Performance Study of Mapping Irregular Computations on GPUs

Steven Solomon and Parimala Thulasiraman
University of Manitoba

Presented by: Ruppa Thulasiram

What Will Be Covered?

(1) Introduction and Motivation

(2) Algorithms Considered

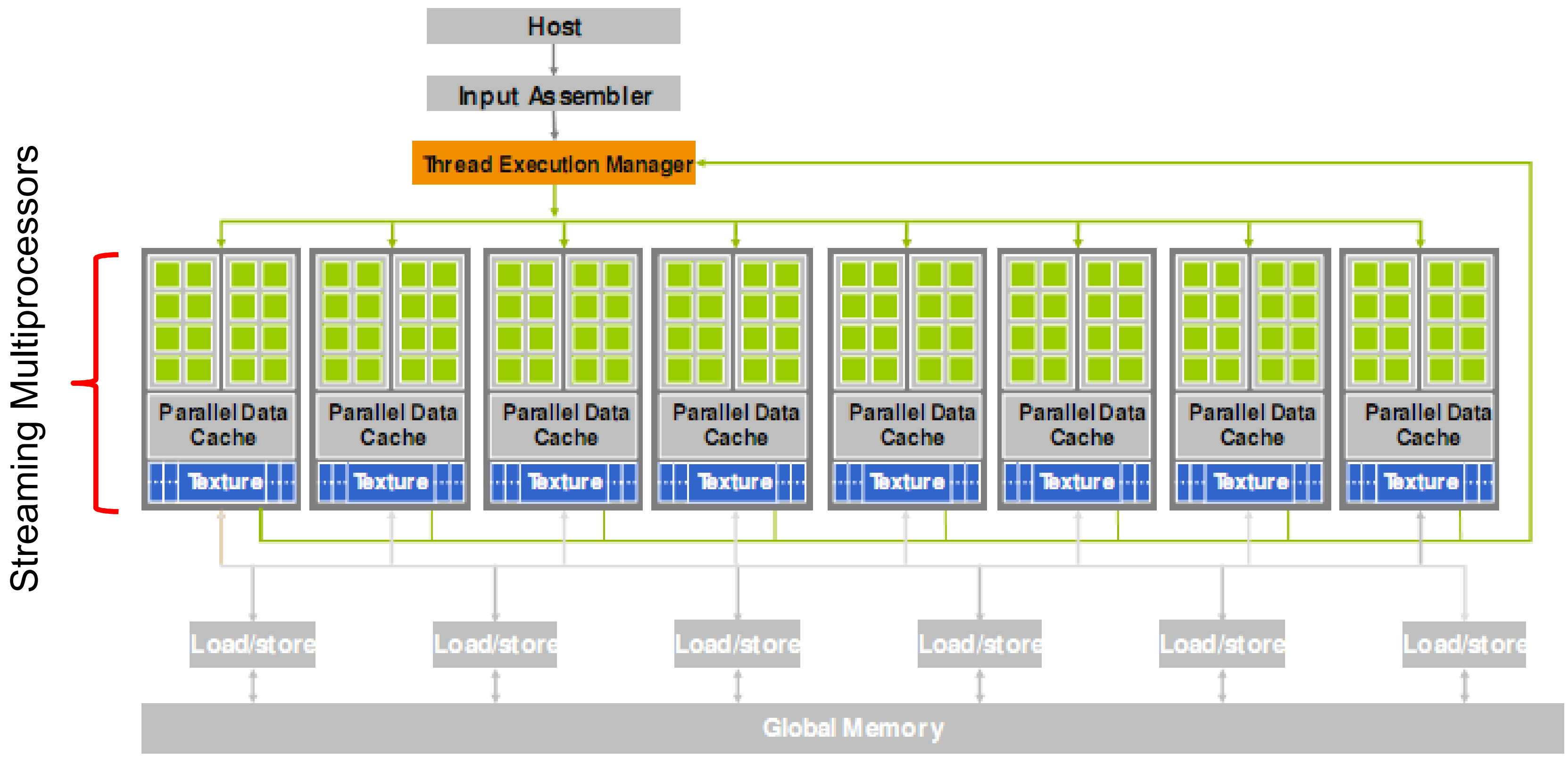
(3) Implementation of Matrix Parenthesization

(4) Implementation of Breadth First Search

(5) Results

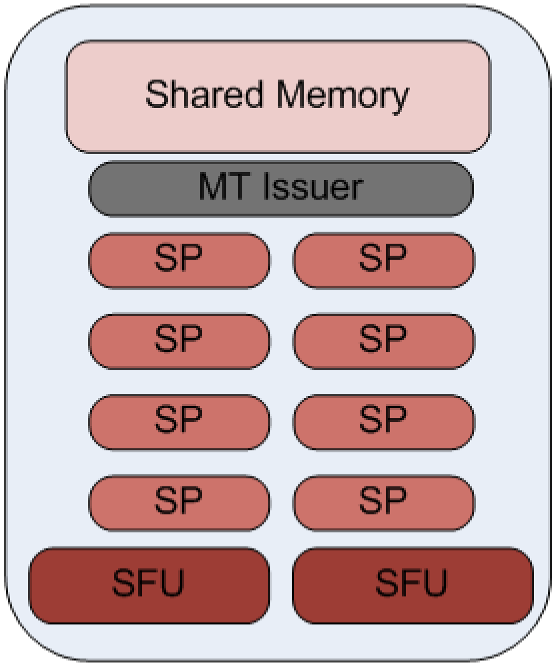
(6) Conclusions and Future Work

GPU Architecture and CUDA



GPU Architecture and CUDA

- Streaming Multiprocessor – the computational cores of the GPU.
- Composed of 8 Scalar Processors (SPs) and 16KB of (fast) Shared Memory.
- Multi-threaded instruction issue unit and 2 Special Function Units.



GPU Architecture and CUDA

- Threads are grouped into warps (32 per warp)
- Warps are grouped into blocks, and blocks into a grid.
- Each block executes on only *one* SM, but multiple blocks can execute on a single SM.

