Toward Efficient Architecture-Independent Algorithms for Dynamic Programs

Mohammad Mahdi Javanmard[1], Pramod Ganapathi[2], Rathish Das[1], Zafar Ahmad[1], Stephen Tschudi[3], and Rezaul Chowdhury[1](✉)

[1] Computer Science, Stony Brook University, Stony Brook, NY, USA
rezaul@cs.stonybrook.edu

[2] Computer Science and Engineering, Indian Institute of Technology, Indore, India

[3] Google Inc., Mountain View, CA, USA

Abstract. We argue that the recursive divide-and-conquer paradigm is highly suited for designing algorithms to run efficiently under both shared-memory (multi- and manycores) and distributed-memory settings. The depth-first recursive decomposition of tasks and data is known to allow computations with potentially high temporal locality, and automatic adaptivity when resource availability (e.g., available space in shared caches) changes during runtime. Higher data locality leads to better intra-node I/O and cache performance and lower inter-node communication complexity, which in turn can reduce running times and energy consumption. Indeed, we show that a class of grid-based parallel recursive divide-and-conquer algorithms (for dynamic programs) can be run with provably optimal or near-optimal performance bounds on fat cores (cache complexity), thin cores (data movements), and purely distributed-memory machines (communication complexity) without changing the algorithm's basic structure.

Two-way recursive divide-and-conquer algorithms are known for solving dynamic programming (DP) problems on shared-memory multicore machines. In this paper, we show how to extend them to run efficiently also on manycore GPUs and distributed-memory machines.

Our GPU algorithms work efficiently even when the data is too large to fit into the host RAM. These are external-memory algorithms based on recursive r-way divide and conquer, where r (≥ 2) varies based on the current depth of the recursion. Our distributed-memory algorithms are also based on multi-way recursive divide and conquer that extends naturally inside each shared-memory multicore/manycore compute node. We show that these algorithms are work-optimal and have low latency and bandwidth bounds.

We also report empirical results for our GPU and distribute memory algorithms.

Keywords: GPU · Recursive divide & conquer · Dynamic programming · Exascale · Distributed memory · Shared memory · I/O efficiency · Communication efficiency

1 Introduction

Many of the world's current fastest supercomputers are networks of distributed-memory hybrid compute nodes where each node houses both latency optimized multicores (a.k.a. fat cores) and throughput optimized manycores (a.k.a. thin cores, e.g., GPU cores) connected through a multilevel memory hierarchy [4][1] which is also what an exascale supercomputer is expected to look like in the near future [41,54]. In addition to allowing various types of parallelism, e.g, distributed-memory, shared-memory, task (on multicores) and data (on manycores), a program running on these supercomputers must exploit data locality at various levels of computation for efficiency. Indeed, higher data locality leads to better intra-node I/O and cache performance and lower inter-node communication complexity, which in turn can reduce running times and lower energy consumption.

We argue in this paper that the recursive divide-and-conquer paradigm is highly suited for designing efficient algorithms for both shared-memory and distributed-memory architectures. The depth-first recursive decomposition of tasks and data is known to allow computations with potentially high temporal locality, and automatic adaptivity when resource availability (e.g., available space in shared caches [13,19]) changes during runtime. Indeed, we show that a class of grid-based parallel recursive divide-and-conquer algorithms for solving dynamic programming problems can be run with provably optimal or near-optimal performance bounds on fat cores (cache complexity), thin cores (data movements), and purely distributed-memory machines (communication complexity) without any change in the algorithm's basic structure.

Dynamic programming (DP) [12,22] is a widely used algorithm design technique for solving optimization problems that can be decomposed into overlapping subproblems whose optimal solutions can be combined to obtain an optimal solution to the original problem. DP is extensively used in computational biology [32,73], and in many other application areas including operations research, compilers, sports, economics, finance, and agriculture (see DP refs in [19]).

Dynamic programs are typically implemented using nested loops that fill out the cells of a DP table using already computed values for other cells. However, such a looping code is usually not suitable for high performance on a modern computer with a memory hierarchy as without any *temporal locality*[2] in its data access pattern it often spends significantly more time in data transfers than in actual computations.

Tiled looping codes reduce the number of data transfers between two specific (adjacent) levels of the memory hierarchy by tiling the DP table so that a

[1] As of November 2018, the supercomputers ranked 1 (Summit), 2 (Sierra), 6 (ABCI), 7 (Piz Daint), and 8 (Titan) in order of *Rpeak* (TFlop/s) are networks of hybrid CPU+GPU nodes [4].

[2] Temporal locality — whenever a block of data is brought into a faster level of cache/memory from a slower level, as much useful work as possible is performed on this data before removing the block from the faster level.

constant number of such tiles completely fit in the smaller[3] of the two levels. Whenever a tile is brought into the smaller memory level, as much computation as possible is done with it before replacing it with another tile. In this approach, the code must know the size of the smaller memory level. The tiled code often differs significantly from the standard looping code both in structure and in complexity. Multilevel iterative tiling is possible, but at the expense of significantly increasing the complexity of the code with each additional level. Fixed tile sizes can be problematic when the program shares the smaller memory level with multiple other concurrently running programs because it can adapt neither automatically nor efficiently as the memory space available to it keeps changing during running time [13,19].

Recursive parallel 2-way divide-and-conquer DP algorithms perform an asymptotically optimal number of data transfers between every two adjacent levels of [19,69]. They do not need to know the sizes of the memories in the hierarchy, can passively self-adapt to cache sharing [13,19], and the complexity of the code is independent of the depth of the memory hierarchy. For any given pair of adjacent memory/cache levels the bounds hold under the *ideal cache* model [29] with a fully automatic optimal offline cache replacement policy. LRU replacement policy also works. It has been shown very recently that for a large class of DP problems these recursive parallel algorithms can be generated automatically [19,37].

Recursive 2-way divide-and-conquer algorithms are not suitable for GPUs as those devices have very limited support for recursion and require the programmer to explicitly transfer data between memory levels (e.g., between global and shared memories). Explicit communications among compute nodes are also required during distributed-memory computations. Moreover, these algorithms may lose parallelism because of artificial dependencies among subtasks [18].

Our Contributions. In this paper, we show how to extend 2-way recursive divide-and-conquer algorithms designed to solve DP problems efficiently on shared-memory multicore machines to run efficiently also on manycore GPUs and distributed-memory machines. The same algorithm without any changes in its basic structure runs with provable efficiency on all three platforms. Our approach works for the wide *fractal DP* class [19] that includes Floyd-Warshall's APSP, the parenthesis problem, pairwise sequence alignment, and the gap problem among many others.
(*i*) **[GPU Algorithms].** We design I/O-optimal algorithms for the *fractal DP* class [19].

[3] I.e., faster and closer to the processing core(s).

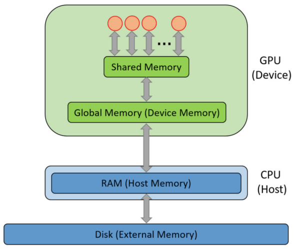

Fig. 1. Memory hierarchy assumed by our GPU algorithms.

Our approach works for arbitrarily deep memory hierarchies. But in this paper, we target the one shown in Fig. 1. We assume that the input DP table is stored either in the RAM or in the disk. Our algorithms are based on r-way recursive divide-and-conquer, where r varies based on the level of recursion (i.e., different levels can have different r values). We use $r = 2$ at every level of recursion until the input size drops below the size of the GPU global memory. At that level we choose r large enough so that the input is decomposed into chunks that are small enough to fit in the GPU shared memory. We do not use the knowledge of the CPU RAM size as the RAM can be maintained as a fully associative memory with an automatic LRU page replacement policy (through STXXL [1]). Through the choice of r we basically resort to iterative tiling once we reach inside the GPU as the device provides only limited support for recursion.

We prove theoretical bounds showing that we perform an asymptotically optimal number of data transfers between every two adjacent levels of the memory hierarchy.

We have implemented our GPU algorithms for four DP/DP-like problems: Floyd-Warshall's APSP, parenthesis problem, gap problem, and Gaussian elimination without pivoting. Our programs run significantly faster than all internal-memory multicore CPU implementations and almost all existing tiled GPU implementations.

(*ii*) **[Distributed-memory Algorithms].** Our distributed-memory algorithms are also based on r-way recursive divide and conquer that extends naturally inside each shared-memory multicore/manycore compute node. Thus these algorithms are, indeed, hybrid distributed-shared-memory algorithms. We show that they are work-optimal and have latency and bandwidth bounds that are within $\log p$ factor of optimal, where p is the number of compute nodes.

We include empirical performance results for Floyd-Warshall's APSP, the parenthesis problem and the gap problem.

Organization. Section 2 presents our approach for designing r-way algorithms. We describe our GPU results in Sect. 3 with the computing model, related work, algorithm design, I/O complexities, and experimental results in Sects. 3.1, 3.2, 3.3, 3.4 and 3.5, respectively. Our distributed-memory results are presented in Sect. 4 with the algorithm design, communication lower bounds, related work, and experimental results in Sects. 4.1, 4.2, 4.3, and 4.4, respectively. Finally, we conclude in Sect. 5.

2 Multi-way Recursive Divide and Conquer

In this section, we describe our methodology for designing multi-way (i.e., r-way) recursive divide-and-conquer algorithms for DP problems in the fractal-DP class [19]. The 2-way and r-way algorithms will be called 2-way and r-way $\mathcal{R}$-$\mathcal{DP}$s, respectively.

