

IMPLEMENTATION OF CACHE-EFFICIENT PARALLEL DYNAMIC PROGRAMMING ALGORITHMS ON GPU_s WITH EXTERNAL MEMORY

Arun Ramachandran, Swaminathan Sivaraman

May 15 2017

1 PROBLEM DESCRIPTION AND MOTIVATION

Dynamic programming solves a complex problem by recursively dividing it into subproblems, solving the subproblems, storing their solutions since they are shared across subproblems, and combining the solutions to give the best solution for the problem. The parallel implementation of recursive dynamic programming (R-DP) problems involves the consideration of memory components such as the cache/memory of individual components of storage and computation such as the CPU, the GPU, the RAM and the external storage(HDD or SSD). The cache misses/hits, and evictions also become important. Thus, parallel implementations must seek to achieve optimal caching complexity(measured by the number of block transfers across all caches) and achieve maximum speedup.

This project seeks to implement cache-efficient solutions to R-DP problems, on multicore machines with a deep memory hierarchy, in particular, on GPUs. GPUs are hardware accelerators that are attached to the CPU through a PCI bus. They are intended primarily for graphics processing, but could be exploited for general purpose computing as well(using APIs like OpenCL, NVIDIA's CUDA, Microsoft DirectCompute etc). The GPU has a global memory and many shared memories, each connected to a constant number of cores with their own registers. There are hundreds of cores, can be used to run highly parallel and data-intensive computations, and can help to achieve maximum throughput of parallel programs. The challenge with designing parallel algorithms for R-DP problems using GPUs is that GPUs don't have automatic page replacement policies, and there are overlapping computation of kernels and data transfers(called streams and events). So, other techniques are

needed to solve R-DP problems using GPUs. The reason for choosing to implement DP problems is that they can be recursively tiled, have excellent cache locality and can be exploited to achieve a very good parallel running time on multicore machines with a deep memory hierarchy.

We take the problem of solving a R-DP problem using GPU, such that the data is stored in external storage(a hard disk drive(HDD)) and is too large to fit in the memory of the GPU all at once. There are multiple layers of transfer, firstly from the HDD to the RAM using Standard Template Library for Extra Large Data Sets(STXXL) (with attendant LRU page replacement policies), and next from the RAM to the GPU in explicit block sizes. If the data present in external storage is too large, such as in the case of a Floyd Warshall's All Pairs Shortest Path(APSP) problem, the cost of transfer is high. We want to show that speedup of cache-efficient parallel implementation of such R-DP problems using GPUs is faster than that by using CPUs.

2 PRIOR WORK

An existing design framework, named AUTOGEN FRACTILE, designs I/O efficient parallel algorithms for R-DP problems with good speedup. The algorithms can run on GPUs, and can run faster than on existing tiled-iterative GPU implementations with shared memory, and implementations with in-memory CPU.

AUTOGEN FRACTILE uses r-way recursive divide-and-conquer to design tiled algorithms, when given the tiled iterative DP algorithm as an input. A d-dimensional hypercubic table of size n^d , is broken into r^d orthants(tables) of size $(n/r)^d$ each. Considering a Floyd Warshall's APSP problem of $n \times n(n^2)$, it is broken into $4(2^2)$ tables of size $(n/2)^2$ each. Then, depending upon the dependencies between the subtables(submatrices), the recursive definitions are made. The parameter r can be set to a constant value, can be a function of n, or can be set to the largest value of the size of the tile at level d which can exactly fit into the memory/cache at that level. Typically, when there are h levels of memory/cache, then there are h recursion levels and different tile-sizes for each level. When a subproblem fits into a memory of the smallest size, then the iterative component starts getting executed. The advantage of this r-way recursive divide-and-conquer technique is that it provides a definitive framework, the parameter r can be modified in each recursion level(based on the memory size into which the table can fit at that level), and this works for a GPU/multiple GPUs.