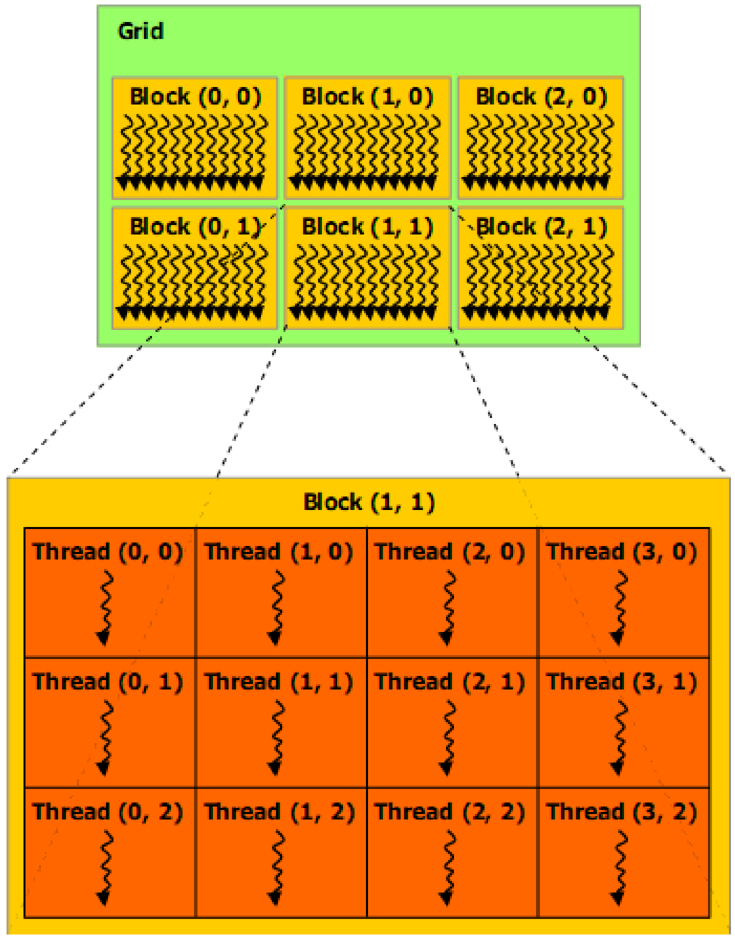
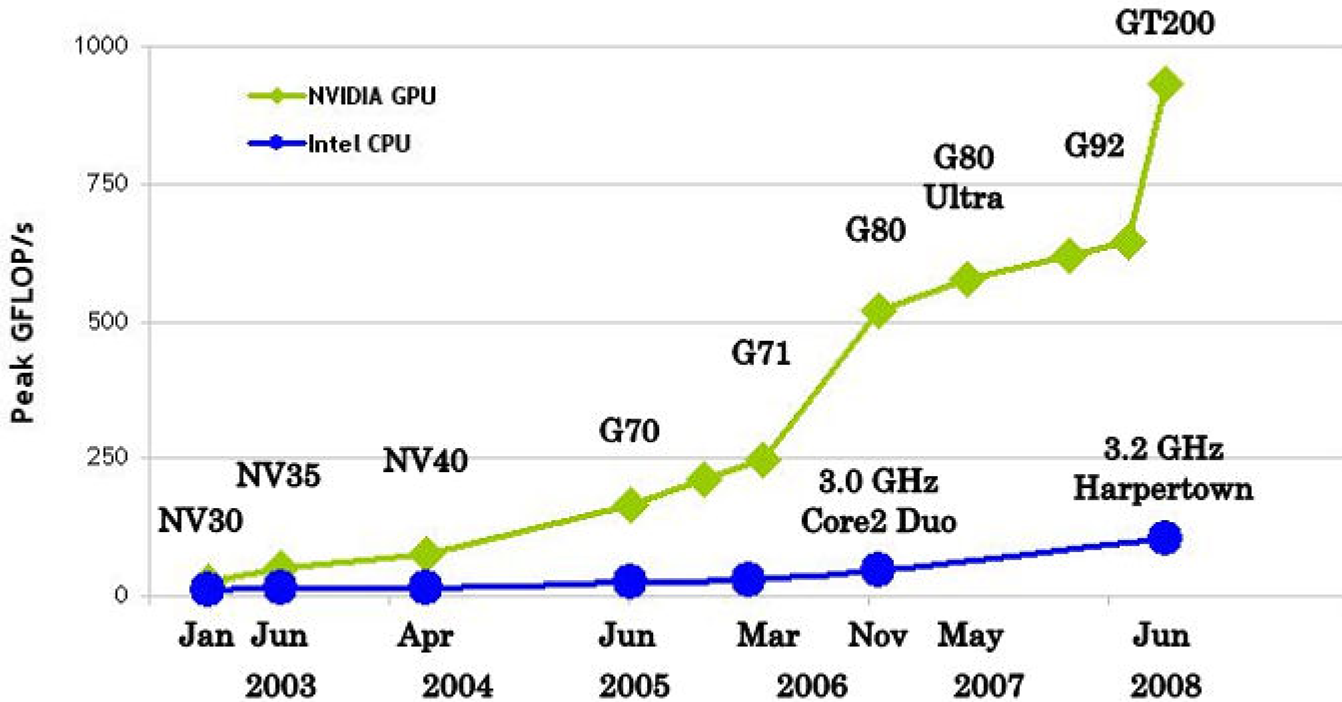


Image Source: NVIDIA CUDA Programming Guide 2.3

Why GPUs?



- A single GPU has a significant amount more potential compute power than a single CPU.
- Adding more CPUs to reach the level of a GPU is expensive.

Image Source: NVIDIA CUDA Programming Guide 2.3

Why GPUs?

- Significant computational power:
  - 1 teraFLOPS of performance on high end Nvidia GT200 GPU.
- Massive parallelism:
  - Thousands of threads in flight.
- Memory bottleneck still exists:
  - Hundreds of cycles to access data in global memory.

Why Irregular Algorithms?

- Unpredictable and unstructured data access patterns, more difficult to parallelize efficiently than regular algorithms.

- Study the effects of memory issues on the GPU – can the computational power and parallelism available outweigh the memory bottleneck?

- Less existing work on irregular algorithms compared to regular algorithms on the GPU.

What Will Be Covered?

(1) Introduction and Motivation

(2) Algorithms Considered

(3) Implementation of Matrix Parenthesization

(4) Implementation of Breadth First Search

(5) Results

(6) Conclusions and Future Work

Matrix Parenthesization

Given a series of matrices:

A * B * C

Determine an order of multiplication such that the number of scalar multiplications are minimized.

Matrix Parenthesization

- Consider two options in this example:

  (A * B) * C   A * (B * C)

- Assume dimensions are:

  A – 100 x 1   B – 1 x 1   C – 1 x 1

(A * B) * C	A * (B * C)
(A * B) = (100 * 1 * 1) = 100 ops AB * C = (100 * 1 * 1) = 100 ops **200 operations total**	(B * C) = (1 * 1 * 1) = 1 op A * BC = (100 * 1 * 1) = 100 ops **101 operations total**

Matrix Parenthesization

- Bottom-up, dynamic programming approach.
  - Optimal solution for each chain dependent on structure of input data.