We will explain our methodology using the parenthesis DP [17,31] which fills out a 2D table $C[0:n, 0:n]$ based on the following recurrence:

$$C[i,j] = \min_{i \le k \le j} \{C[i,k] + C[k,j] + w(i,j,k)\} \ \text{ for } 0 \le i < j-1 < n;$$

assuming $C[i,j] = \infty$ for $0 \le i = j \le n$ and $C[i,j] = x_j$ for $0 \le i = j-1 < n$, where, x_j's are given constants and $w(i,j,k)$ does not incur any I/Os.

The class of problems defined by the recurrence above includes optimal chain matrix multiplication, RNA secondary structure prediction, optimal polygon triangulation, string parsing for context-free grammar, and optimal database joins among others. A 2-way $\mathcal{R}$-$\mathcal{DP}$ for the problem can be found in [19].

2.1 r-way $\mathcal{R}$-$\mathcal{DP}$ Design

We first use either Autogen [19] or Bellmania [37] to automatically derive the standard 2-way $\mathcal{R}$-$\mathcal{DP}$ for the given DP problem. We then derive an r-way $\mathcal{R}$-$\mathcal{DP}$ from the 2-way $\mathcal{R}$-$\mathcal{DP}$. Indeed, assuming $r = 2^t$ for some positive integer t, each level of recursion of an r-way $\mathcal{R}$-$\mathcal{DP}$ can be obtained by unrolling t levels of recursion of the corresponding 2-way $\mathcal{R}$-$\mathcal{DP}$.The resulting r-way $\mathcal{R}$-$\mathcal{DP}$s typically have more parallelism than their 2-way counterparts (see Figure 3 in [64] for an example).

To obtain a multi-way $\mathcal{R}$-$\mathcal{DP}$ from a 2-way $\mathcal{R}$-$\mathcal{DP}$, we start with $t = 1$ and keep applying the following two refinement steps until we can identify the pattern in which the recursive functions are called in the resulting unrolled $\mathcal{R}$-$\mathcal{DP}$.

Step 1. Take the 2^t-way $\mathcal{R}$-$\mathcal{DP}$, and unroll each recursive function call by one level (of recursion) based on the 2-way $\mathcal{R}$-$\mathcal{DP}$ version of that function.
Step 2. To execute the recursive function calls in the unrolled version from step 1 in as few stages as possible, we move each such function call to the lowest possible stage without violating dependency constraints. We follow the following rules where by $\mathbf{W}(\mathcal{F})$ we denote the DP subtable function $\mathcal{F}$ writes to and by $\mathbf{R}(\mathcal{F})$ we denote the set of DP subtables $\mathcal{F}$ reads from. We say that $\mathcal{F}$ is *flexible* provided $\mathbf{W}(\mathcal{F}) \notin \mathbf{R}(\mathcal{F})$, and *inflexible* otherwise. By $\mathcal{F}_1 \rightarrow \mathcal{F}_2$ we mean that $\mathcal{F}_1$ is executed before $\mathcal{F}_2$, $\mathcal{F}_1 \leftrightarrow \mathcal{F}_2$ means that order does not matter, and $\mathcal{F}_1 || \mathcal{F}_2$ indicates parallel execution.

#1: If $\mathbf{W}(\mathcal{F}_1) \neq \mathbf{W}(\mathcal{F}_2)$ and $\mathbf{W}(\mathcal{F}_1) \in \mathbf{R}(\mathcal{F}_2)$, then $\mathcal{F}_1 \rightarrow \mathcal{F}_2$.
#2: If $\mathbf{W}(\mathcal{F}_1) = \mathbf{W}(\mathcal{F}_2)$ and only $\mathcal{F}_1$ flexible, then $\mathcal{F}_1 \rightarrow \mathcal{F}_2$.
#3: If $\mathbf{W}(\mathcal{F}_1) = \mathbf{W}(\mathcal{F}_2)$ and both $\mathcal{F}_1$ and $\mathcal{F}_2$ are flexible, then $\mathcal{F}_1 \leftrightarrow \mathcal{F}_2$.
#4: If $\mathcal{F}_1$ and $\mathcal{F}_2$ satisfy none of the rules above, then $\mathcal{F}_1 || \mathcal{F}_2$.

The new 2^{t+1}-way $\mathcal{R}$-$\mathcal{DP}$ has potentially more parallelism than its 2^t-way version.

```
$\mathcal{A}_{par}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. if $r \geq m$ then $\mathcal{A}_{loop\text{-}par}(X, U, V)$
   else
3.   $\delta \leftarrow (j - i), d' \leftarrow d + 1$
4.   parallel: $\mathcal{A}_{par}(X_{i,j}, U_{i,j}, V_{i,j}, d')\ \forall i, j \in [1, r], \delta = 0$
5.   for $k \leftarrow 1$ to $r - 1$ do
6.     parallel: $\mathcal{C}_{par}(X_{i,j}, U_{i,i+k-1}, V_{i+k-1,j}, d')$
                 $\forall i, j \in [1, r],\ \delta \in [k, \min\{2k - 2, r - 1\}]$
7.     parallel: $\mathcal{C}_{par}(X_{i,j}, U_{i,j-k+1}, V_{j-k+1,j}, d')$
                 $\forall i, j \in [1, r],\ \delta \in [k, \min\{2k - 3, r - 1\}]$
8.     parallel: $\mathcal{B}_{par}(X_{i,j}, U_{i,i}, V_{j,j}, d')\ \forall i, j \in [1, r],\ \delta = k$

$\mathcal{B}_{par}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. if $r \geq m$ then $\mathcal{B}_{loop\text{-}par}(X, U, V)$
   else
3.   Let $\langle U'_{i,\ell}, V'_{\ell,j}\rangle = \begin{cases} \langle X_{i,\ell}, V_{\ell,j}\rangle & \text{if } \ell > 0, \\ \langle U_{i,\ell+r}, X_{\ell+r,j}\rangle & \text{if } \ell \leq 0. \end{cases}$
4.   $\delta \leftarrow (j - i), d' \leftarrow d + 1, r' \leftarrow 2r - 1$
5.   for $k \leftarrow 1$ to $r'$ do
6.     parallel: $\mathcal{C}_{par}(X_{i,j}, U'_{i,k-r+i-1}, V'_{k-r+i-1,j}, d')$
                 $\forall\, i, j \in [1, r], \delta + r \in [k, \min\{2k - 2, r'\}]$
7.     parallel: $\mathcal{C}_{par}(X_{i,j}, U'_{i,1+i-r}, V'_{1+i-r,j}, d')$
                 $\forall\, i, j \in [1, r], \delta + r \in [k, \min\{2k - 3, r'\}]$
8.     parallel: $\mathcal{B}_{par}(X_{i,j}, U_{i,i}, V_{j,j}, d')$
                 $\forall\, i, j \in [1, r], \delta + r = k$

$\mathcal{C}_{par}(X, U, V, d)$

1. $r \leftarrow tilesize[d]$
2. if $r \geq m$ then $\mathcal{C}_{loop\text{-}par}(X, U, V)$
   else
3.   for $k \leftarrow 1$ to $r$ do
4.     parallel: $\mathcal{C}_{par}(X_{i,j}, U_{i,k}, V_{k,j}, d + 1)\ \forall i, j \in [1, r]$
```

Fig. 2. An r-way $\mathcal{R}$-$\mathcal{DP}$ for parenthesis problem [20]. Here, X, U and V are $m \times m$ tables.

Based on the dimension m of the DP (sub-)table(s) at any given level of recursion of an r-way $\mathcal{R}$-$\mathcal{DP}$, r can be set to a constant or a function of either m or both m and a particular cache or memory size such that the resulting tile exactly fits into that memory. When a subproblem fits into a memory of the smallest size, we execute an iterative kernel. Given the original DP table dimension n we precompute the value of r at each recursion level d and store that in $tilesize[d]$.

In Fig. 2 we show an r-way $\mathcal{R}$-$\mathcal{DP}$ for the parenthesis problem with functions $\mathcal{A}_{par}$, $\mathcal{B}_{par}$, and $\mathcal{C}_{par}$. The initial function call is $\mathcal{A}_{par}(C, C, C, 1)$, where C is the input DP table. The term m in all the functions represents the dimension length at a particular recursion level. The keyword **parallel** means that the functions can be invoked in parallel (Fig. 3).

2.2 Additional r-way $\mathcal{R}$-$\mathcal{DP}$ Algorithms

In this work, we have designed and implemented r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms for the following three additional problems.

Gaussian Elimination w/o Pivoting. This DP-like algorithm is used for solving systems of linear equations and LU decomposition of symmetric positive-definite or diagonally dominant real matrices [22].