Cache-efficient parallel algorithms for problems such as the paranthesization problem, Floyd-Warshall APSP problem, the Gaussian Elimination without pivot, and the sequence alignment problem have been designed and implemented on multicore machines. In particular, they have also been

implemented using GPUs. The external-memory GPU implementations used 2-way R-DP in external memory until the problem fit in the GPU’s global memory, after which it used the r-way R-DP technique. The speedup had been calculated and compared by plotting against the speedup obtained with a 2-way serial R-DP running on CPU, and the speedup on a 2-way parallel R-DP running on a CPU with 128 cores.

3 GOALS / DELIVARABLES STATED IN THE PROPOSAL

- The first task is to use the AUTOGEN FRACTILE framework to implement the cache-efficient parallel algorithms for the Floyd Warshall’s APSP problem and the paranthesization problem, on GPUs, with the data residing on external storage(HDD).
 - The implementation will be an r-way R-DP approach all the way from the external storage to the GPU. The speedup obtained using an r-way R-DP on a GPU is expected to be similar to the reported values in the work done above.
 - This speedup will be plotted for $n = 2^{10}$ to 2^{15} , and compared against the corresponding speedup values obtained with a parallel R-DP running on a CPU with 16 cores (Stampede system). The CPU implementation will be R-DP from the external storage to the CPU, after which it will be 2-way.
- Next, the task will be extended to 2 more problems :
 - Longest Common Subsequence (LCS)
 - Matrix Multiplication

The results will be obtained for the above 2 problems, and tabulated, for $n = 2^{10}$ to 2^{15} . This is also expected to have more speedup on a GPU, as compared to a CPU(single or multicore).

3.1 UPDATED GOALS

We faced the following problems over the course of implementation, which led us to update the goals:

1. Getting the environment setup, for solving problems with data in external storage, using STXXL. Enabling STXXL to work with parallel cilk-code was a tough task.
 - We had trouble with installing Cilk in our MAC laptops, so we tried installing in a Virtual Box but that did not support Cuda (as there was no access to the native hardware’s GPU drivers). We finally

installed ICPC in MAC, but still could not make use of Cuda as our MAC laptops had only INTEL graphics cards. So, we have decided to directly make use of Stampede, to debug and test GPU code using Cuda.

- We were able to install STXXL, but unable to integrate STXXL library with our parallel implementation of Floyd Warshall’s APSP in Cilk. The problem we faced was that the compilation and building of the STXXL library did not recognize the “cilk” APIs. The external memory implementation using GPUs does not need this, as we will not be using “cilk” APIs with STXXL at the CPU-level in that case, but will be using only Cuda-APIs at the GPU level. But for the external memory implementation using CPUs, we need the integration of STXXL with Cilk.
2. Beyond that, obtaining complete parallelism was difficult due to the cache replacement problems and a non-provision of STXXL to take care of them.
 3. We tried implementing the parenthesization problem, but could not make the base case work for one of the functions of the AUTOGEN-generated algorithm. More details regarding this are given in Section 7.

Based on the problems we faced (which took time for us to explore and solve), and given the time constraints that we had, we had reduced our goals/deliverables as follows:

- Implement the parallel r-way R-DP solutions for the Floyd-Warshall’s APSP problem, on a CPU and a GPU with data in the external memory.
- Implement the parallel r-way recursive matrix multiplication algorithm, on a CPU and GPU with data in the external memory.

4 OVERVIEW OF WORK

We implemented the cache-efficient R-DP problems of Floyd-Warshall’s APSP, and matrix multiplication, with data residing on external storage. For the CPU implementation, we used an r-way approach for the transfer from the external storage to the CPU, after which we used a 2-way approach. For the GPU implementation, we used an r-way approach from the external storage to CPU RAM to GPU global and shared memories. We compared the performance (in terms of run time) on a CPU vs GPU, and found out that the GPU was significantly faster (30 – 40 times faster) than the CPU runtime for Floyd-Warshall’s APSP and around 4-5 times faster for matrix multiplication (which is still significant speedup).