- Smallest sub-problems are solved first (matrix chain of length 1).

- *Reuse* the previously computed sub-problem solutions for longer chains.

Matrix Parenthesization

Phase 0

Opt(A)		
	Opt(B)	
		Opt(C)

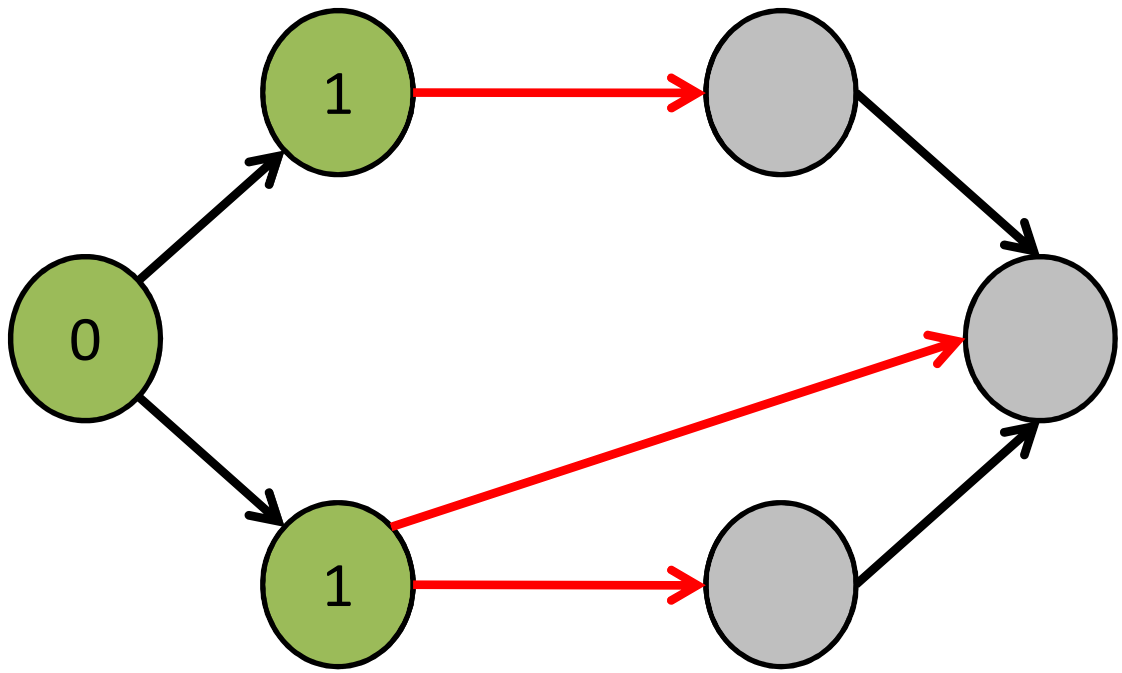
Matrix Parenthesization

Phase 0	Opt(A)		
Phase 1	**Opt(AB)**	Opt(B)	
		Opt(BC)	Opt(C)

Matrix Parenthesization

Phase 0	Opt(A)		
Phase 1	Opt(AB)	Opt(B)	
Phase 2	**Opt(ABC)**	Opt(BC)	Opt(C)

Breadth First Search



What Will Be Covered?