Floyd-Warshall's APSP. This all-pairs shortest path algorithm [22] uses the recurrence below. Let $D[i, j, k]$ be the length of the shortest path from vertex v_i to vertex v_j with no intermediate vertex higher than v_k. Let $\ell(i, j)$ be the distance between v_i and v_j. Then $D[i, j, k] = 1$ if $k = 0$ and $i = j$; $D[i, j, k] = \ell(v_i, v_j)$ if $k = 0$ and $i \neq j$; and $D[i, j, k] = \min(D[i, j, k-1], D[i, k, k-1] + D[k, j, k-1])$ if

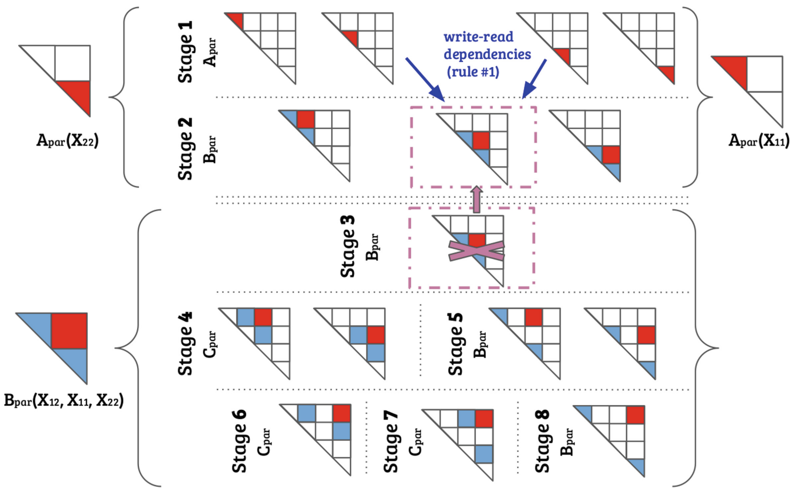

Fig. 3. Reducing the number of parallel stages in $\mathcal{A}_{par}$ after unrolling the recursive function calls by one level.

$k > 0$. The third dimension of D can be dropped to solve the problem in space quadratic in the number of vertices in the graph.

Sequence Alignment with Gap Penalty (Gap Problem). In this problem [30,31,73], a sequence of consecutive deletes or inserts corresponds to a gap and they are handled with generic cost functions w and w'. The optimal alignment cost for strings $X = x_1x_2 \ldots x_m$ and $Y = y_1y_2 \ldots y_n$ is defined by the following recurrence: $G[i,j] = 0$ if $i = j = 0$, $G[i,j] = w(0,j)$ if $i = 0 \ \wedge \ j \in [1,n]$, $G[i,j] = w'(i,0)$ if $j = 0 \wedge i \in [1,m]$, and $G[i,j] = \min\{G[i-1,j-1]+S(x_i,y_j), \min_{0 \le q < j} \{G[i,q] + w(q,j)\}, \min_{0 \le p < i} \{G[p,j] + w'(p,i)\}\}$ otherwise; where, w and w' do not incur any I/Os.

3 External-Memory GPU Algorithms

3.1 GPU Computing Model

We give a brief overview of the GPU architecture, its programming model, and GPU programming challenges.

General Purpose Computing on GPUs. GPUs are attached to CPUs through PCI bus as hardware accelerators. They have a manycore architecture with hundreds to thousands of cores, and are designed to have thousands of light-weight threads, perform highly data-parallel and compute-intensive tasks, and maximize the throughput of the parallel programs. GPUs support multi-threading, SIMD, and instruction-level parallelism.

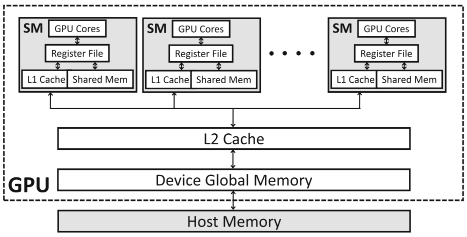

Fig. 4. Organization of an NVIDIA GPU.

An NVIDIA GPU is a set of Streaming Multiprocessors (SMs) employing an SIMT computational architecture. Each SM consists of many processing cores connected to a shared memory/L1 cache. The SMs are connected to the device (global) memory through an L2 cache. Figure 4 shows this memory organization.

The most commonly used APIs for general purpose computing on GPUs include OpenCL, NVIDIA CUDA, Microsoft DirectCompute, OpenACC, and AMD's APP SDK.

GPU Programming Challenges. Recursion and divide-and-conquer are powerful tools for designing efficient (I/O-efficient, energy-efficient, and highly parallel), portable (cache- and processor-oblivious) and robust (cache- and processor-adaptive) algorithms. However, these design techniques involve complicated control logic and hence they are either unsupported or have very limited support in GPUs.

Optimizing a GPU program is hard as many factors have big influence on its performance: thread organization (in blocks of different dimensions with different dimension lengths), warp size (the granularity at which the SMs can execute computations), memory coalescing (consecutive numbered threads access consecutive memory locations), and streams and events (overlapping compute kernel execution and data transfers).

3.2 Related Work (GPU)

Several GPU algorithms exist that solve DP problems: Floyd-Warshall's APSP [14,26,27,34,40,47,49,58,71], parenthesis problem family [51–53,56,61,74], and sequence alignment [45,46,48,62,75]. Most of them are loops tiled for GPU global and shared memories to exploit temporal locality. Some are based on tiling derived from recursive divide-and-conquer algorithms that use only matrix-matrix multiplications on a semiring, e.g., R-Kleene's algorithm [14,23,55] for Floyd-Warshall's APSP. Major limitations of existing results are as follows. First, almost all existing GPU algorithms assume that the entire DP table fits into the GPU global memory, and none of them work when the table is too large for the host RAM. Thus, the size of the problem they can handle is limited by the size of one of those two levels of memory. Second, no general methodology is known that work for a large class of DP problems. Third, theoretical performance guarantees for data transfers and parallelism are often missing.

3.3 GPU Algorithm Design

We will explain how to port the r-way $\mathcal{R}$-$\mathfrak{DP}$ given in Fig. 2 to a GPU system. The approach works for all fractal-DP problems. For simplicity, we assume the

4-level memory hierarchy shown in Fig. 1. Handling deeper hierarchies, multiple GPUs, and multiple shared memories connected to a global memory are not difficult.

Let us first assume that we know the sizes of the CPU RAM and both GPU memories, and let the input DP table be present in the external memory. Hence, the data from the DP table will pass through CPU RAM, GPU global memory, and GPU shared memory. We define functions host_disk_$\mathcal{A}_{par}$, host_RAM_$\mathcal{F}_{par}$, device_global_$\mathcal{F}_{par}$, and device_shared_$\mathcal{F}_{par}$, where $\mathcal{F} \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$. The suffixes $\mathcal{A}_{par}$, $\mathcal{B}_{par}$, and $\mathcal{C}_{par}$ correspond to the three recursive functions. Functions with keywords host and device run on the CPU and GPU, respectively. Input and the output matrices accessed by functions with keywords disk, RAM, global, and shared reside on CPU disk, CPU RAM, GPU global memory, and GPU shared memory, respectively.

Initially, host_disk_$\mathcal{A}_{par}$ is invoked with the entire DP table as input. The function splits the $n \times n$ DP table into $r_d \times r_d$ subtables each of size $(n/r_d) \times (n/r_d)$, assuming r_d divides n for simplicity. The value of r_d is chosen such that the input subtables for the function exactly fit in the RAM. The function invokes host_RAM_$\mathcal{F}_{par}$, where $\mathcal{F} \in \{\mathcal{A}, \mathcal{B}, \mathcal{C}\}$, as per the r-way $\mathcal{R}$-$\mathcal{DP}$ algorithm, after copying the subtables required by that child function to RAM. We do not define host_disk_$\mathcal{B}_{par}$ and host_disk_$\mathcal{C}_{par}$ as they will never be invoked.

Function host_RAM_$\mathcal{F}_{par}$ splits each of its $(n/r_d) \times (n/r_d)$ sized input/output tables into $r_m \times r_m$ subtables each of size $(n/(r_d r_m)) \times (n/(r_d r_m))$, assuming r_m divides (n/r_d) for simplicity. It invokes appropriate functions device_global_$\mathcal{F}_{par}$ after copying the relevant subtables to the GPU global memory. This process continues till the functions device_shared_$\mathcal{F}_{par}$ are reached. Inside these functions we execute the looping kernels using GPU cores.

Now let's assume that we do not know the size of the CPU RAM, but it is maintained as a fully associative memory with an automatic LRU page replacement policy. Then instead of host_disk_$\mathcal{A}_{par}$ and host_RAM_$\mathcal{A}_{par}$ we will only have host_$\mathcal{A}_{par}$, and similarly host_$\mathcal{B}_{par}$ and host_$\mathcal{C}_{par}$. Initially, the function host_$\mathcal{A}_{par}$ is invoked with the entire DP table as input. The function splits the entire $n \times n$ DP table into 2×2 subtables each of size $(n/2) \times (n/2)$, assuming n is divisible by 2 for simplicity. Now if a $(n/2) \times (n/2)$ subtable fits into the GPU global memory we invoke device_global_$\mathcal{F}_{par}$, otherwise we recursively invoke host_$\mathcal{F}_{par}$.

3.4 I/O Complexities

We present theoretical bounds on the I/O's performed by our GPU algorithms.

Let M_m, M_g, and M_s be the sizes of the CPU main memory, GPU global memory, and GPU shared memory, respectively, and suppose these sizes are known to the algorithm. So there will be exactly three levels of recursion, and in each level the algorithm will choose the largest tile size (i.e., the smallest possible value of r) such that the required number (a constant) of tiles fit in the next smaller level of memory. Let B, B_m, B_g, and B_s denote the block sizes between disk and RAM, RAM and global memory, global memory and shared memory, and shared memory and processor, respectively. All M's, n's, and B's are natural numbers.