Over the course of implementation, we did the following tricks and optimizations:

- Setting up of the RAM size, GPU global and shared memories as configurable parameters, to mix and match and test accordingly
- Preprocessing of the input data and storing them in the Z-morton layout for optimal cache-locality
- During the CPU implementation using STXXL, avoidance of unnecessary storage back to the disk and subsequent re-copy into RAM without any modification
- Enablement of maximal storage from external storage to the CPU RAM using STXXL. This accounts for both the input and output objects(matrices)
- Use of pinned CUDA memory, for faster DMA transfers
- Sequential copy from the GPU global to shared memory, to avoid a random copy during memory coalescing
- Execution of kernels with multiple threads in multiple blocks, and achieving maximum parallelism possible.

5 DETAILED EXPLANATION OF WORK

5.1 GENERIC FLOW FOR ALL R-DP PROBLEMS USING GPUs

- Data in External Storage(host-DISK)
 - Split full problem into $r_1 \times r_1$ sub-problems of size $\frac{n}{r_1} \times \frac{n}{r_1}$: $(\frac{n}{r_1})$ fits into the RAM. Copy each sub-problem to RAM and call relevant RAM functions.
- Data in CPU RAM (host-RAM)
 - Split problem into $r_2 \times r_2$ sub-problems of size $\frac{n}{r_1 * r_2} \times \frac{n}{r_1 * r_2}$: $(\frac{n}{r_1 * r_2})$ fits into the global GPU memory. Copy each sub-problem to GPU global memory and call relevant GPU kernel launcher functions. (For CPU implementations, instead of going to GPU, the RAM functions themselves will do parallel 2-way divide-and-conquer until the problem size becomes small enough to run iterative base cases).
- Data in GPU global memory (host-GPU)
 - Split sub-problem into $r_3 \times r_3$ problems of size $\frac{n}{r_1 * r_2 * r_3} \times \frac{n}{r_1 * r_2 * r_3}$: $(\frac{n}{r_1 * r_2 * r_3})$ fits into the GPU shared memory. Note that unlike previous cases, no real "splitting" is done here. The function simply

decides on a value of r_3 such that a sub-problem can fit in GPU shared memory. It then launches the relevant kernels, passing in this value of r_3 .

- Data in GPU shared memory (device-GPU) - GPU kernels
 - Each sub-problem (a matrix) is assigned to a GPU block. Each block copies the data (one write portion and 0 or more read portions) required to solve that sub-problem from global memory into its shared memory, solves the sub-problem and copies the computed data back to global memory. Note that though it is easier to think of this as work being done in parallel at a block-level, in reality, work is being done in parallel at a thread-level, where each thread corresponds to a cell in the matrix. CUDA's thread-syncing feature at a block-level is used to ensure algorithm correctness.

5.2 FLOYD-WARSHALL CPU CODE FLOW

- Input is in Z-morton format, with the data residing in external storage (HDD)
- There are 2 variants of the recursive DP algorithm, 2-way and r-way
- Since the data can't fit into the RAM all at once, we use STXXL to facilitate transfer back and forth between the external storage to the CPU RAM.
 - But difficulties lie in coordinating parallelism with cache replacement, when using STXXL
 - The optimal way is to copy maximal portions from external storage into RAM, parallelize, and copy back to external storage.
- There are 4 major functions(A-*, B-*, C-*, D-*) in each memory level in the hierarchy. Each of the functions operate on submatrices X, U and V such that:
 - $X = U = V$ in A-*
 - $X = V \neq U$ in B-*
 - $X = U \neq V$ in C-*
 - $X \neq U \neq V$ in D-*
- **FLOW**
 - A-host-DISK splits the input in a serial r-way manner and copies maximal portions from the external storage to the RAM

- A-host-RAM, B-host-RAM, C-host-RAM, D-host-RAM perform computations using a 2-way parallel technique with Cilk
- The computed portions are copied back from the RAM to the external storage

