this level.

For each k in Equation 4: all the blocks A(i, j) in the same diagonal are computed in a single compute_rectangular kernel execution in parallel. For Equation 5, we merge the two tensor operations in a single kernel and all the blocks A (i, j) in the same diagonal are computed in a single compute_triangular kernel execution in parallel. As we explored the different design options through profiling the workload, we concluded that the load balancing algorithm plays a more important role in achieving the best performance versus micro-optimization of the kernels.

## V. LOAD BALANCING ALGORITHM

The DP matrix can be filled in two fashions. First is the divide-and-conquer approach, which is also used in the multi-core implementation of the RNA secondary structure prediction in [1]. We can use this divide-and-conquer approach to obtain the decomposition of computations described in Section III to exploit higher fine parallelism. Another approach to achieve the same decomposition of the computations is a bottom-up iterative strategy. In the bottom-up iterative approach, the DP matrix is partitioned into fixed-size blocks and all the blocks A(i, j) in the same diagonal are computed in each iteration based on equation 3.

The best load balancing algorithm is the one that reduces the proportion of triangular computation as there is very limited parallelism in that kernel. With regards to the rectangular computations, as we increase the problem size the kernel execution time does not increase linearly as matrix multiplication is $O(n^3)$,. This is also the same for the dense matrix multiplication kernel, which has the same characteristics as the compute_rectangular kernel. As a consequence, a good load balancing algorithm should result in more calls to the compute_rectangular kernel with smaller size.

By increasing the problem size, the iterative bottom-up approach results in significantly more calls to the compute_rectangular kernel and less calls to the compute_triangular kernel. The divide-and-conquer approach performs the rectangular computations with problem size n and divides the triangular computations into rectangular and triangular computations with problem size n/2. This pattern of execution results in less calls to the compute_rectangular kernel with variable sizes. On the other hand, the iterative bottom-up approach results in more calls to the compute_rectuangular with fixed size (and smaller than those in the divide-and-conquer approach in overall). More detail on the performance evaluation of the two approaches is explained in Section VI.

## VI. PERFORMANCE ANALYSIS

We evaluate the performance of the parallel non-serial polyadic dynamic programming on three different NVIDIA GPUs: Tesla C1060, Quadro FX 5600, and GeForce 8800 GT with CUDA 3.2 paired with an Intel Core i7 965 CPU2. The three GPUs have a different number of parallel processor cores and clock rates. We use these differences to evaluate how the workload scales by increasing the number of available processor cores. We report performance in GFlops, determined by dividing the required arithmetic operations by the average execution time. The execution time is obtained by taking the minimum time over multiple runs. The time for transferring the data to GPU is considered in the execution time. We compare our GPU implementations with a single-threaded CPU.

[2] For the sake of reproducibility, source code and data used in this evaluation will be published online.

### *A. Load Balancing*

In this section, using an exhaustive set of experiments we show that the iterative bottom-up approach results in better load balancing compared to the divide-and-conquer approach previously used in multi-core architectures. As mentioned earlier, a good load balancing algorithm is to reduce the proportion of triangular computations as it exposes very limited parallelism and to increase the proportion of rectangular computations.

We conducted experiments to compare the divide-and-conquer load balancing approach with the iterative bottom-up. Table 1 shows that the iterative bottom-up approach achieves significantly better results compared to divide-and-conquer. For the rectangular computations, the divide-and-conquer approach schedules less blocks with larger size, as compared to the iterative bottom-up approach. This strategy does not achieve the best results because as we increase the problem size the compute_rectangular kernel execution time does not increase linearly. The reason is that we launch O(n2) execution threads on each kernel, but the number of processor cores is constant. Therefore, execution is serialized. Hence, scheduling more blocks with smaller size for rectangular computation achieves best results.

Block size can also make a big difference in overall running time. Although the key to performance on this platform is using massive multithreading to utilize the large number of cores and hide global memory latency, partitioning the matrix to larger sub-matrices does not result in reduced overall running time. As the running time does not increase linearly as we increase the size of the compute_rectangular kernel, significantly better result is achieved with smaller block sizes compared to larger block sizes in this workload.

**Table 1. Speedup of iterative bottom-up approach over divide-and-conquer approach**

| Size | 1024 | 2048 | 4096 | 8192 |
|---|---|---|---|---|
| Speedup | 1.01X | 1.49X | 2.34X | 3.92X |

Note that the block size is not equal to the size of the load on each kernel, but equal to the number of blocks in the current diagonal times the block size. Figure 2 shows the result of our experiments on the Tesla using iterative bottom-up load balancing. On the Tesla, for problem size n < 512 the block size of 32 gives best results; and for problem sizes n > 512, the block size of 64 gives the best results. On the GeForce, a block size of 32 always gives the best result. The Quadro FX 5600 exhibits the same behavior as the Tesla. Figure 3 presents the speedups achieved on the three GPUs as compared to a single-threaded implementation on an Intel Core i7 CPU. We achieved up to

228x speedup on the Tesla. Figure 4 demonstrates the performance in GFLOP/s.