Theorem 1 (I/O complexity of GPU algorithms). *When run on the GPU memory hierarchy of Fig. 1, the number of data blocks transferred by the external-memory GPU algorithm (i.e., I/O complexity) between:* ***(a)*** *disk & RAM:* $\Theta\left(\frac{n^w}{BM_m^{w/d-1}} + \frac{n^w}{M_m^{(w+1)/d-1}}\right)$, ***(b)*** *RAM & global memory:* $\Theta\left(\frac{n^w}{B_m M_g^{w/d-1}} + \frac{n^w}{M_g^{(w+1)/d-1}}\right)$, *and* ***(c)*** *global & shared memories:* $\Theta\left(\frac{n^w}{B_g M_s^{w/d-1}} + \frac{n^w}{M_s^{(w+1)/d-1}}\right)$; *where,* $\Theta\left(n^w\right)$ *is the total work (i.e., time spent in computation only) performed by the GPU algorithm, and* n^d *is the size of the original input DP table.*

Proof. We assume that the number of submatrices accessed by each recursive function is upper bounded by a constant. Let n_q be the largest tile dimension a function can use while still making sure that the required number of tiles fit into a memory of size M_q. Then $n_m^d = \Theta\left(M_m\right)$, $n_g^d = \Theta\left(M_g\right)$ and $n_s^d = \Theta\left(M_s\right)$.

Let M_L and M_S be the sizes of two adjacent levels of memory and $M_L \geq M_S$. Let B_L be the block transfer size between the two levels. Then the I/O-complexity of filling the smaller memory once is $\mathcal{O}\left(n_S^{d-1}(n_S/B_L + 1)\right)$. The smaller memory will be filled $\Theta\left((n_L/n_S)^w\right)$ times. Hence, the I/O-complexity between the two memories is $\mathcal{O}\left((n_L/n_S)^w n_S^{d-1}(n_S/B_L + 1)\right)$.

We now apply the result above to prove the theorem. The I/O-complexity between disk and RAM is $\mathcal{O}((n/n_m)^w n_m^{d-1}\ (n_m/B + 1))$. The I/O-complexity between RAM and global memory to work on all data present in RAM is $\mathcal{O}((n_m/n_g)^w n_g^{d-1}\ (n_g/B_m + 1))$. However, the RAM will be filled $\Theta\left((n/n_m)^w\right)$ times. Hence, the total I/O-complexity between RAM and global is $\mathcal{O}((n/n_g)^w n_g^{d-1}\ (n_g/B_m + 1))$. We use a similar reasoning to compute the total I/O-complexity between global and shared memories. Putting $n_m^d = \Theta\left(M_m\right)$, $n_g^d = \Theta\left(M_g\right)$ and $n_s^d = \Theta\left(M_s\right)$ we obtain the claimed bounds.

I/O Optimality. The optimality of the I/O bounds presented in Theorem 1 for any given r-way $\mathcal{R}$-$\mathcal{DP}$ follows from the known optimality of the corresponding 2-way $\mathcal{R}$-$\mathcal{DP}$ proved under the ideal cache model [29]. We compute the I/O complexity of an r-way $\mathcal{R}$-$\mathcal{DP}$ as the product of the number of subproblems that exactly fit into the memory and the number of block transfers required to scan the input and output matrices for a subproblem. Say $r = 2^t$ for some $t \in \mathbb{N}$. Then, to compare the I/O complexities of the two algorithms, the r-way $\mathcal{R}$-$\mathcal{DP}$ can be viewed as the 2-way $\mathcal{R}$-$\mathcal{DP}$ unrolled t times. The number of subproblems that exactly fit in the memory will be asymptotically same for both 2-way and r-way $\mathcal{R}$-$\mathcal{DP}$s. Also, the I/Os required to scan the matrices that exactly fit in the memory will also be asymptotically the same for both $\mathcal{R}$-$\mathcal{DP}$s. Hence, the I/O complexities of the two $\mathcal{R}$-$\mathcal{DP}$s will match.

3.5 GPU Experimental Results

We present empirical results showing the performance benefits of our GPU algorithms.

Setup. All our experiments were performed on a heterogeneous node of the Stampede supercomputer [2,70]. The multicore machine had a dual-socket 8-core 2.7 GHz Intel Sandy Bridge processors ($2 \times 8 = 16$ cores in total) and 32 GB RAM. Each core was connected to a 32 KB private L1 cache and a 256 KB private L2 cache. All cores in a processor shared a 20 MB L3 cache. The node was attached to a single NVIDIA K20 GPU. The GPU had an on-board GDDR5 memory of 5 GB, and 2496 CUDA cores.

All our algorithms were implemented in C++. We used Intel Cilk Plus extension to parallelize and Intel® C++ Compiler v13.0 to compile the CPU implementations with optimization parameters -O3 -ipo -parallel -AVX -xhost. Our GPU programs were written in CUDA. The programs were compiled with nvcc compiler with parameters -O3 -gencode arch=compute_35,code=sm_35.

Implementations. We focus on four DP/DP-like problems: Floyd-Warshall's APSP (FW-APSP), Gaussian elimination without pivoting [21], parenthesis problem [17,31], and sequence alignment with gap penalty (gap problem) [30,31].

For all problems we consider the following two implementations where cpu and gpu prefixes are used to indicate programs written for CPUs and GPUs, respectively:

For FW-APSP, we also consider: (*iii*) gpu-tidp-harish: Harish and Narayanan's [34] tiled-iterative code, (*iv*) gpu-tidp-lund: Lund and Smith's [47] tiled-iterative code, (*v*) gpu-tidp-katz: Katz and Kider's [40] tiled-iterative code, (*vi*) gpu-rec-buluc: Buluc et al.'s implementation of the 2-way R-Kleene algorithm with Volkov and Demmel's optimization [72] for the matrix multiplication (MM) kernel, and (*vii*) gpu-rdp-opt: r-way $\mathcal{R}$-$\mathcal{DP}$ replaced with Buluc et al.'s MM-like kernel for MM-like functions (i.e., functions reading from and writing to disjoint matrices).

For the other three problems (i.e., parenthesis, Gaussian elimination w/o pivoting, and gap) we could not find any publicly available GPU codes for comparison.

Optimizations. We list below the optimizations we applied on various programs in addition to the compiler optimizations enabled by the optimization flags we used.

Major optimizations applied on gpu-rdp and gpu-rdp-opt are as follows.

(*i*) We used GPU shared memory by setting BLOCK_SIZE $= 32$ so that 1024 threads could work on matrices of size 32×32 simultaneously. Also, two blocks with 1024 threads each were run in parallel. But since NVIDIA K20 can run up to 2496 hardware threads at a time, 448 threads remained unused. Use of more than 2048 threads required dropping to 16×16 or some non-power-of-2 size, and then either run into extra overhead for launching jobs or be way under the 48KB shared memory limit per block. This ended up being the bigger bottleneck on the system and our preliminary experiments showed that even with 256 more threads 16×16 was worse than 32×32.

(*ii*) All our DP applications have very predictable data access patterns, and so a user-managed shared-memory seems more appropriate for them than an L1 cache. But we tried both Shared Memory/L1 cache configurations and also tried varying threads, and the best configuration was 48KB shared memory with 2048 threads. Our most memory-hungry computations access three disjoint matrices. Hence, a block with $\mathsf{BLOCK\_SIZE} = 32$ and single precision floats uses $3 \times \frac{32 \times 32 \times 4}{1024} = 12\,\text{KB}$ of the shared memory and with double precision floats uses 24KB. Though $\mathsf{BLOCK\_SIZE} = 64$ with single precision floats will fill up the 48KB shared memory, we won't have enough threads to compute all $64 \times 64 = 4096$ output cells in parallel. Our preliminary experiments showed that 32 was a better choice for BLOCK_SIZE than 64.

(*iii*) If a function kernel reads only from submatrices it is not writing to, then we do not synchronize inside it.

(*iv*) Row-major order was used for all submatrices. Flipping a submatrix to column-major degrades performance. Row-major was used for the grid and inside each block.

(*v*) GRID_SIZE was set to $\min\{n, 16384\}$, where 16384 was the maximum size such that our subproblems can exactly fit in the 5 GB of global memory.

(*vi*) Allocating memory using gpuMalloc() on GPU global memory is slow. Instead we simply malloc once and then copy the submatrices to the respective regions.

(*vii*) We allocate directly in the host's pinned memory using cudaMallocHost(). This reduces the block transfers between pageable host memory and pinned memory.

With more aggressive optimizations gpu-rdp and gpu-rdp-opt are likely to perform even better than what we report in this paper. However, we want to show that GPU algorithms designed based on recursive divide-and-conquer are I/O-efficient and remain compute-bound for modern GPUs even when the data is in external-memory. Once that goal is reached, additional optimizations will only improve the results.