5.3 FLOYD-WARSHALL GPU CODE FLOW

FLOW

- A-host-DISK splits the input in a serial r-way manner and copies maximal portions from the external storage to the RAM (CUDA pinned host memory is used as RAM memory for faster GPU transfers later on). It then calls A-host-RAM, B-host-RAM, C-host-RAM and D-host-RAM as appropriate.
- A-host-RAM, B-host-RAM, C-host-RAM, D-host-RAM split in a parallel r-way manner and copies data from the RAM to GPU global memory. Then, the A-host-GPU, B-host-GPU, C-host-GPU and D-host-GPU functions are called as needed.
- A-host-GPU, B-host-GPU, C-host-GPU and D-host-GPU are the kernel launcher functions. They split in a parallel r-way manner such that each sub-problem fits into GPU shared memory and launch the kernels with maximum parallelism possible for each kernel launch and specify the value of 'r' to the kernels. The kernels are launched in the order dictated by the algorithm.
- Separate kernels have been written for each launcher function, for clarity,
 - For A-host-GPU, there are 3 kernels - A-device-GPU-from-A, B-C-device-GPU-from-A and D-device-GPU-from-A.
 - For B-host-GPU, there are 2 kernels - B-device-GPU-from-B and D-device-GPU-from-B.
 - For C-host-GPU, there are 2 kernels - C-device-GPU-from-C and D-device-GPU-from-C.
 - For D-host-GPU, there is 1 kernel - D-device-GPU-from-D.
- The kernels are launched such that each sub-problem or sub-matrix is assigned to a block. Each cell within that matrix is assigned to a thread in that block. When the kernel is launched, each thread figures out its sub-matrix location based on its block id. Then each thread copies one unique data point from that sub-matrix located in global memory to the block-level shared memory. Care is taken to ensure that warps of threads access global memory sequentially. Then the parallel base case is run. Since the threads are already running in parallel, the actual computation is done directly. CUDA's thread-syncing feature at a block-level is used to ensure data correctness. Finally, only the solved sub-problem is copied back to global memory, using the same copying mechanism like before.

- Some points to note
 - CUDA only allows a maximum of 1024 threads per block, so each sub-matrix size cannot be greater than 32×32 ($2^5 \times 2^5$). However, since this would max out GPU resources, it is best to use only 256 threads per block, or 16×16 ($2^4 \times 2^4$). We tested with both configurations and observed slightly better results for the second case.
 - However, the maximum blocks allowed (in two dimensions) are 65536×65536 , so we can schedule lots of blocks in the kernel. Of course, memory constraints don't allow us to launch these many blocks, so we are not limited by CUDA in any way for this case.

5.4 PARALLELISM FOR FLOYD-WARSHALL

Essentially, these are the unique kernels with regards to parallelism,

- A-function parallelizes 1 block
- B-C-function parallelizes $\sim r$ blocks each for B and C (total $\sim 2r$)
- B-function alone parallelizes $\sim r$ blocks
- C-function alone parallelizes $\sim r$ blocks
- D-function parallelizes $\sim r^2$ blocks

Assuming 256 threads per block, the actual parallelism is $256 \times$ the block parallelism value, for each kernel launch.

5.5 MATRIX MULTIPLICATION CODE FLOW

The r-way parallel divide-and-conquer matrix multiplication algorithm really has only one function. So, it is easier to understand than the Floyd-Warshall's APSP algorithm. So, there is only one "A" function at all levels - disk, RAM, GPU launcher and GPU kernel.

The implementation is same as that for the Floyd-Warshall's algorithm. Data is initially in HDD. It is split r-way and loaded into RAM. The CPU code then does parallel 2-way divide-and-conquer while the GPU code splits r-way at at the GPU global and GPU shared memory levels. The GPU implementation details are the same as for Floyd-Warshall's algorithm. The only difference is the base case operation and the fact that there is only one function here at all levels, upto the kernel. Each kernel launch will have r^2 blocks when r-way splitting is used, and each block will have 256 threads as before.