(1) Introduction and Motivation

(2) Algorithms Considered

(3) Implementation of Matrix Parenthesization

(4) Implementation of Breadth First Search

(5) Results

(6) Conclusions and Future Work

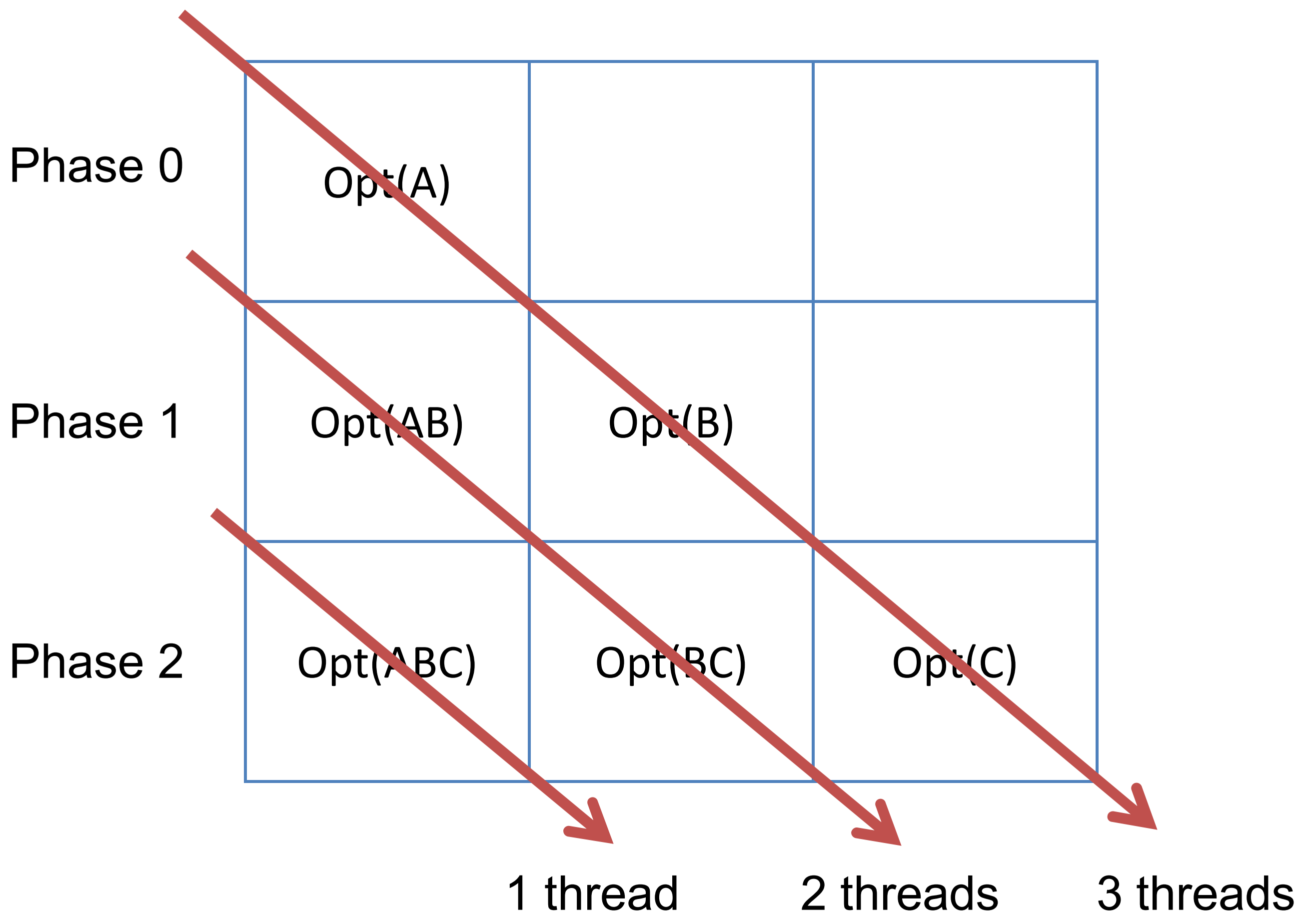
Matrix Parenthesization

- Synchronous algorithm, runs through diagonals in solution array each "phase".

```
for i = 1 to matrix chain length do
    Call MatrixParenGPU kernel for the current phase
    phase ← phase + 1
end for
```

- Phases of GPU computation controlled by CPU loop.
- Each GPU thread responsible for one cell in current diagonal of optimal costs matrix.

Matrix Parenthesization



What Will Be Covered?

(1) Introduction and Motivation

(2) Algorithms Considered

(3) Implementation of Matrix Parenthesization

(4) Implementation of Breadth First Search

(5) Results

(6) Conclusions and Future Work

Breadth First Search

- Synchronous traversal of graph.

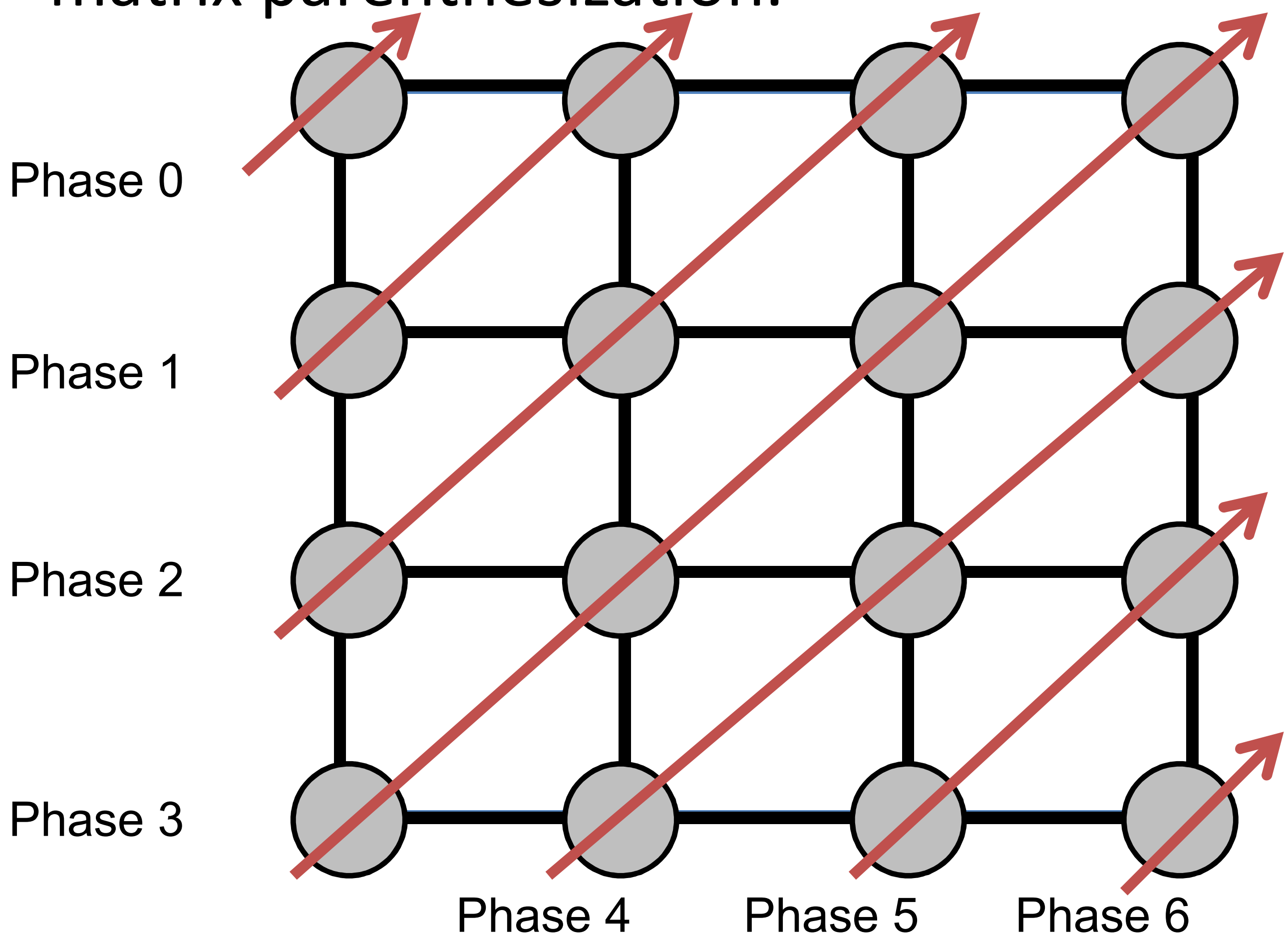
while There are still nodes to be processed **do**
    Call BFSGPU kernel for the current level
    level $\leftarrow$ level + 1
end while

- Phases of GPU computation controlled by CPU loop.

- Single thread manages one node in the graph.

Breadth First Search

- Grid graphs only, traversal structure is similar to matrix parenthesization.



What Will Be Covered?