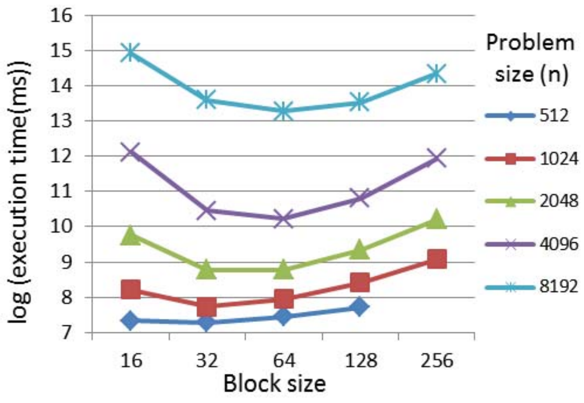

Figure 2. Execution time on TeslaC1060

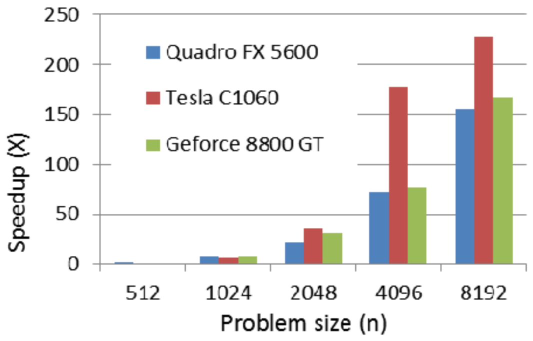

Figure 3. Speedup over single-threaded CPU implementation

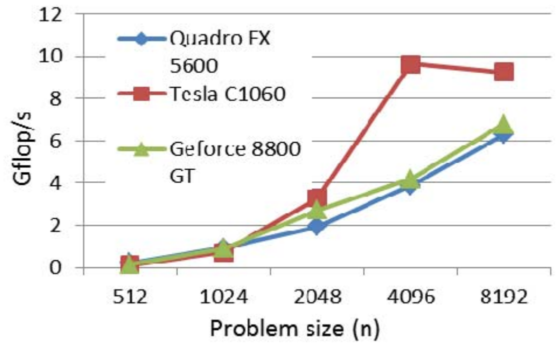

Figure 4. Performance of the three GPUs

For the Quadro and the GeForce GPUs, performance continues to increase as we grow the problem size. However, on the Tesla, the performance remains almost constant after reaching a size of n = 4096. This workload is very interesting for data-parallel many-core architectures because even after the decomposition of operations, the workload is still not purely SIMD.

## B. Scaling

We examined the scaling properties of our algorithm with respect to the number of processor cores. Figure 5 shows the execution time of our algorithm running on 112, 128, and 240 processors. Again, the iterative bottom-up approach for load balancing was used. For $n < 512$, a block size of 32 was used; and for $n > 512$, a block size of 64 was employed for optimal load balancing. As shown in this experiment, for $n < 1024$, the execution time on the Tesla GPU is greater than the execution time on the Quadro GPU. However as the problem size increases, the execution time on the Tesla GPU is between 2.4 to 1.4 times less than the Quadro and GeForce GPUs. The Tesla has 1.8 and 2.1 times more processor cores than the Quadro and GeForce GPUs, respectively. This suggests that the algorithm has potential scalability on many-core architectures, as increasing the number of processing cores results in proportionally higher performance.

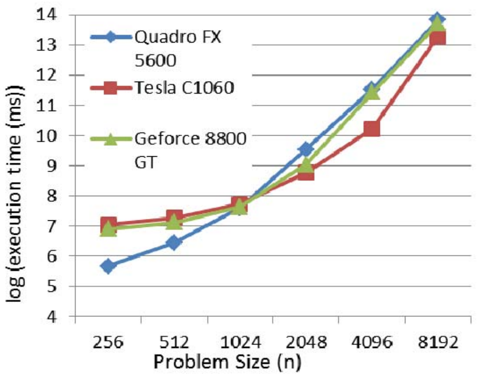

Figure 5. Comparison of execution time on three GPUs

## VII. CONCLUSION

We studied different approaches for porting a family of Dynamic Programming (DP) algorithms to GPUs. To do so, we used an abstract formulation of non-serial polyadic DP groups of algorithms. We presented how load balancing can be an important factor in achieving the right level of parallel granularity on many-core architectures, in contrast to multi-core architectures where locality and cache performance are more critical concerns. We achieved up to 228x speedup and 10 GFLOP/s on the Tesla C1060.

## REFERENCES

[1] Guangming Tan, Ninghui Sun, and Guang R. Gao. 2007. A parallel dynamic programming algorithm on a multi-core architecture. In Proceedings of SPAA '07.

[2] Guangming Tan, Shengzhong Feng, and Ninghui Sun. 2006. Locality and parallelism optimization for dynamic programming algorithm in bioinformatics. In Proceedings of SC '06.

[3] Krste Asanovic, et. al., 2009. A view of the parallel computing landscape. Communications. ACM 52, 10 (October 2009), 56-67.

[4] A. Grama, A. Gupta, G.Karypis, V. Kumar, 2003. Introduction to parallel computing, Addison Wesley.