5.6 PARALLELISM FOR MATRIX MULTIPLICATION

Each kernel launch parallelizes r^2 blocks. Assuming 256 threads per block, the actual parallelism is $256 \times r^2$.

r

A0	B1	B1	B1
C1	D2	D2	D2
C1	D2	D2	D2
C1	D2	D2	D2

r

GRID SHOWING PARALLELIZATION OF RECURSIVE FUNCTIONS
FOR FLOYD WARSHALL'S APSP FOR $R = 4$

6 EXPERIMENTAL RESULTS

- All experiments were run on Stampede.
- The CPU tests were run on a node that had two Xeon E5-2680 8-core Sandy Bridge processors, a total of 16 cores.
- The GPU tests were run on a NVIDIA Tesla K20m GPU, whose warp size is 32, and number of multiprocessors is 13.
- The input and output were in Z-morton format, input size varying from 2^{10} to 2^{15} .
- STXXL was used for large memory computations, when the input data could not fit all at once in the RAM. To simulate this, the RAM size was configured to be ~ 100 MB.
 - PageSize (number of blocks per page) was set to 1
 - CachePages (number of pages) was set to 1
 - Block Size was set to ~ 100 MB
 - LRU was the chosen cache-replacement policy

Since the STXXL vector had to account for input and output objects(matrices), which were 3 in number in this case, a single matrix of size 32 MB could fit into the RAM at a point of time. Hence, an input matrix of unsigned long datatype of size $2^{11} \times 2^{11}$ could fit into the RAM at a point. Any size beyond it had to be divided accordingly.

- As mentioned before, for the GPU kernels, each block was assigned 256 threads to maximize performance. This also allowed each sub-problem to fit in GPU shared memory.
- Since we were using artificial memory limits, and RAM size was set to ~ 100 MB, we set GPU global memory to be around 7 MB. This would mean that the matrix that fits into GPU global memory would be of size 2^9 .
- So, according to the memory limits, following are the sizes of the sub-problems at each memory level,

- RAM - $2^{11} \times 2^{11}$
- GPU Global Memory - $2^9 \times 2^9$
- GPU Shared Memory - $2^4 \times 2^4$

As seen, when using GPU's the r value when launching the kernel is $2^9 / 2^4 = 2^5$. So, for highly parallel kernels where the parallelism is $256 * r^2$ (where 256 is the threads per block), the parallelism would be $256 * (2^5 * 2^5) = 262144 = 2^{18}$.

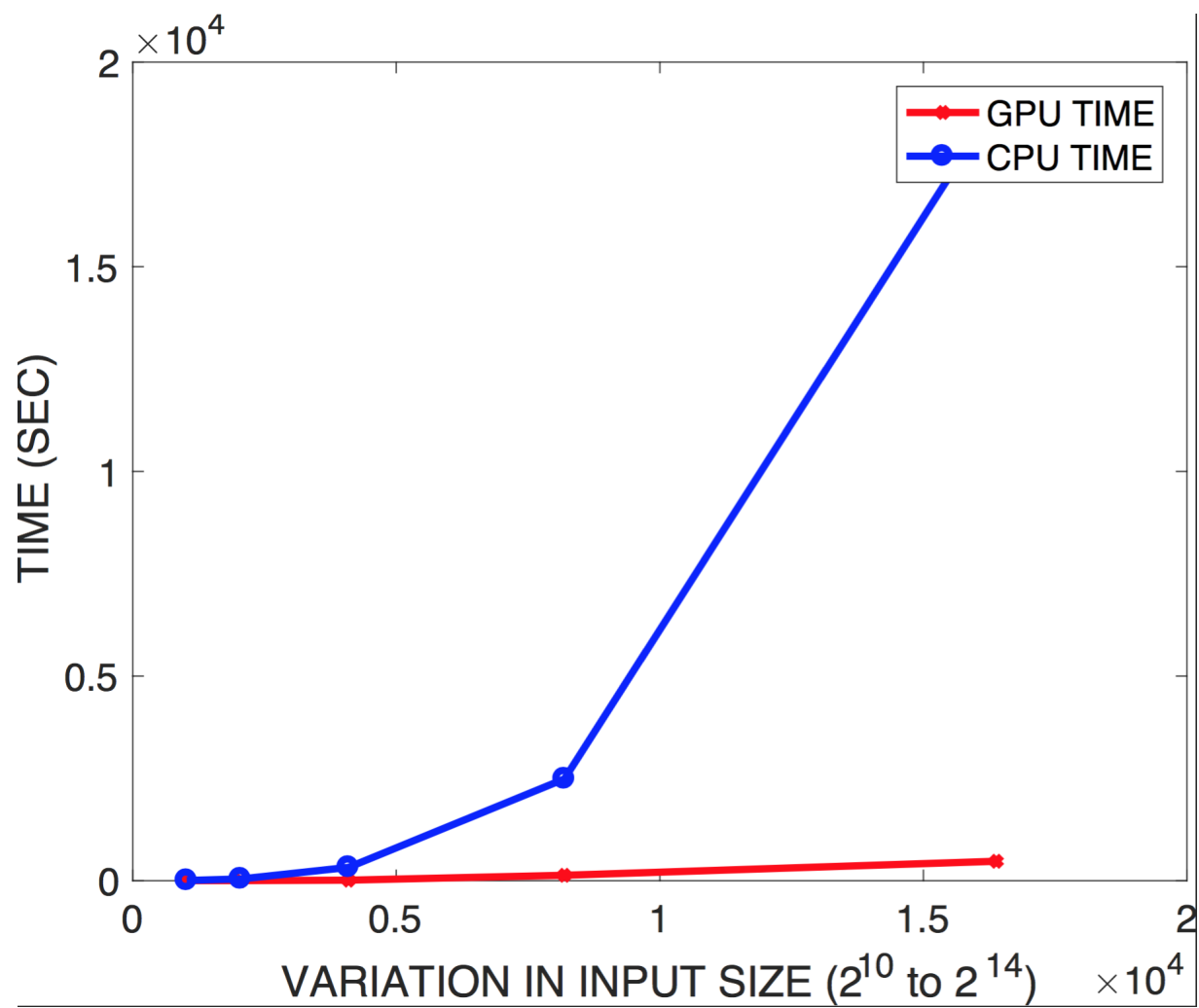
- All compiler optimizations were enabled, while using the standard C++11 compiler.