(1) Introduction and Motivation

(2) Algorithms Considered

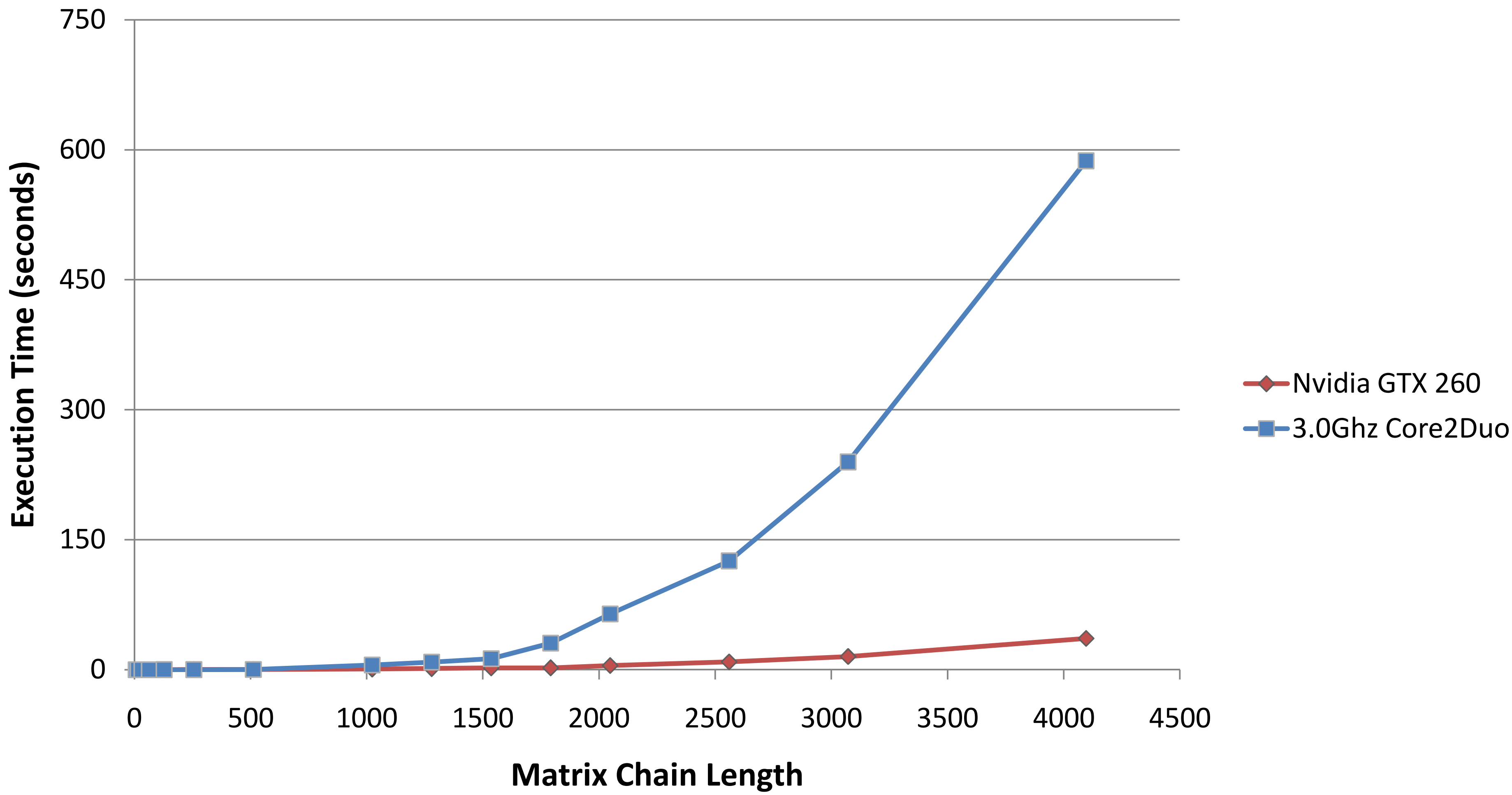
(3) Implementation of Matrix Parenthesization

