Attempting to Improve APSP on the GPU

Terence Henriod

Intro

- Graphs are an abstraction of binary relationships between entities and the cost or weights of those relationships

- Sometimes we want to know the best way to navigate between one entity and another

- Sometimes we are interested in optimizing cost or reward of traversal, such as in the case of a road map or a probabilistic model

- Maybe we want to know the best places to start from in order to disperse something (related to the largest clique problem)

- No matter what, we want to find our answer FAST

My Results In Short...

- I failed to improve upon the APSP algorithm in a significant way
- I have found ways that don’t work or only work in non-target cases → maybe I have pruned the “search tree” of ways to improve
- I attempted the important things that I proposed and added some others

My Ideas for Improvement

1. **Solve subtrees of the recursion tree on separate, concurrent GPUs**
2. **Increase the β parameter to shorten the recursion tree**
3. **Increase the number of threads used in one of the existing kernels**
4. **Decrease the number of threads used (I’ll explain)**
5. Use shared memory to execute *some* of the kernels concurrently
6. Try to improve on Volkov’s SGEMM
7. Choose a different project (just kidding)

Solving Subtrees Independently

- This failed due to my own oversight
- Could’ve seen this by studying the math/regular expressions more closely; could’ve tried reordering the operations sooner rather than writing code to re-arrange it
- In short: impossible, one subtree is dependent on the solution of the other.

$A^* : \mathbb{R}^{N\times N} = \mathrm{APSP}(A : \mathbb{R}^{N\times N})$

if $N < \beta$

  then $A \leftarrow \mathrm{FW}(A)$ ▷ Base case: perform iterative FW serially

  else

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

$A_{11} \leftarrow \mathrm{APSP}(A_{11})$

$A_{12} \leftarrow A_{11}A_{12}$

$A_{21} \leftarrow A_{21}A_{11}$

$A_{22} \leftarrow A_{22} \oplus A_{21}A_{12}$

$A_{22} \leftarrow \mathrm{APSP}(A_{22})$

$A_{21} \leftarrow A_{22}A_{21}$

$A_{12} \leftarrow A_{12}A_{22}$

$A_{11} \leftarrow A_{11} \oplus A_{12}A_{21}$

Can be Rearranged

Can be Rearranged

Fig. 3. Pseudocode for recursive in-place APSP.

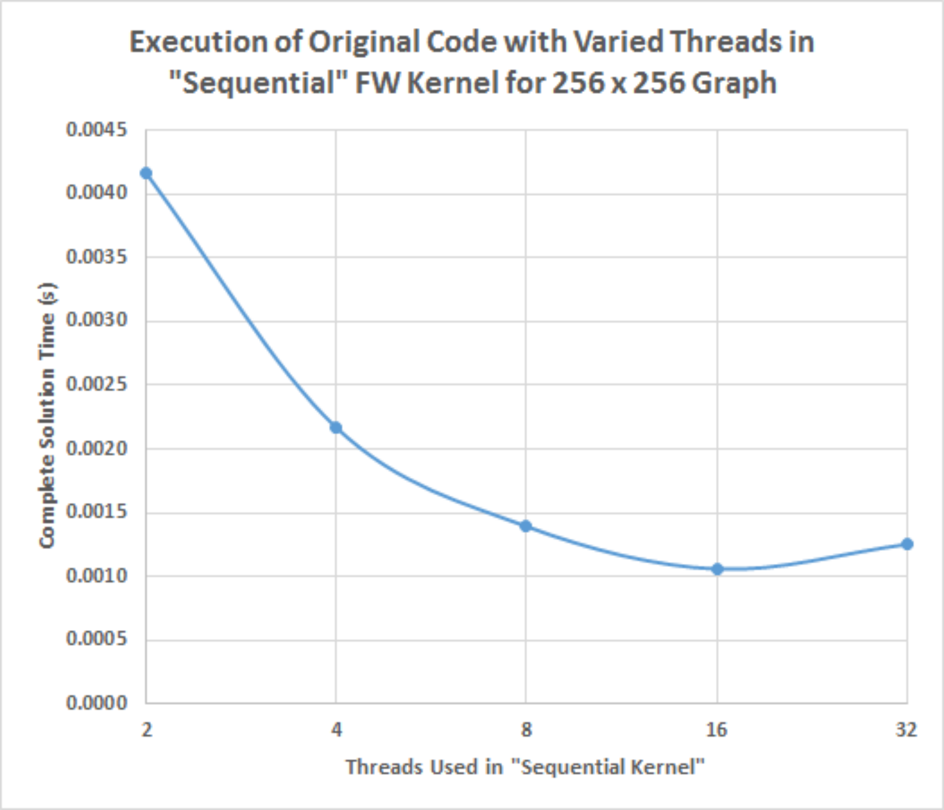
DOH!