6.1 FLOYD-WARSHALL's APSP

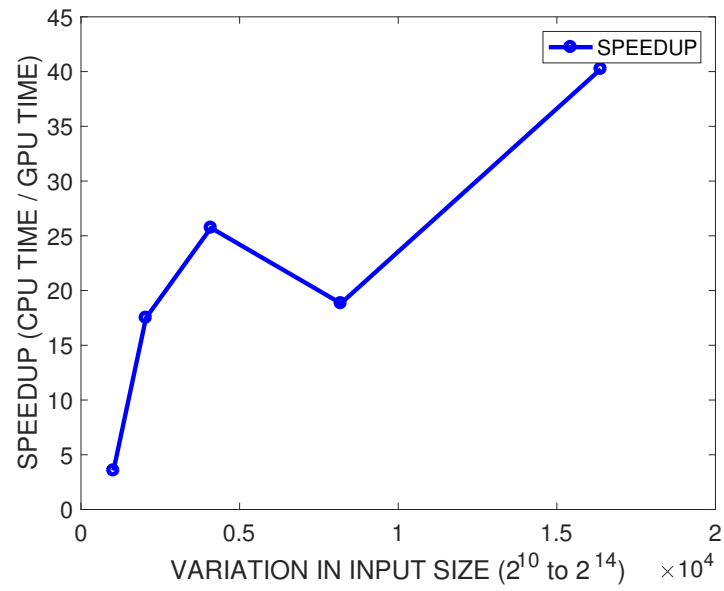
GPU vs CPU TIME FOR INPUT FROM 2^{10} TO 2^{14}

INPUT	CPU TIME (secs)	GPU TIME (secs)
2^{10}	5.9	1.66
2^{11}	42	2.4
2^{12}	325.7	12.66
2^{13}	2487	132.1
2^{14}	19339.3	474.8
2^{15}		3800.29