(4) Implementation of Breadth First Search

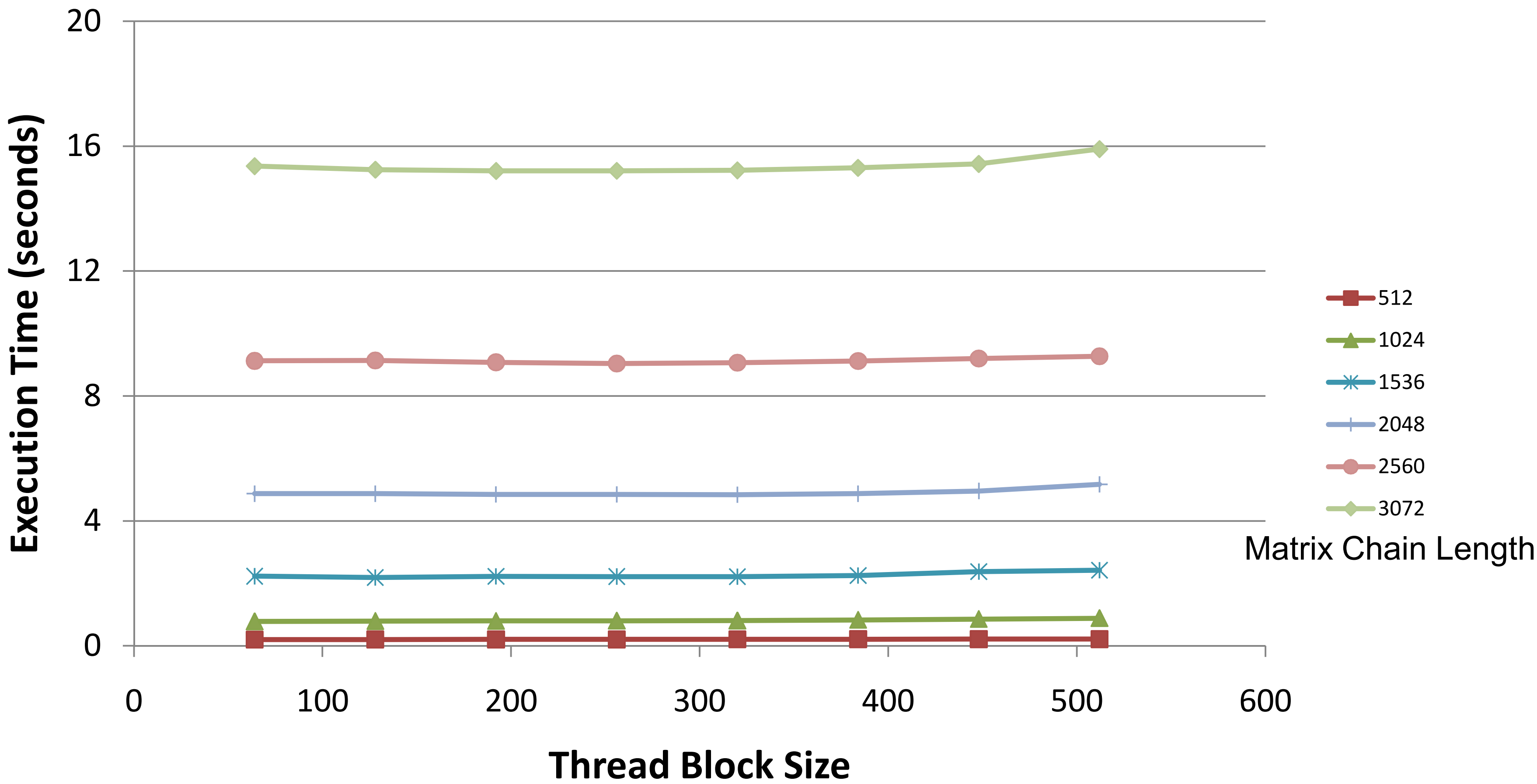
(5) Results

(6) Conclusions and Future Work

Results – Matrix Parenthesization

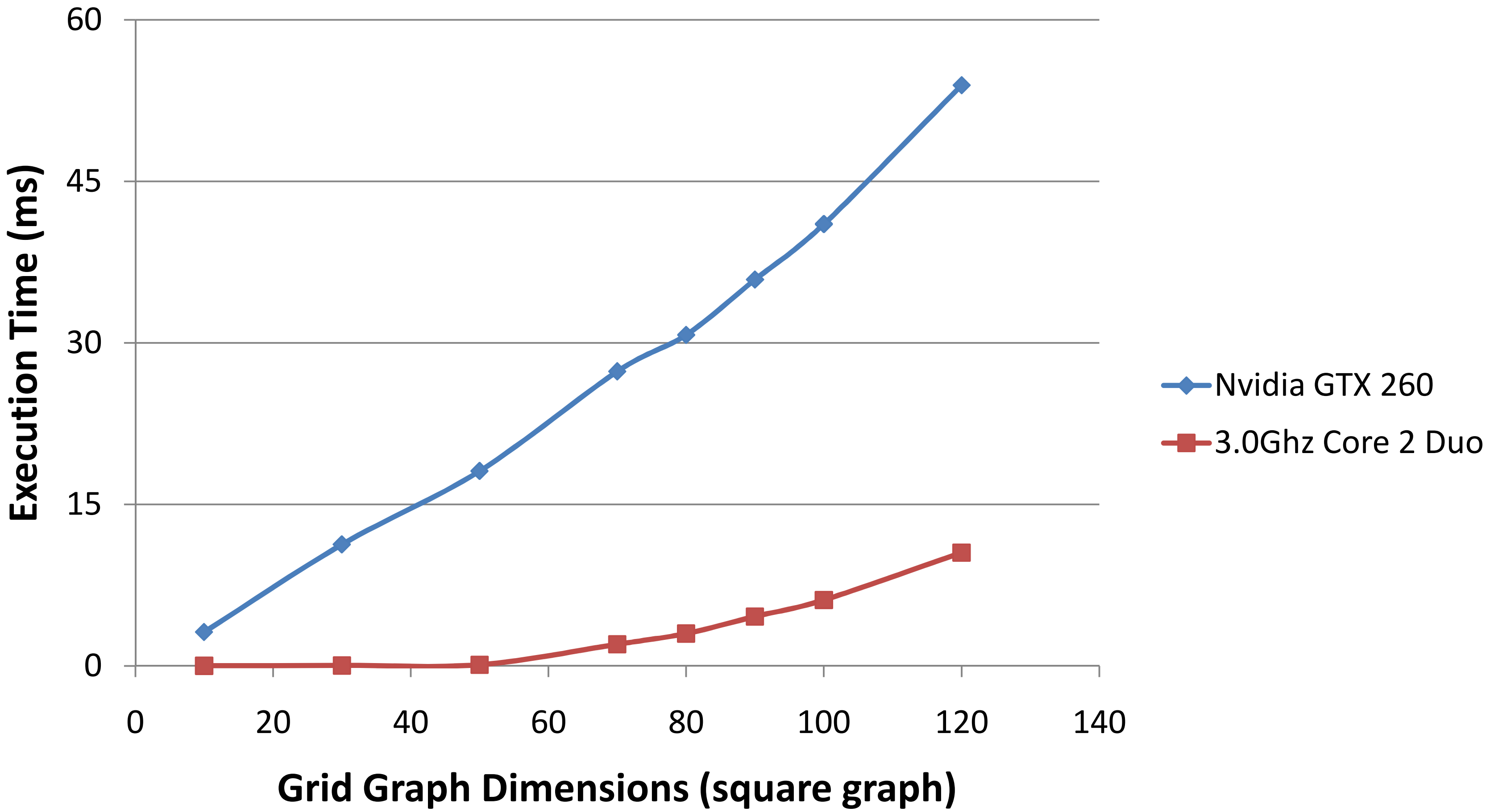


Results – Matrix Parenthesization



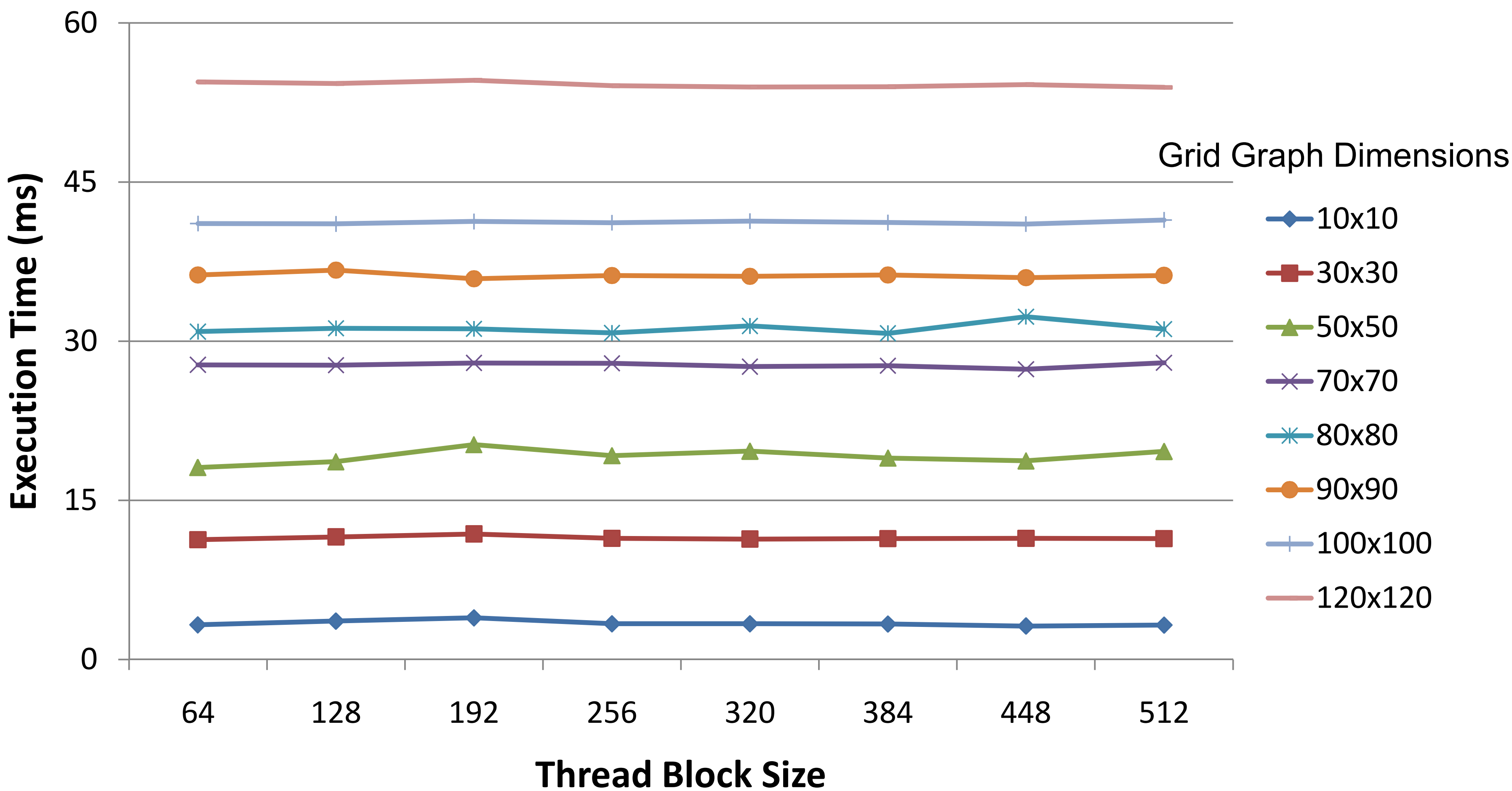
- Only a block size of 512 typically displayed noticeably worse performance

Results – Breadth First Search