Increase the β Parameter

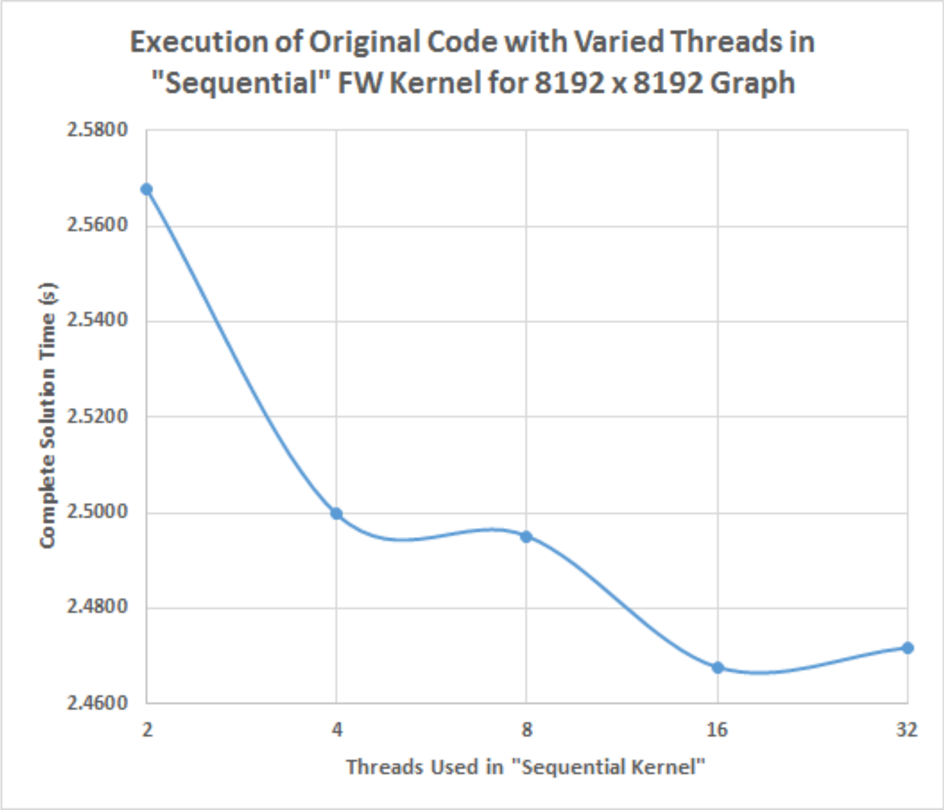
- Reduce recursion depth, reduce kernel launches, and maybe eliminate the need for a 3rd kernel!
- The kernels used:
  - if (block_width < β) → "sequential" multiply
  - else if (block_width < 64) → "regular" multiply
  - else → "Volkov" multiply

Increase the β Parameter / Increase the Number of Threads Used

- Attempt to reduce the number of kernel launches/have a faster kernel
- It's not very effective…
- Anecdotally: slightly better on a mobile GPU (but that wasn't the goal…)
- Actually a little worse than the 16 originally used on a 480



Execution of Original Code with Varied Threads in "Sequential" FW Kernel for 256 x 256 Graph
Complete Solution Time (s)
0.0045
0.0040
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
2
4
8
16
32
Threads Used in "Sequential Kernel"



Execution of Original Code with Varied Threads in "Sequential" FW Kernel for 8192 x 8192 Graph
Complete Solution Time (s)
2.5800
2.5600
2.5400
2.5200
2.5000
2.4800
2.4600
2
4
8
16
32
Threads Used in "Sequential Kernel"

Increase the β Parameter/Use Fewer Threads

- Again, attempt to reduce the number of kernel launches, also reduce global memory accesses, and give threads more work
- Not always correct - lost correctness above 8 threads
- Achieved speedup on smaller inputs/vs. smaller numbers of threads (but that’s not the goal…)