Additional optimizations used for cpu-rdp include:

(*i*) #pragma parallel, #pragma ivdep, and min loop count(B),

(*ii*) 64 byte-aligned matrices,

(*iii*) write optimization in the basecase – if an innermost loop updates the same DP table cell repeatedly we apply all of them on a register instead of the DP cell, and update that cell only once at the end of the loop using the value in the updated register,

(*iv*) copy optimization in the basecase – copying the transpose of a column-major input matrix to a local array in order to access it in unit stride during computation, *(v)* pointer arithmetic, and

(vi) Z-morton layout (only for the gap problem). Each of our DP applications in this paper runs in $\Theta\left(n^3\right)$ time which asymptotically dominates the $\Theta\left(n^2\right)$ time needed for layout conversion of the input matrix. Indeed, we have found that the layout conversion time has very little impact on the actual running times.

The three tiled-iterative implementations of FW APSP used 32×32 tiles.

Internal-Memory Results. Figure 5 shows the speedup of various programs w.r.t. cpu-rdp for four DP problems. For each program, the DP table dimension n is varied from 2^{10} to 2^{15}. For single precision floats $n = 2^{15}$ is the largest power of 2 for which an $n \times n$ DP table (using 4 GB space) completely fits inside the 5 GB GPU global memory.

For FW-APSP, gpu-rdp-opt was the second fastest running program with gpu-rec-buluc running 1.5 times faster for $n = 2^{15}$. This is because unlike gpu-rec-buluc, all kernels of gpu-rdp-opt were not MM-like and hence it ran slower than Buluc et al.'s implementation. While our gpu-rdp and gpu-rdp-opt continued to run for $n = 2^{16}$ and beyond, none of the other GPU implementations did as they did not have mechanisms to expand beyond the GPU global memory.

When $n = 2^{15}$, our gpu-rdp programs for the Gaussian elimination, parenthesis and gap problems ran $1.3\times$, $11\times$, and $2.8\times$ faster, respectively, than their cpu-rdp counterparts. The speedup factors achieved by the GPU algorithms for the parenthesis and gap problems are higher than that for FW-APSP/Gaussian elimination because gpu-rdp for the former two problems have higher parallelism than the latter two.

External-Memory Results. It is easy to extend our algorithms to work for DP tables that are too large to fit in the CPU RAM and hence must be stored in external-memory (or disks). We can use either a 2-way or an r-way $\mathcal{R}$-$\mathcal{DP}$ for external-memory until a subproblem fits in the GPU global memory, after which we use an r-way $\mathcal{R}$-$\mathcal{DP}$ between GPU global memory and GPU shared memory. When an r-way $\mathcal{R}$-$\mathcal{DP}$ is used between two levels of memory, r is chosen as the smallest integer such that if an $m \times m$ DP (sub-)table is stored in the larger memory dividing it into tiles of size $(m/r) \times (m/r)$ each will make sure that $1 + s$ such tiles completely fit in the smaller memory, where s is the maximum number of additional tiles one must read from to update one tile. Using a 2-way $\mathcal{R}$-$\mathcal{DP}$ between the external-memory and the GPU global memory makes our algorithm oblivious of the CPU RAM size provided an appropriate automatic page replacement protocol is functional between the external-memory and the CPU RAM.

We use Standard Template Library for Extra Large Data Sets (STXXL) [1] 1.4.1 to implement our algorithms for external-memory. STXXL is a C++ library for implementing containers and algorithms that process vast amounts of disk data. In STXXL, we set the external block size as 4 MB, #pages as 1024, and #blocks per page as 1. This gives us a RAM of size 4 GB. STXXL maintains the CPU RAM as a fully associative memory with an automatic LRU page replacement policy.

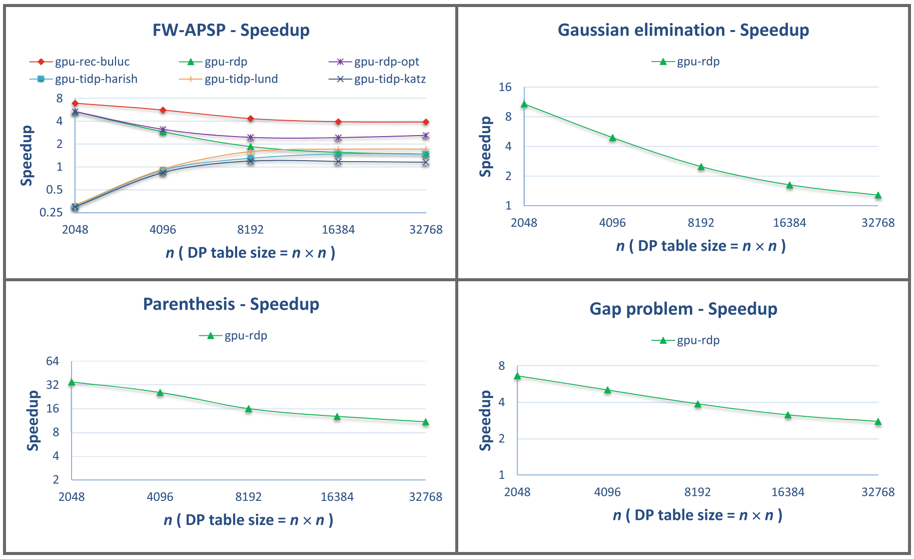

Fig. 5. Speedup of gpu-rdp programs over cpu-rdp for various dynamic programs. For FW-APSP, gpu-buluc-rec and gpu-rdp-opt are also included.

For each of the four DP problems we compare: (a) cpu-rdp-1: serial $\mathcal{R}$-$\mathcal{DP}$ running on CPU, (b) cpu-rdp-128: parallel $\mathcal{R}$-$\mathcal{DP}$ running on a CPU with 128 cores (details will follow), and (c) gpu-rdp: parallel $\mathcal{R}$-$\mathcal{DP}$ running on a GPU machine.

For gpu-rdp we store the input DP table in Z-Morton layout in the external-memory until we reach a submatrix size that fits in the GPU global memory at which point it is stored in row-major order. While the input problem accesses a single matrix in external-memory, a subproblem may access multiple submatrices of the DP table and they all have to fit in the GPU global memory. Once we compute a submatrix, we write the output to the same location in the DP table in the external-memory.

For cpu-rdp-1 and cpu-rdp-128, the base case dimension length is set to 256 and we run iterative kernels inside each base case. Since these two programs take too long to run, we approximate their running times as follows (instead of measuring time during real runs). The DP table is stored as a grid of blocks of size 16 K × 16 K each and it is stored in Z-Morton order. We use r-way $\mathcal{R}$-$\mathcal{DP}$ in external-memory and whenever a subproblem is brought to RAM, we use 2-way $\mathcal{R}$-$\mathcal{DP}$ to execute it on CPU. Observe that unlike our GPU program gpu-rdp, the two CPU programs are kept aware of the CPU RAM size in order to get faster running times. Let n_{base}, n_{base}^{128}, n_{chunk}, t_{base}, and t_{chunk} represent the number of invocations of base case kernels, number of parallel steps of execution of the base case kernels when we assume 128 cores, number of times RAM (of size 16 K × 16 K) is loaded/unloaded, minimum time taken (among several runs) to execute a base case kernel, and time taken to copy data between external-memory and RAM as given in STXXL I/O statistics, respectively. Then the running time of cpu-rdp-1 is $(n_{base} \cdot t_{base} + n_{chunk} \cdot t_{chunk})$, and that of cpu-rdp-128 is $(n_{base}^{128} \cdot t_{base} + n_{chunk} \cdot t_{chunk})$.

When $n = 2^{17}$, in our experiments for FW-APSP, Gaussian elimination, parenthesis and gap problems gpu-rdp ran 3.1×, 1×, 3.5×, and 1.6× faster, respectively, than cpu-rdp-128.

4 Distributed-Memory Algorithms

4.1 Distributed-Memory r-way $\mathcal{R}$-$\mathcal{DP}$

Our r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms can be easily modified to run efficiently on distributed-memory machines. We modify the top level of the recursion by setting r to an appropriate value based on the number of compute nodes available and adding appropriate communication instructions. Only the first level of recursion under function $\mathcal{A}$ (e.g., $\mathcal{A}_{par}$) will have a distributed-memory implementation. Every other function and every other level of recursion of $\mathcal{A}$ will run completely inside a compute node and thus will have either a multicore or a manycore implementation.

Table 1. Our distributed-memory r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms.