GPU VS CPU TIME FOR FLOYD WARSHALL'S APSP FOR INPUT SIZE
 2^{10} TO 2^{14}



GPU VS CPU PLOT FOR FLOYD WARSHALL'S APSP FOR INPUT SIZE
 2^{10} TO 2^{14}



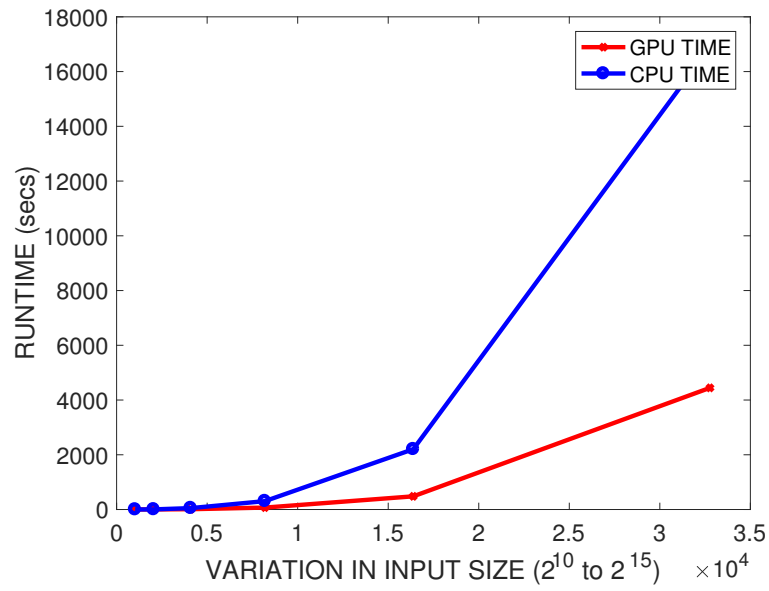
SPEEDUP PLOT (GPU/CPU TIMES) FOR FLOYD WARSHALL'S APSP
FOR INPUT SIZE 2^{10} TO 2^{14}

6.2 MATRIX MULTIPLICATION

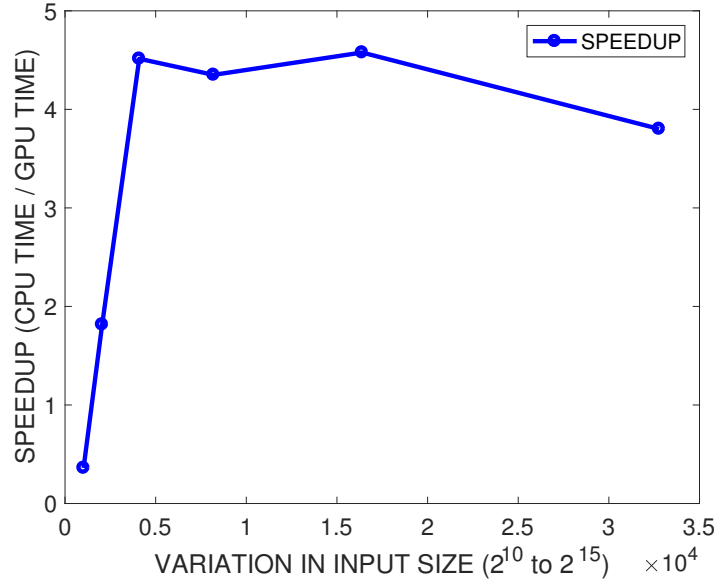
GPU vs CPU TIME FOR INPUT FROM 2^{10} TO 2^{15}

INPUT	CPU TIME (secs)	GPU TIME (secs)
2^{10}	0.65	1.8
2^{11}	4.74	2.61
2^{12}	51.23	11.35
2^{13}	306.29	70.41
2^{14}	2203.61	481.45
2^{15}	16896.1	4444.49