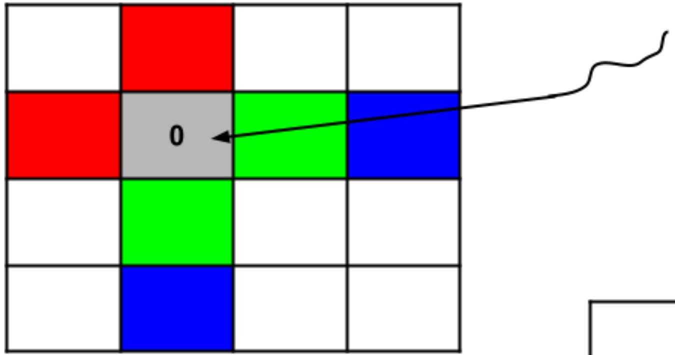
- The strategy:
  - Fetch memory into shared memory to reduce global memory accesses
  - Have threads do more work by having each thread compute results for 4 or 16 elements of the result rather than 1 (exploit Instruction Level Parallelism; see Volkov's work http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf)
  - Should be correct since in the original algorithm, each thread assessed one pair of neighbors independently at a time; I am simply simulating a larger number of threads doing the same thing

- Assumed values for example analysis:
  - 4 x 4 thread block → 4 x 4 small block, 8 x 8 large block
  - 600 cycles (~150 GB/s) per global memory access
  - 6 (~1.7 TB/s) for shared memory accesses
  - reads/writes considered similar for simplicity (I know they are different)
- Original Method Memory Access
  - each thread (4^2 threads) performs 2 reads and 1 write 4 times → 4^2 * (2 + 1) * 4 = 192 global accesses per call
  - 4 small blocks in a big block
  - 4 small blocks * 192 global accesses * 600 cycles = 460800 total cycles
- My method
  - each thread performs 4 global memory reads (start) and 4 writes (end)
  - each thread (4^2 threads) performs 8 shared memory reads and 4 writes 8 times → 4^2 * (8 + 4) * 8 = 1536 shared memory accesses
  - 1 big block * (1536 * 6 + 8 * 600) = 14016 total cycles
- 14016 << 460800

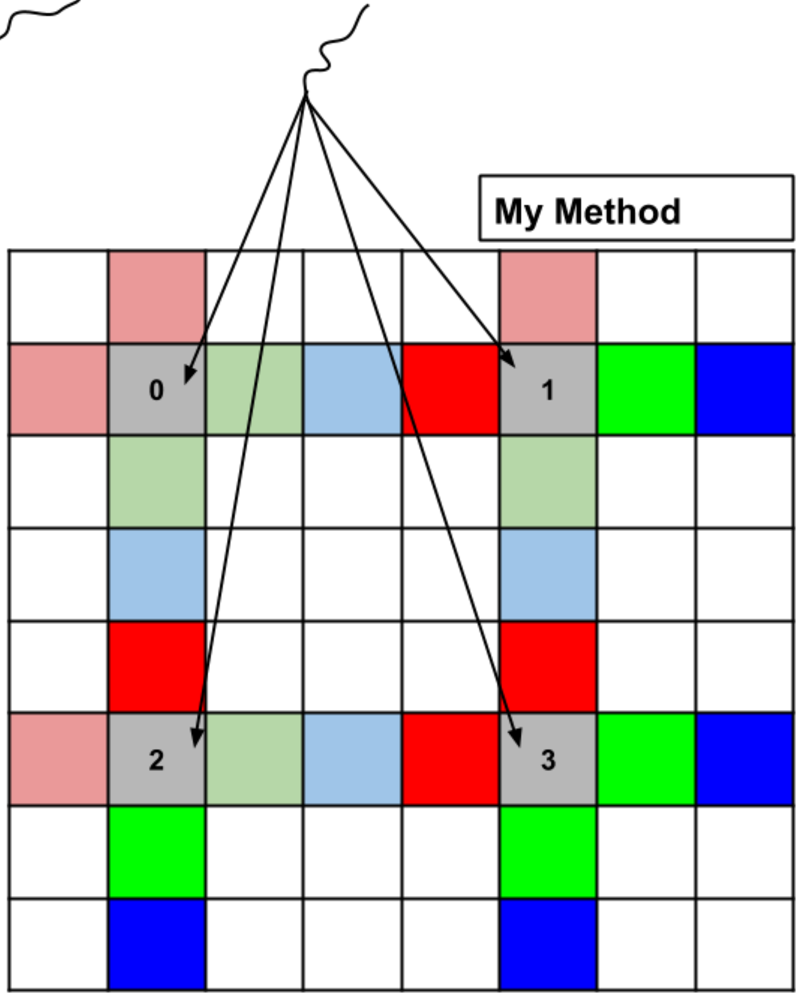
- In Big-Oh
- Original
  - 4 tiles * (t^2 * (2gr + 1 gw) * 4)
  - = 32 * t^2 * gr + 16 t^2 * gw
  - = O(28800 * t^2)