• Worse performance on GPU – however, a linearly increasing execution time!

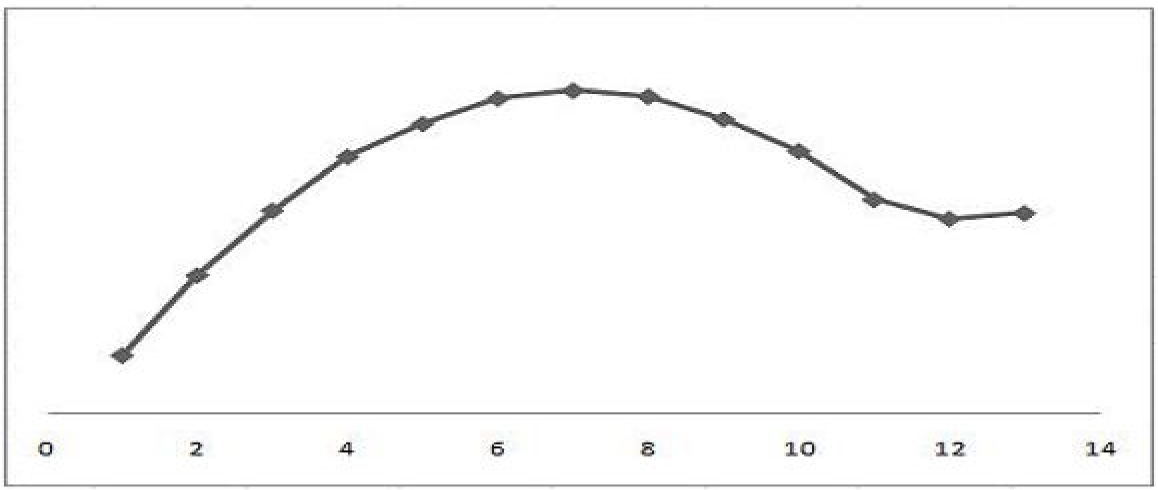
Results – Breadth First Search



- As with matrix parenthesization, no significant effects of thread block size on execution time are observed.

Results – Phase Performance

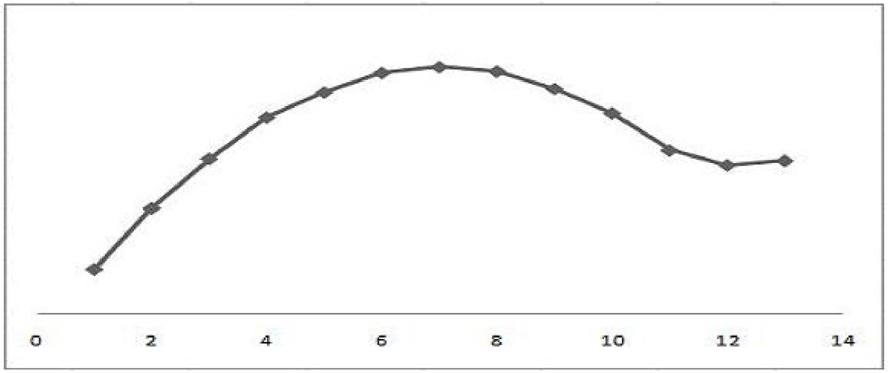
- Matrix Parenthesization – Gradual increase in execution time of phase groups.