GPU VS CPU TIME FOR MATRIX MULTIPLICATION FOR INPUT SIZE
 2^{10} TO 2^{15}



GPU VS CPU PLOT FOR MATRIX MULTIPLICATION FOR INPUT SIZE
 2^{10} TO 2^{15}



SPEEDUP PLOT (GPU/CPU TIMES) FOR MATRIX MULTIPLICATION
FOR INPUT SIZE 2^{10} TO 2^{15}

6.3 INCLUDED SOURCE CODE AND DATA

We have included the complete CPU and GPU implementations for the Floyd-Warshall's APSP problem and matrix multiplication. We have also included the scripts we used to generate the data and convert the matrices to Z-morton layout. We have included data files of size $2^{12} \times 2^{12}$, since they are the smallest data file size that will exercise all portions of the code, from disk-splitting to RAM-splitting. We haven't included the larger sizes due to memory constraints, but can provide them on demand. Alternatively, the provided scripts can be used to generate larger datasets. The uploaded data files can be found at <https://drive.google.com/drive/folders/0B8ggcM2weaMqSUV0R2x5ajh5ZDQ?usp=sharing>.

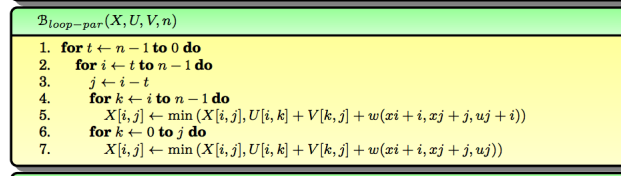
We have also included the partial implementation of the parenthesization problem. We have entailed the problems faced in its implementation below. If we have more time, we will look to fix and implement it completely.

7 POTENTIAL EXTENSIONS

7.1 PARENTHEZIZATION PROBLEM

We tried implementing the parenthesization problem on GPU and CPU as well. But we faced an issue where the B-function base case was not generating correct

results, even when using serial code. We investigated the code flow, and found that the cells in the X matrix never get updated with non-infinity values. We used the base case given in the Fig 7.1 Even without implementations, one can try hand-checking this for matrices of size $n = 2$.



```

 $\mathcal{B}_{loop-par}(X, U, V, n)$ 
1. for  $t \leftarrow n - 1$  to 0 do
2.   for  $i \leftarrow t$  to  $n - 1$  do
3.      $j \leftarrow i - t$ 
4.     for  $k \leftarrow i$  to  $n - 1$  do
5.        $X[i, j] \leftarrow \min(X[i, j], U[i, k] + V[k, j] + w(xi + i, xj + j, uj + i))$ 
6.       for  $k \leftarrow 0$  to  $j$  do
7.          $X[i, j] \leftarrow \min(X[i, j], U[i, k] + V[k, j] + w(xi + i, xj + j, uj))$ 

```

Figure 7.1: B-Loop base case code for the parenthesization problem.

As future work, we would like to investigate this further. Perhaps the given code has a typo somewhere. We would like to get a working B base case and implement the CPU and GPU code around it.

7.2 USE OF STREAMS IN THE GPU

Currently, the work done is completely sequential until kernel launches take place. We would like to use the GPU streams functionality, which allow multiple parallel kernel launches. This will allow us to parallelize the outer CPU launcher code as well, and launch parallel kernels in different streams. This would also play well with GPU devices which natively allow data copy and computation overlap.

7.3 IMPLEMENTATIONS FOR MORE PROBLEMS

If we had more time, we would also extend the work to implement other parallel r-way divide-and-conquer algorithms.

8 REFERENCES

- <http://pramodganapathi.com/pdf/Pramod%20Ganapathi%20-%20Dissertation.pdf>
- <http://www.cs.utexas.edu/~vlr/papers/spaa08.pdf>
- http://stxxl.org/tags/1.4.1/install_unix.html
- <https://www.tacc.utexas.edu/documents/13601/69f5e3f1-d146-466b-ae99-eaa479123cc9>
- <http://developer.download.nvidia.com/books/cuda-by-example/cuda-by-example-sample.pdf>
- <http://www3.cs.stonybrook.edu/~rezaul/Spring-2013/CSE638/CSE638-lectures-16-17.pdf>

- https://cvw.cac.cornell.edu/gpu/memory_arch
- <https://devblogs.nvidia.com/parallelforall/separate-compilation-linking-cuda-device-code/>
- <https://fgiesen.wordpress.com/2009/12/13/decoding-morton-codes/>