  - Mine: 1 tile * (t^2 * (4gr + 4gw + 8sr + 4sw))
  - = 4*t^2*gr + 4*t^2*gw + 8*t^2*sr + 4*t^2*sw
  - = O(4860 * t^2)

***If we stick to our 600 cycles for global and 6 cycles for shared assumption



Original Method



My Method

Example: The thread with
threadId.x == 1,
threadId.y == 1

x16 threads in both cases

Each operation is presumably independent.

- Could it be that allocating a small array in registers in the “regular” multiplies is better than dynamically allocating shared memory in the “sequential” multiply?

Didn’t Try: Execute *Some* Kernels Independently

- Without any machines with *integrated* GPUs available, this seemed like a lost cause

- With physically unified memory, using “pinned memory” techniques or true Unified Memory (UM) might have produced some improvement

- As it stands, the hit from accessing non-UM multiple times from a discrete GPU probably would have been detrimental

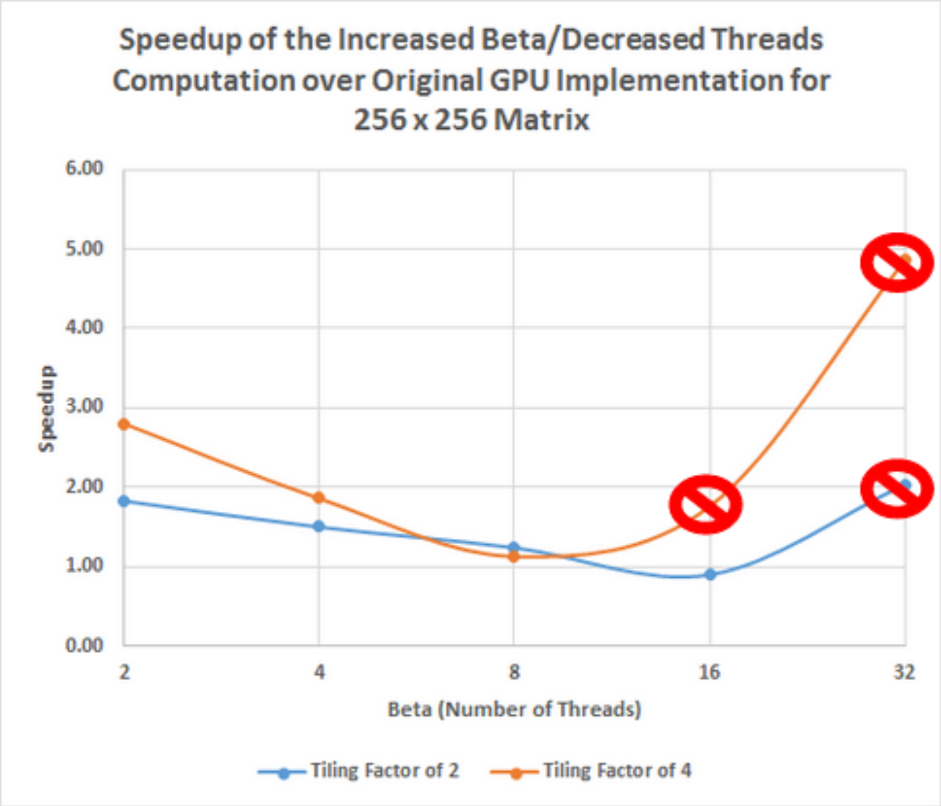
Didn’t Try: Improve Volkov’s SGEMM

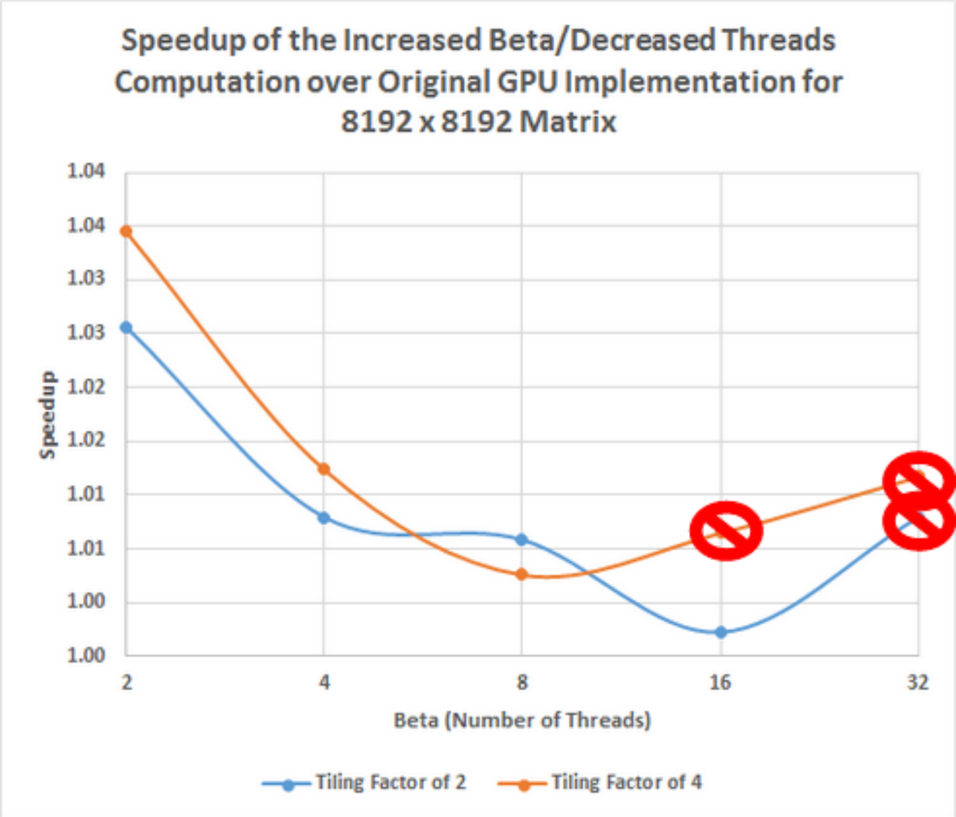
- Volkov is a wiz at SGEMM - likely little room for improvement

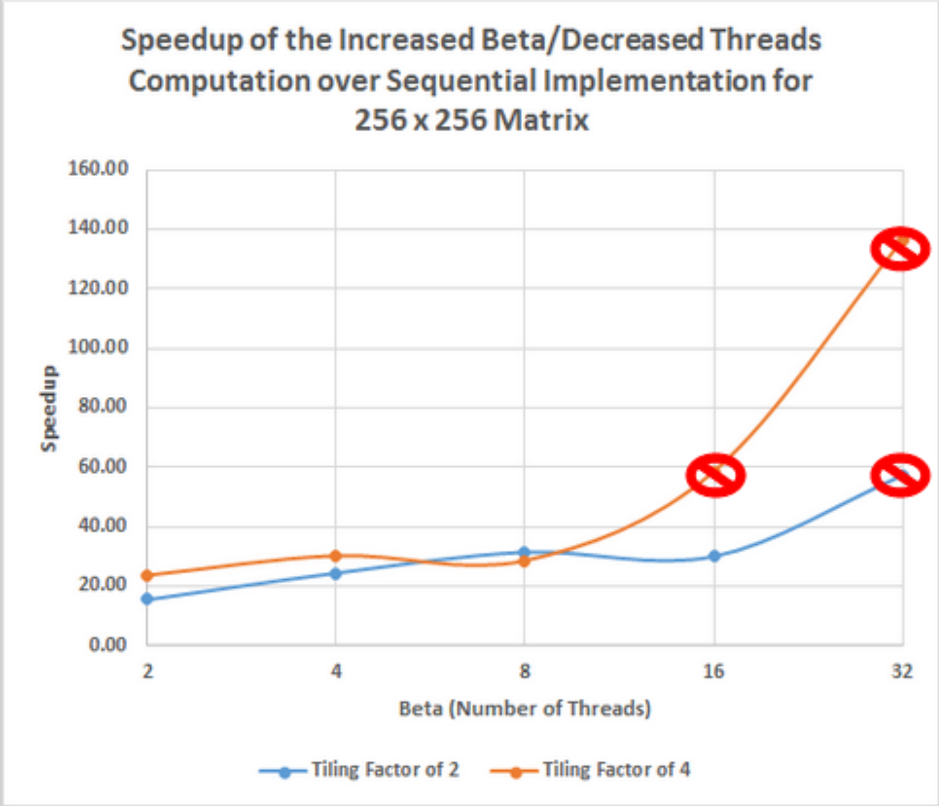
- There seems to be intricate hand tuning involved - low probability of improving it anyway