DP	Work	Latency	Bandwidth
LCS/Edit distance	$\mathcal{O}\left(\frac{n^2}{p}\right)$	$\mathcal{O}(\sqrt{p})$	$\mathcal{O}(n)$
Parenthesis, Floyd-Warshall's APSP, Gaussian elimination w/o pivoting, Gap, Protein accordion folding	$\mathcal{O}\left(\frac{n^3}{p}\right)$	$\mathcal{O}(\sqrt{p}\log p)$	$\mathcal{O}\left(\frac{n^2}{\sqrt{p}}\log p\right)$

```
distributed_$\mathcal{A}_{par}(X)$

1.  $d \leftarrow 1$
2.  $r \leftarrow tilesize[d]$                    {assuming $tilesize[1] = \sqrt{p}$}
3.  Split $X$ into $r \times r$ submatrices of size $\frac{n}{r} \times \frac{n}{r}$ each. The submatrix
    of $X$ at the $i$-th position from the top and the $j$-th position from the left
    is denoted by $X_{i,j}$, where $i, j \in [1, r]$.
4.  We assume that for $1 \le i, j \le r$, initially $X_{i,j}$ belongs to node $P_{i,j}$.
    STEP 1 (LINES 5–6): corresponds to Line 4 of $\mathcal{A}_{par}$ in Figure 2
5.  parallel for $i \leftarrow 1$ to $r$ do
6.    $P_{i,i}$ calls $\mathcal{A}_{par}(X_{i,i}, X_{i,i}, X_{i,i}, d+1)$
    STEP 2 (LINES 7–23): corresponds to Lines 5–8 of $\mathcal{A}_{par}$ in Figure 2
7.  for $k \leftarrow 1$ to $r-1$ do
      STEP 2.1 (LINES 8–13): corresponds to Line 6 of $\mathcal{A}_{par}$ in Figure 2
8.    parallel for $i \leftarrow 1$ to $r-k$ do
9.      $P_{i,i+k-1}$ broadcasts $X_{i,i+k-1}$ to $P_{i,i+k \ldots i+\min\{2k-2,r-i\}}$
10.   parallel for $i \leftarrow 1$ to $r-k$ do
11.     parallel for $j \leftarrow i+k$ to $i+\min\{2k-2, r-i\}$ do
12.       $P_{i+k-1,j}$ sends $X_{i+k-1,j}$ to $P_{i,j}$
13.       $P_{i,j}$ calls $\mathcal{C}_{par}(X_{i,j}, X_{i,i+k-1}, X_{i+k-1,j}, d+1)$
      STEP 2.2 (LINES 14–19): corresponds to Line 7 of $\mathcal{A}_{par}$ in Figure 2
14.   parallel for $j \leftarrow k+1$ to $r$ do
15.     $P_{j-k+1,j}$ broadcasts $X_{j-k+1,j}$ to $P_{j-\min\{2k-3,j-1\} \ldots j-k,j}$
16.   parallel for $j \leftarrow k+1$ to $r$ do
17.     parallel for $i \leftarrow j-\min\{2k-3, j-1\}$ to $j-k$ do
18.       $P_{i,j-k+1}$ sends $X_{i,j-k+1}$ to $P_{i,j}$
19.       $P_{i,j}$ calls $\mathcal{C}_{par}(X_{i,j}, X_{i,j-k+1}, X_{j-k+1,j}, d+1)$
      STEP 2.3 (LINES 20–23): corresponds to Line 8 of $\mathcal{A}_{par}$ in Figure 2
20.   parallel for $i \leftarrow 1$ to $r-k$ do
21.     $j \leftarrow i+k$
22.     $P_{i,i}$ sends $X_{i,i}$ to $P_{i,j}$, and $P_{j,j}$ sends $X_{j,j}$ to $P_{i,j}$
23.     $P_{i,j}$ calls $\mathcal{B}_{par}(X_{i,j}, X_{i,i}, X_{j,j}, d+1)$
```

Fig. 6. Distributed-memory implementation of A_{par} from Fig. 2. Here, X is an $n \times n$ DP table and p is the number of compute nodes.

We explain our approach by applying it to $\mathcal{A}_{par}$ from Fig. 2. The modified function is shown in Fig. 6. We map the given p compute nodes onto the nodes of a $\sqrt{p} \times \sqrt{p}$ grid P. We set r to $\sqrt{p}$, and split the $n \times n$ input matrix X into $r \times r$ submatrices of size $\frac{n}{r} \times \frac{n}{r}$ each. The submatrix of X (resp. compute node of P) at the i-th position from the top and the j-th position from the left is denoted by $X_{i,j}$ (resp. $P_{i,j}$), where $i, j \in [1, r]$. For $1 \le i, j \le r$, initially only $P_{i,j}$ holds

$X_{i,j}$. Only $P_{i,j}$ updates $X_{i,j}$, and all other submatrices needed for the purpose are brought to $P_{i,j}$ through either broadcasts or direct sends. Steps 1, 2.1, 2.2 and 2.3 of distributed-$\mathcal{A}_{par}$ in Fig. 6 correspond respectively to lines 4, 6, 7 and 8 of $\mathcal{A}_{par}$.

We use the following three cost metrics [60] computed along the critical path to evaluate our algorithms: (i) *computational cost* (F): #flops executed, (ii) *latency cost* (S): #messages transferred, and (iii) *bandwidth cost* (W): #words transferred. For example, for Fig. 6, $F = \mathcal{O}\left(\left(n/\sqrt{p}\right)^3\right)$ (Step 1) + $\mathcal{O}\left(\sqrt{p} \times \left(n/\sqrt{p}\right)^3\right)$ (Step 2) $= \mathcal{O}\left(n^3/p\right)$. The latency cost is dominated by the cost of broadcasts in the loop of Step 2 which iterates $\sqrt{p}$ times, and hence $S = \mathcal{O}\left(\sqrt{p}\log p\right)$. Since each message is of size $\mathcal{O}\left(n^2/p\right)$ the bandwidth cost of each broadcast is $\mathcal{O}\left(\left(n^2/p\right)\log p\right)$. Taking into account that the loop in Step 2 iterates $\sqrt{p}$ times, the overall bandwidth cost of distributed-$\mathcal{A}_{par}$ is $W = \mathcal{O}\left(\sqrt{p}\left(n^2/p\right)\log p\right) = \mathcal{O}\left(\left(n^2/\sqrt{p}\right)\log p\right)$. Table 1 lists these cost metrics for a number of distributed-memory r-way $\mathcal{R}$-$\mathcal{DP}$ algorithms.

We have designed distributed-memory r-way $\mathcal{R}$-$\mathcal{DP}$s for Floyd-Warshall's APSP, Gaussian elimination w/o pivoting and the gap problem, too.

4.2 Bandwidth and Latency Lower Bounds

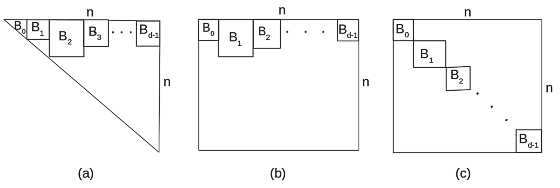

Fig. 7. Critical path in the (a) parenthesis DP evaluation, (b) GAP DP evaluation, and (c) APSP/GE DP evaluation.

In the parenthesis problem, we need to compute the cells in the triangle as shown in Fig. 7(a). The sequence of blocks $\langle B_0, B_1, B_2, .., B_{d-1}\rangle$ form a critical path as the values of the cells in a block depend on the cells of the block preceding it. Hence, unless B_i's values are computed, they can't be used for B_{i+1}. Let's assume that each block is computed by a single processor and there is no re-computation. As there are d blocks on the critical path, its latency is d. Let the block dimensions be $k_0, k_1, ..., k_{d-1}$, respectively. Then $W = \sum_{i=0}^{d-1} \Omega(k_i^2)$ and $F = \sum_{i=0}^{d-1} \Omega\left(k_i^3\right)$. We also know that $\sum_{i=0}^{d-1} k_i = n$. Hence, to minimize bandwidth and computation cost, we make each $k_i = k$ for some k. Thus $d = n/k$ which gives us $F = \sum_{i=0}^{d-1} \Omega\left(k_i^3\right) = \Omega\left(nk^2\right)$.

If $F = \mathcal{O}\left(n^3/p\right)$, then combining with $F = \Omega\left(nk^2\right)$, we get $k = \mathcal{O}\left(n/\sqrt{p}\right)$. Latency, $S = d = \Omega\left(\sqrt{p}\right)$, and bandwidth, $W = \sum_{i=0}^{d-1} \Omega\left(k_i^2\right) = \Omega\left(n^2/\sqrt{p}\right)$.

For other problems such as the Gaussian elimination without pivoting and the gap problem, similar arguments hold.

4.3 Related Work (Distributed Memory)

Communication lower bounds have been established for several linear algebra algorithms, including QR and LU decomposition [10,11,16,25,28,57,59,65–67].

Classical $2D$ distributed-memory matrix multiplication (MM) algorithms use only one copy of the input/output matrix which is distributed across all p processors (by making a $\sqrt{p} \times \sqrt{p}$ processor grid [15]). They have $\Theta\left(n^2/\sqrt{p}\right)$ bandwidth cost and $\Theta\left(\sqrt{p}\right)$ latency cost, while they balance the load $\left(F = \Theta\left(n^3/p\right)\right)$ [8,10,36]. Our distributed-memory $\mathcal{R}$-$\mathfrak{DP}$ algorithms also use only one copy of the input, and the ones that access $n \times n$ matrices also distribute them evenly across processors arranged in a $\sqrt{p} \times \sqrt{p}$ processor grid. While our algorithms also balance load they are a $\log p$ factor away from the bandwidth and latency costs of the best 2D MM algorithm.