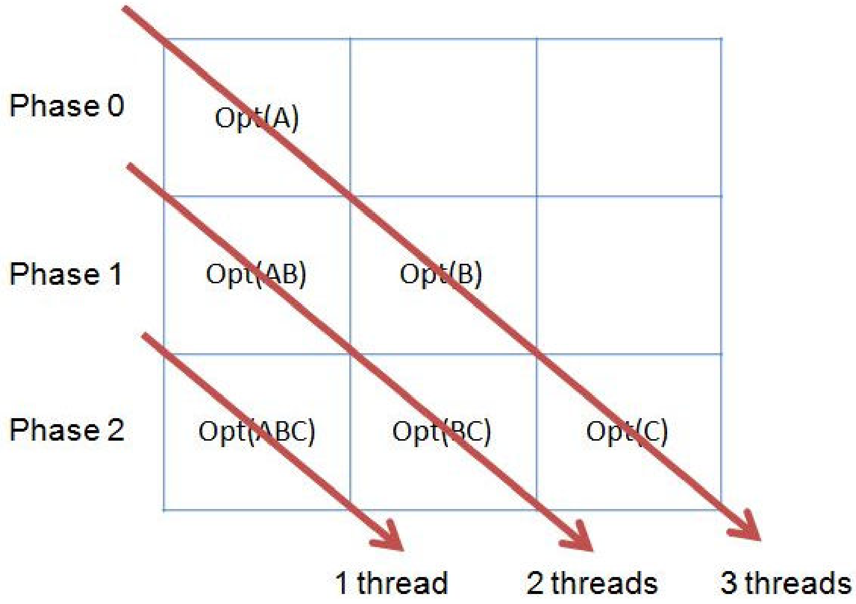
- Lowers at halfway point but never drops down fully.

Results - Matrix Parenthesization

- Losing parallelism at each subsequent phase.

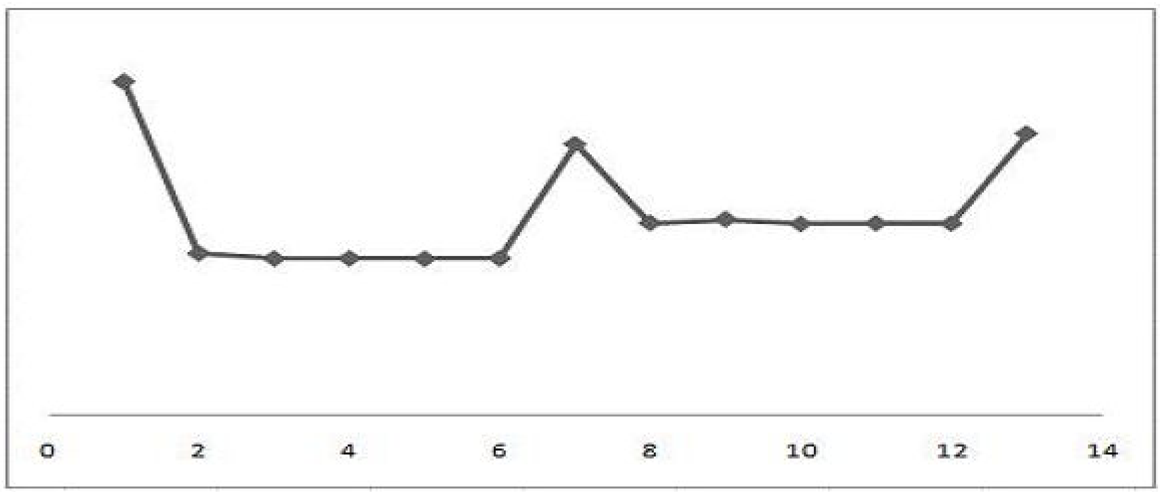


- Yet individual threads have more work to do in the later phases (optimal cost determination for longer and longer chains)



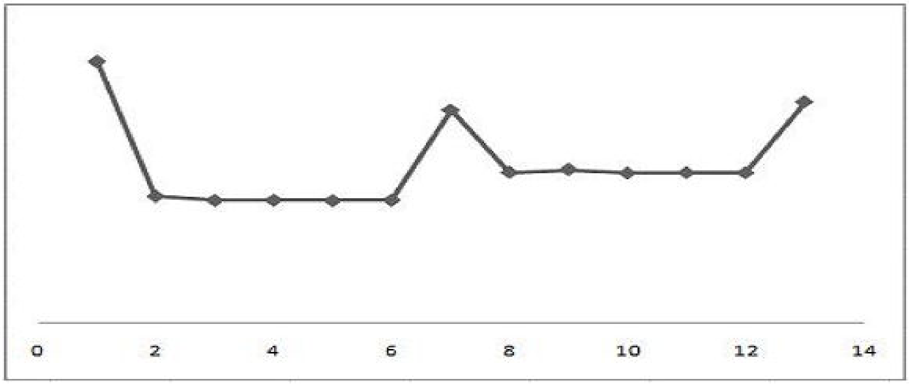
Results – Phase Performance

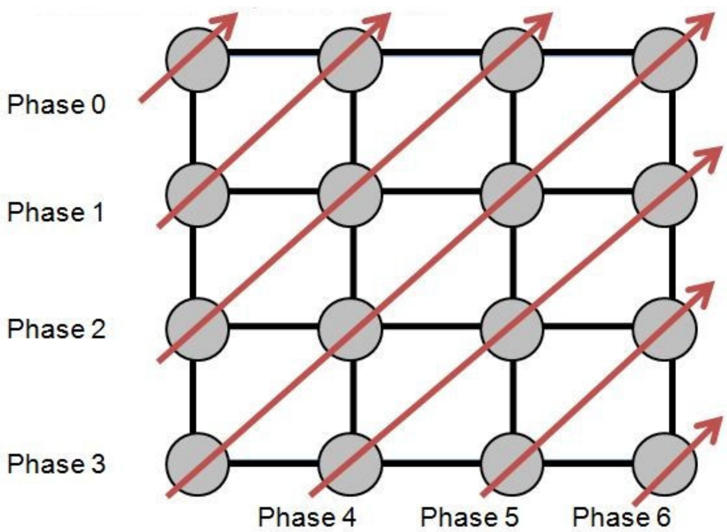
- Breadth First Search – higher execution time at start/end and middle phases.



Results – Breadth First Search

- Peak at middle not unexpected (largest number of active threads, greatest global memory accesses)
- Beginning/end phases a surprise, unsure exactly what is causing the peaks.





What Will Be Covered?

(1) Introduction and Motivation

(2) Algorithms Considered

(3) Implementation of Matrix Parenthesization

(4) Implementation of Breadth First Search

(5) Results

(6) Conclusions and Future Work

Future Work

- Currently, all GPU threads are launched even if they have no work to accomplish this phase.
  - Improved performance likely if we only launch threads that have work to do in the phase.

- CPU is used only to manage synchronization between phases.
  - Perhaps the CPU can do some useful work as well.

Conclusions

- Global memory latency is likely the significant factor impacting the performance of both algorithms.

- Irregular memory access prohibits memory optimization strategies.

- Enforced synchronization acts as another cause of performance degradations.

Conclusions

- The GPU provides significant computational power and parallelism.

- Global memory acts as a serious bottleneck for applications on the GPU, especially irregular applications.