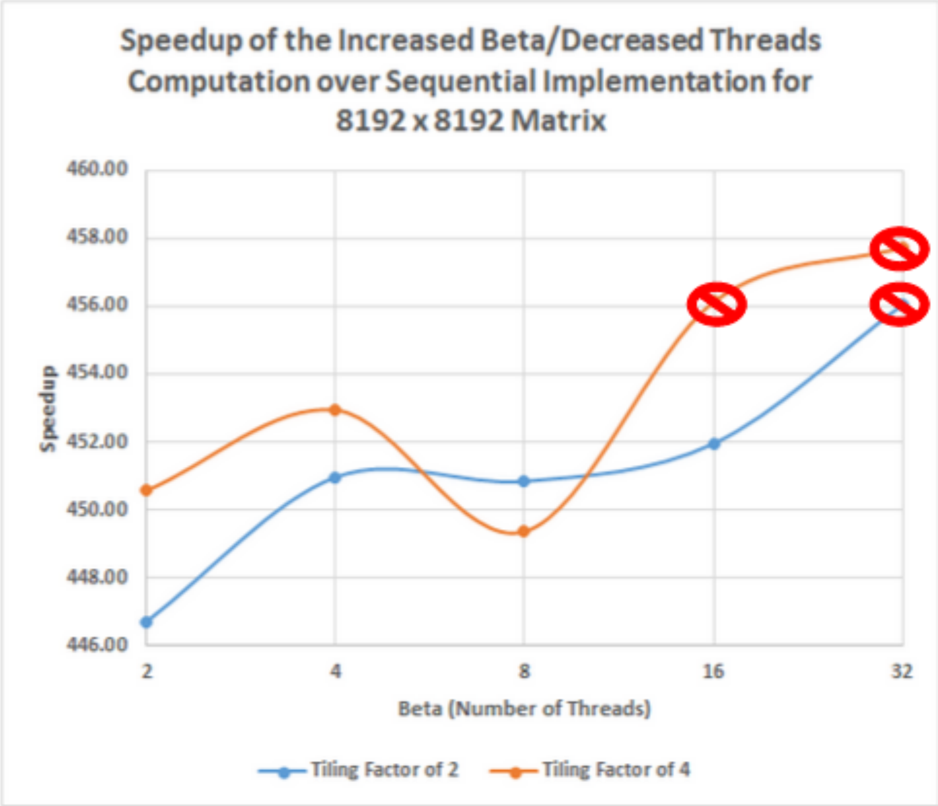
```
__global__ void sgemmNN_MinPlus(
  const float *A, int lda, const float *B, int ldb, float* C,
  int ldc, int k, float beta ) {

  const int inx = threadIdx.x;
  const int iny = threadIdx.y;
  const int ibx = blockIdx.x * 64;
  const int iby = blockIdx.y * 16;
  const int id = inx + iny*16;

  A += ibx + id;
  B += inx + __mul24( iby + iny, ldb );
  C += ibx + id  + __mul24( iby, ldc );

  const float *Blast = B + k;

  float c[16] = {FLOATINF, FLOATINF, FLOATINF, FLOATINF,
          FLOATINF, FLOATINF, FLOATINF, FLOATINF,
          FLOATINF, FLOATINF, FLOATINF, FLOATINF,
           FLOATINF, FLOATINF, FLOATINF, FLOATINF};

  do {
    float a[4] = { A[0*lda], A[1*lda], A[2*lda], A[3*lda] };

    __shared__ float bs[16][17];
    bs[inx][iny]    = B[0*ldb];
    bs[inx][iny+4]  = B[4*ldb];
    bs[inx][iny+8]  = B[8*ldb];
    bs[inx][iny+12] = B[12*ldb];
    __syncthreads();
```

```
    A += 4*lda;
    saxpy_MinPlus<16>( a[0], &bs[0][0], c );    a[0] = A[0*lda];
    saxpy_MinPlus<16>( a[1], &bs[1][0], c );    a[1] = A[1*lda];
    saxpy_MinPlus<16>( a[2], &bs[2][0], c );    a[2] = A[2*lda];
    saxpy_MinPlus<16>( a[3], &bs[3][0], c );    a[3] = A[3*lda];

    A += 4*lda;
    saxpy_MinPlus<16>( a[0], &bs[4][0], c );    a[0] = A[0*lda];
    saxpy_MinPlus<16>( a[1], &bs[5][0], c );    a[1] = A[1*lda];
    saxpy_MinPlus<16>( a[2], &bs[6][0], c );    a[2] = A[2*lda];
    saxpy_MinPlus<16>( a[3], &bs[7][0], c );    a[3] = A[3*lda];

    A += 4*lda;
    saxpy_MinPlus<16>( a[0], &bs[8][0], c );    a[0] = A[0*lda];
    saxpy_MinPlus<16>( a[1], &bs[9][0], c );    a[1] = A[1*lda];
    saxpy_MinPlus<16>( a[2], &bs[10][0], c );   a[2] = A[2*lda];
    saxpy_MinPlus<16>( a[3], &bs[11][0], c );   a[3] = A[3*lda];

    A += 4*lda;
    saxpy_MinPlus<16>( a[0], &bs[12][0], c );
    saxpy_MinPlus<16>( a[1], &bs[13][0], c );
    saxpy_MinPlus<16>( a[2], &bs[14][0], c );
    saxpy_MinPlus<16>( a[3], &bs[15][0], c );

    B += 16;
    __syncthreads();
  } while( B < Blast );


  for( int i = 0; i < 16; ++i, C += ldc )
  {
    C[0] = fminf(c[i],beta*C[0]);
  }

}
```