There is a class of distributed-memory MM algorithms, called $3D$, where $p^{\frac{1}{3}}$ copies of the input matrix are spread across processors which make a $3D$ processor grid of $p^{\frac{1}{3}} \times p^{\frac{1}{3}} \times p^{\frac{1}{3}}$ [5,6,24,39]. These algorithms also load balance $\left(F = \Theta\left(n^3/p\right)\right)$ as well as minimize the communication, with $\Theta\left(\log(p)\right)$ latency and $\Theta\left(n^2/p^{\frac{2}{3}}\right)$ bandwidth [10,36,60]. The third class of MM algorithms interpolate between the two classes where they take advantage of having c copies of the input matrix ($c \in \{1, 2, ..., p^{1/3}\}$) to build a $(p/c)^{1/2} \times (p/c)^{1/2} \times c$ processor grid, and hence they are called $2.5D$. These algorithms have $\Theta\left(n^2/\sqrt{cp}\right)$ bandwidth and $\mathcal{O}\left(\sqrt{p/c^3} + \log(c)\right)$ latency. The same technique can be used for Gaussian-elimination style LU algorithm to obtain bandwidth cost of $\Theta\left(n^2/\sqrt{cp}\right)$ and a latency cost of $\mathcal{O}\left(\sqrt{cp}\log p\right)$ which is asymptotically optimal for any choice of c (modulo $\log(p)$ factor for latency). Ballard et al. [9] have extended the communication cost analysis of distributed memory algorithms to fast MM algorithms

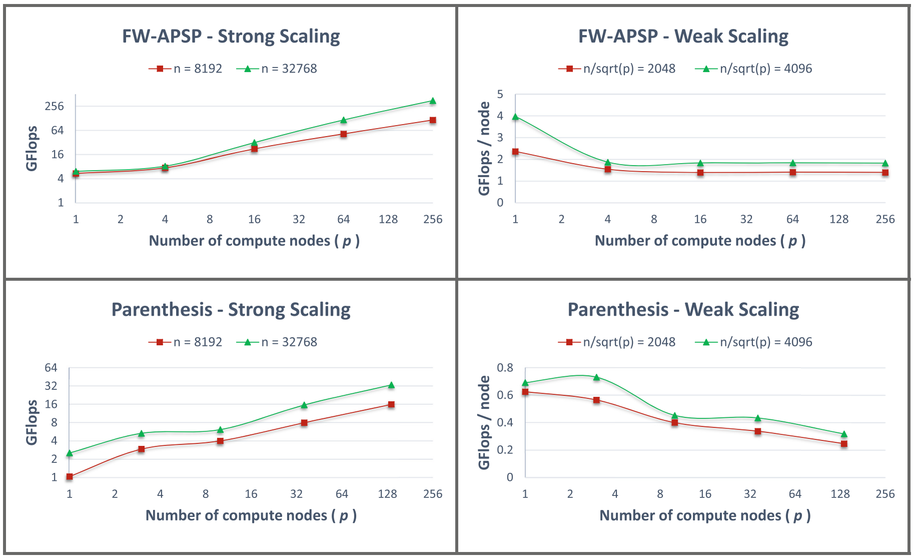

Fig. 8. Strong and weak scaling of our distributed-memory FW-APSP and parenthesis DP.

(Strassen's) and have proved that Communication-Avoiding Parallel Strassen (CAPS), running on a distributed-memory parallel machine meets the lower bounds on bandwidth and latency costs.

Distributed-memory graph algorithms [43,50] and DP algorithms also exist [33,35,38,42–44,63,68]. Solomonik et al. [58] presented a FW-APSP algorithm based on a block-cyclic approach which performs $\mathcal{O}\left(n^3/p\right)$ work and has $\mathcal{O}\left(n^2/\sqrt{p}\right)$ bandwidth and $\mathcal{O}\left(\sqrt{p}\log^2 p\right)$ latency. The 2.5D APSP algorithm given in [60] which builds on a recursive divide-and-conquer FW-APSP (Kleene) algorithm [7] has $\mathcal{O}\left(n^2/\sqrt{cp}\right)$ bandwidth and $\mathcal{O}\left(\sqrt{cp}\log^2 p\right)$ latency and performs $\mathcal{O}\left(n^3/p\right)$ work, where $c \in \{1, 2, ..., p^{1/3}\}$.

4.4 Distributed Memory Experimental Results

In this section, we present empirical results showing the performance benefits of our distributed memory algorithms that are based on r-way $\mathcal{R}$-$\mathcal{DP}$.

Setup. All experiments were performed on the SKX nodes of Stampede2 [3,70]. Each SKX node has dual-socket 24-core 2.1 GHz Intel Skylake processors ($2 \times 24 = 48$ cores in total) and 192 GB of DDR4 RAM. Each core is connected to a 32 KB L1 and a 1 MB L2 private caches. All 24 cores on a socket share one 33 MB L3 cache. Our Stampede2 allocation allowed us to use up to 128 SKX nodes simultaneously. We ran each MPI task on a separate socket, enabling us to run up to 256 MPI task for our experiments.

Implementations and Optimizations. All our algorithms (FW-APSP, parenthesis, gap) were implemented in C++. For distributed memory, we used intel MPI. Inside each process, we used Intel Cilk Plus extension to parallelize and Intel® C++ Compiler version 17.0.4 to compile the CPU implementations with optimization parameters -O3 -ipo -parallel -AVX -xhost. Additional intra-node CPU optimizations were the same as the ones explained in Sect. 3.5.

Distributed-Memory Results. Figure 8 shows the strong and weak scaling for FW-APSP and the parenthesis problem. Both algorithms show good scalability properties.

5 Conclusion

We have shown that 2-way recursive divide-and-conquer algorithms for a wide class of DP problems can be generalized so that they run with provable efficiency on shared-memory multicores and manycores (GPUs) as well as on distributed-memory machines without any changes in their basic structure. We have proved bounds on I/O and communication costs of these algorithms.

We believe that "Architecture-Independent Algorithms" holds promise for harnessing the full power of networks of hybrid compute nodes with both multicores and manycores because of their ability to run efficiently under multicore, manycore, shared-memory and distributed-memory settings. Many modern supercomputers already have such heterogeneous structures and exascale supercomputers in the near future are expected to look similar.

Acknowledgements. This work is supported in part by NSF grants CCF-1439084, CNS-1553510 and CCF-1725428. Part of this work used the Extreme Science and Engineering Discovery Environment (XSEDE) which is supported by NSF grant ACI-1053575. The authors would like to thank anonymous reviewers for valuable comments and suggestions that have significantly improved the paper.

References

1. Standard Template Library for Extra Large Data Sets (STXXL). http://stxxl.sourceforge.net/
2. The Stampede Supercomputing Cluster. https://www.tacc.utexas.edu/stampede/
3. The Stampede2 Supercomputing Cluster. https://www.tacc.utexas.edu/systems/stampede2/
4. Top 500 Supercomputers of the World. https://www.top500.org/lists/2018/06/
5. Agarwal, R.C., Balle, S.M., Gustavson, F.G., Joshi, M., Palkar, P.: A three-dimensional approach to parallel matrix multiplication. IBM J. Res. Dev. **39**(5), 575–582 (1995)
6. Aggarwal, A., Chandra, A.K., Snir, M.: Communication complexity of PRAMs. Theor. Comput. Sci. **71**(1), 3–28 (1990)
7. Aho, A.V., Hopcroft, J.E.: The Design and Analysis of Computer Algorithms. Pearson Education India, Noida (1974)
8. Ballard, G., Carson, E., Demmel, J., Hoemmen, M., Knight, N., Schwartz, O.: Communication lower bounds and optimal algorithms for numerical linear algebra. Acta Numer. **23**, 1–155 (2014)
9. Ballard, G., Demmel, J., Holtz, O., Lipshitz, B., Schwartz, O.: Communication-optimal parallel algorithm for strassen's matrix multiplication. In: Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 193–204. ACM (2012)
10. Ballard, G., Demmel, J., Holtz, O., Schwartz, O.: Minimizing communication in numerical linear algebra. SIAM J. Matrix Anal. Appl. **32**(3), 866–901 (2011)
11. Ballard, G., Demmel, J., Holtz, O., Schwartz, O.: Graph expansion and communication costs of fast matrix multiplication. J. ACM (JACM) **59**(6), 32 (2012)
12. Bellman, R.: Dynamic Programming. Princeton University Press, Princeton (1957)
13. Bender, M., Ebrahimi, R., Fineman, J., Ghasemiesfeh, G., Johnson, R., McCauley, S.: Cache-adaptive algorithms. In: SODA (2014)
14. Buluç, A., Gilbert, J.R., Budak, C.: Solving path problems on the GPU. Parallel Comput. **36**(5), 241–253 (2010)
15. Cannon, L.E.: A cellular computer to implement the Kalman filter algorithm. Technical report, Montana State University. Bozeman Engineering Research Labs (1969)
16. Carson, E., Knight, N., Demmel, J.: Avoiding communication in two-sided Krylov subspace methods. Technical report, EECS, UC Berkeley (2011)
17. Cherng, C., Ladner, R.: Cache efficient simple dynamic programming. In: AofA, pp. 49–58 (2005)
18. Chowdhury, R., Ganapathi, P., Tang, Y., Tithi, J.J.: Provably efficient scheduling of cache-oblivious wavefront algorithms. In: Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 339–350. ACM, July 2017