Speedup of the Increased Beta/Decreased Threads Computation over Original GPU Implementation for 256 x 256 Matrix
6.00
5.00
4.00
3.00
2.00
1.00
0.00
Speedup
2
4
8
16
32
Beta (Number of Threads)
Tiling Factor of 2
Tiling Factor of 4


Speedup of the Increased Beta/Decreased Threads Computation over Original GPU Implementation for 8192 x 8192 Matrix
1.04
1.04
1.03
1.03
1.02
1.02
1.01
1.01
1.00
1.00
Speedup
2
4
8
16
32
Beta (Number of Threads)
Tiling Factor of 2
Tiling Factor of 4


Speedup of the Increased Beta/Decreased Threads Computation over Sequential Implementation for 256 x 256 Matrix
160.00
140.00
120.00
100.00
80.00
60.00
40.00
20.00
0.00
Speedup
2
4
8
16
32
Beta (Number of Threads)
Tiling Factor of 2
Tiling Factor of 4


Speedup of the Increased Beta/Decreased Threads Computation over Sequential Implementation for 8192 x 8192 Matrix
460.00
458.00
456.00
454.00
452.00
450.00
448.00
446.00
Speedup
2
4
8
16
32
Beta (Number of Threads)
Tiling Factor of 2
Tiling Factor of 4

The Hardware

- CPU
  - Intel Core i7-4790
  - 15.6 GiB RAM
  - 8 MB cache
  - 3.60 GHz
- GPU
  - GeForce GTX 480
  - 15 SM; 448 Cores
  - 1280 MB RAM
  - 786 kB L2 cache
  - 1.4 GHz

If I had another month...

- I’d allow myself a week to try and fix the increased β/fewer threads approach
- Spend 2 days (at most) trying the dual GPU with shared memory for concurrent kernels approach
- Add blocks to the “normal” multiply kernel
- Try out concurrent kernel launches where possible
- Tuning or improving Volkov’s SGEMM

Thank You

Questions?