19. Chowdhury, R., et al.: AUTOGEN: automatic discovery of efficient recursive divide-&-conquer algorithms for solving dynamic programming problems. ACM Trans. Parallel Comput. **4**(1), 4 (2017). https://doi.org/10.1145/3125632
20. Chowdhury, R.A., Ramachandran, V.: Cache-efficient dynamic programming algorithms for multicores. In: SPAA, pp. 207–216 (2008)
21. Chowdhury, R.A., Ramachandran, V.: The cache-oblivious Gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation. Theory Comput. Syst. **47**(4), 878–919 (2010)
22. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. The MIT Press, Cambridge (2009)
23. D'Alberto, P., Nicolau, A.: R-Kleene: a high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks. Algorithmica **47**(2), 203–213 (2007)
24. Dekel, E., Nassimi, D., Sahni, S.: Parallel matrix and graph algorithms. SIAM J. Comput. **10**(4), 657–675 (1981)
25. Demmel, J., Grigori, L., Hoemmen, M., Langou, J.: Communication-optimal parallel and sequential QR and LU factorizations. SIAM J. Sci. Comput. **34**(1), A206–A239 (2012)
26. Diament, B., Ferencz, A.: Comparison of parallel APSP algorithms (1999)
27. Djidjev, H., Thulasidasan, S., Chapuis, G., Andonov, R., Lavenier, D.: Efficient multi-GPU computation of all-pairs shortest paths. In: IPDPS, pp. 360–369 (2014)
28. Driscoll, M., Georganas, E., Koanantakool, P., Solomonik, E., Yelick, K.: A communication-optimal n-body algorithm for direct interactions. In: IPDPS, pp. 1075–1084. IEEE (2013)
29. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: FOCS, pp. 285–297 (1999)
30. Galil, Z., Giancarlo, R.: Speeding up dynamic programming with applications to molecular biology. TCS **64**(1), 107–118 (1989)
31. Galil, Z., Park, K.: Parallel algorithms for dynamic programming recurrences with more than $O(1)$ dependency. JPDC **21**(2), 213–222 (1994)
32. Gusfield, D.: Algorithms on Strings, Trees and Sequences. Cambridge University Press, New York (1997)
33. Habbal, M.B., Koutsopoulos, H.N., Lerman, S.R.: A decomposition algorithm for the all-pairs shortest path problem on massively parallel computer architectures. Transp. Sci. **28**(4), 292–308 (1994)
34. Harish, P., Narayanan, P.: Accelerating large graph algorithms on the GPU using CUDA. In: HiPC, pp. 197–208 (2007)
35. Holzer, S., Wattenhofer, R.: Optimal distributed all pairs shortest paths and applications. In: PODC, pp. 355–364. ACM (2012)
36. Irony, D., Toledo, S., Tiskin, A.: Communication lower bounds for distributed-memory matrix multiplication. J. Parallel Distrib. Comput. **64**(9), 1017–1026 (2004)
37. Itzhaky, S., et al.: Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In: OOPSLA, pp. 145–164. ACM (2016)
38. Jenq, J.F., Sahni, S.: All pairs shortest paths on a hypercube multiprocessor (1987)
39. Johnsson, S.L.: Minimizing the communication time for matrix multiplication on multiprocessors. Parallel Comput. **19**(11), 1235–1257 (1993)
40. Katz, G.J., Kider Jr., J.T.: All-pairs shortest-paths for large graphs on the GPU. In: ACM SIGGRAPH/EUROGRAPHICS, pp. 47–55 (2008)
41. Kogge, P., Shalf, J.: Exascale computing trends: adjusting to the "new normal" for computer architecture. Comput. Sci. Eng. **15**(6), 16–26 (2013)

42. Krusche, P., Tiskin, A.: Efficient longest common subsequence computation using bulk-synchronous parallelism. In: Gavrilova, M.L., et al. (eds.) ICCSA 2006. LNCS, vol. 3984, pp. 165–174. Springer, Heidelberg (2006). https://doi.org/10.1007/11751649_18
43. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing: Design and Analysis of Algorithms, vol. 400. Benjamin/Cummings, Redwood City (1994)
44. Kumar, V., Singh, V.: Scalability of parallel algorithms for the all-pairs shortest-path problem. J. Parallel Distrib. Comput. **13**(2), 124–138 (1991)
45. Liu, W., Schmidt, B., Voss, G., Muller-Wittig, W.: Streaming algorithms for biological sequence alignment on GPUs. TPDS **18**(9), 1270–1281 (2007)
46. Liu, W., Schmidt, B., Voss, G., Schroder, A., Muller-Wittig, W.: Bio-sequence database scanning on a GPU. In: IPDPS, 8 pp. (2006)
47. Lund, B., Smith, J.W.: A multi-stage CUDA kernel for Floyd-Warshall. arXiv preprint arXiv:1001.4108 (2010)
48. Manavski, S.A., Valle, G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. BMC Bioinform. **9**(2), 1 (2008)
49. Matsumoto, K., Nakasato, N., Sedukhin, S.G.: Blocked all-pairs shortest paths algorithm for hybrid CPU-GPU system. In: HPCC, pp. 145–152 (2011)
50. Meyerhenke, H., Sanders, P., Schulz, C.: Parallel graph partitioning for complex networks. IEEE Trans. Parallel Distrib. Syst. **28**(9), 2625–2638 (2017)
51. Nishida, K., Ito, Y., Nakano, K.: Accelerating the dynamic programming for the matrix chain product on the GPU. In: ICNC, pp. 320–326 (2011)
52. Nishida, K., Nakano, K., Ito, Y.: Accelerating the dynamic programming for the optimal polygon triangulation on the GPU. In: Xiang, Y., Stojmenovic, I., Apduhan, B.O., Wang, G., Nakano, K., Zomaya, A. (eds.) ICA3PP 2012. LNCS, vol. 7439, pp. 1–15. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33078-0_1
53. Rizk, G., Lavenier, D.: GPU accelerated RNA folding algorithm. In: Allen, G., Nabrzyski, J., Seidel, E., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2009. LNCS, vol. 5544, pp. 1004–1013. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01970-8_101
54. Schulte, M.J., et al.: Achieving exascale capabilities through heterogeneous computing. IEEE Micro **35**(4), 26–36 (2015)
55. Sibeyn, J.F.: External matrix multiplication and all-pairs shortest path. IPL **91**(2), 99–106 (2004)
56. Solomon, S., Thulasiraman, P.: Performance study of mapping irregular computations on GPUs. In: IPDPS Workshops and PhD Forum, pp. 1–8 (2010)
57. Solomonik, E., Ballard, G., Demmel, J., Hoefler, T.: A communication-avoiding parallel algorithm for the symmetric eigenvalue problem. In: SPAA, pp. 111–121. ACM (2017)
58. Solomonik, E., Buluc, A., Demmel, J.: Minimizing communication in all-pairs shortest paths. In: IPDPS, pp. 548–559 (2013)
59. Solomonik, E., Carson, E., Knight, N., Demmel, J.: Trade-offs between synchronization, communication, and computation in parallel linear algebra computations. TOPC **3**(1), 3 (2016)
60. Solomonik, E., Demmel, J.: Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011. LNCS, vol. 6853, pp. 90–109. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23397-5_10

61. Steffen, P., Giegerich, R., Giraud, M.: GPU parallelization of algebraic dynamic programming. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009. LNCS, vol. 6068, pp. 290–299. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14403-5_31
62. Striemer, G.M., Akoglu, A.: Sequence alignment with GPU: performance and design challenges. In: IPDPS, pp. 1–10 (2009)
63. Tan, G., Sun, N., Gao, G.R.: A parallel dynamic programming algorithm on a multi-core architecture. In: SPAA, pp. 135–144. ACM (2007)
64. Tang, Y., You, R., Kan, H., Tithi, J., Ganapathi, P., Chowdhury, R.: Improving parallelism of recursive stencil computations without sacrificing cache performance. In: WOSC, pp. 1–7 (2014)
65. Tiskin, A.: Bulk-synchronous parallel Gaussian elimination. J. Math. Sci. **108**(6), 977–991 (2002)
66. Tiskin, A.: Communication-efficient parallel gaussian elimination. In: Malyshkin, V.E. (ed.) PaCT 2003. LNCS, vol. 2763, pp. 369–383. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45145-7_35
67. Tiskin, A.: Communication-efficient parallel generic pairwise elimination. Future Gener. Comput. Syst. **23**(2), 179–188 (2007)
68. Tiskin, A.: All-pairs shortest paths computation in the BSP model. In: Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) ICALP 2001. LNCS, vol. 2076, pp. 178–189. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-48224-5_15
69. Tithi, J.J., Ganapathi, P., Talati, A., Aggarwal, S., Chowdhury, R.: High-performance energy-efficient recursive dynamic programming with matrix-multiplication-like flexible kernels. In: IPDPS, pp. 303–312 (2015)
70. Towns, J., et al.: XSEDE: accelerating scientific discovery. Comput. Sci. Eng. **16**(5), 62–74 (2014)
71. Venkataraman, G., Sahni, S., Mukhopadhyaya, S.: A blocked all-pairs shortest-paths algorithm. JEA **8**, 2–2 (2003)
72. Volkov, V., Demmel, J.: LU, QR and Cholesky factorizations using vector capabilities of GPUs. EECS, UC Berkeley, Technical report UCB/EECS-2008-49, May 2008
73. Waterman, M.S.: Introduction to Computational Biology: Maps. Sequences and Genomes. Chapman & Hall Ltd., New York (1995)
74. Wu, C.C., Wei, K.C., Lin, T.H.: Optimizing dynamic programming on graphics processing units via data reuse and data prefetch with inter-block barrier synchronization. In: ICPADS, pp. 45–52 (2012)
75. Xiao, S., Aji, A.M., Feng, W.c.: On the robust mapping of dynamic programming onto a graphics processing unit. In: ICPADS, pp. 26–33 (2009)