CSE 260 Programming Assignment 2

Honam Bang, Myron Liu

Friday 30^{th} October, 2015

<u>Note</u>: Our program only accepts block sizes $\{bx, by\}$ such that by divides bx. *e.g.* $\{bx, by\} = \{128, 32\}$

1 Notation

In this document we will use the following notation:

- A, B, C refer to the original input matrices
- A', B', C' refer to blocks of the input matrices
- n is the dimension of the input matrices
- w is the width of blocks B' and C'. This is equal to the <u>number of threads per block</u>
- h is the height of blocks A', B', C'. It is also the width of B'
- To reiterate: A' is $h \times h$ and B', C' are $h \times w$

2 Implementation

Our final implementation is based on the paper *Benchmarking GPUs to Tune Dense Linear Algebra* by Volkov and Demmel. We won't bother restating the algorithm in detail here. However, we will note that in our implementation, rather than sharing blocks of B between threads, we share blocks of A (which is a square $h \times h$ matrix). In the same vein, our w threads correspond to the columns of B and C, each of which has dimensions of $h \times w$. The reason for this change is that the paper assumes column-major matrix format, while our implementation uses row-major matrix format. This isn't merely for convenience. Among other things, this modification is necessary to coalesce our accesses to global memory, on which we elaborate in the subsequent section.

3 Discussion: Coalescing Accesses to Global Memory

Before we dive into the key advantages of Volkov's technique, we address some common pitfalls that we must avoid. First and foremost, we should make sure that we coalesce our accesses to global memory.

To accumulate C', we perform the dot product between the rows of A' and their corresponding columns of B'. Since our threads are associated with successive columns of B', each subsequent thread accesses the adjacent column in the same row. As such, threads in the same warp will simultaneously load contiguous elements of B into their own local register b. This is precisely what it means for the accesses to global memory to coalesce. To illustrate this pictorially, suppose our warps are 4 threads wide, then for a given warp our accesses to global B would look something like:

$$\begin{pmatrix} \ddots & \vdots & \vdots & \vdots & \vdots & \iddots \\ \cdots & \colorbox{lavender}{a} & \colorbox{lavender}{b} & \colorbox{lavender}{c} & \colorbox{lavender}{d} & \cdots \\ \cdots & e & f & g & h & \cdots \\ \cdots & i & j & k & l & \cdots \\ \iddots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \longrightarrow \begin{pmatrix} \ddots & \vdots & \vdots & \vdots & \vdots & \iddots \\ \cdots & a & b & c & d & \cdots \\ \cdots & \colorbox{lavender}{e} & \colorbox{lavender}{f} & \colorbox{lavender}{g} & \colorbox{lavender}{h} & \cdots \\ \cdots & i & j & k & l & \cdots \\ \iddots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \longrightarrow \begin{pmatrix} \ddots & \vdots & \vdots & \vdots & \vdots & \iddots \\ \cdots & a & b & c & d & \cdots \\ \cdots & e & f & g & h & \cdots \\ \cdots & \colorbox{lavender}{i} & \colorbox{lavender}{j} & \colorbox{lavender}{k} & \colorbox{lavender}{l} & \cdots \\ \iddots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \longrightarrow \quad \cdots$$

which is ideal for row-major matrix formats.

The same considerations apply when constructing shared B' from global B. That is, we also want the threads in a warp to load adjacent elements of B concurrently. For instance, in our row-major framework, with block dimensions $\{w, h\} = \{64, 16\}$, we should have the 64 threads load the first four rows of B' (*i.e.* the first 64 elements of B'), followed by rows 5 to 8, etc... This coalesces the global memory access. We were careful to coalesce our write-backs to C as well.

4 Discussion: Bank Conflicts

In the GPU, the local memory is divided into memory banks. Each bank can only address one data-set at a time, so if a warp tries to load/store data from/to the same bank, the access has to be serialized (this is called a bank conflict). For this assignment, we used Nvidia's GPU (Tesla, K80). There are 32 banks, which are interleaved with a granularity of 32 bits (bytes 0-3 are in bank 1, bytes 4-7 in bank 2, ..., bytes 128-131 in bank 32) or with a granularity of 64 bits (bytes 0-8 in bank 1, bytes 9-12 in bank 2, etc...).

The following pseudocode performs the rank-1 update. It is a point of confusion for us because it appears that there would be bank conflicts galore. After all, each thread in the warp simultaneously accesses the same element As[i][k] of A', which is the worst case scenario.

Algorithm 1 Bank-conflict-riddled (?) rank-1 update

```
// As is the shared block of A
for (int k = 0; k < h; ++k)
{
    b = B[Bs_index + N*k + thread_index]; // Bs_index is the index at which the B block begins

    unroll for (int i = 0; i < h; ++i)
    {
        c[i] += b * As[i][k];
    }
}
```

The best explanation we can come up with is that the hardware executes the instruction in the body of i-loop out-of-order. This is somewhat plausible because we have unrolled the loop. We actually tried to avoid bank conflicts manually by replacing all occurrences of k with (k+thread_index)%w. Even if this solves the issue of bank conflicts, this replacement ends up slowing down our implementation to a crawl, so we excluded it.

Regardless, we are confident that in actuality we have no bank conflicts because we profiled our code using the 'nvprof' command.

nvprof –events shared_load_replay ./mmpy -n 1024 -r 1 -R

5 Discussion: Using Fewer Threads to Increase Overall Performance

The conventional advice for achieving peak performance is to increase the number of threads. However, this is rarely optimal unless we have an enormous number of registers (enough to store local copies of all shared data). The limited number of registers means that we must rely on shared memory for data that is common between threads, so that we do not store an excessive number of duplicates. The problem with shared memory is that although it is blisteringly fast relative to global memory, it operates at a snail's pace relative to registers.

Volkov's methodology leverages the speed of registers by using shared memory for A' only. In contrast to A', B' is loaded into register b from global memory. Since we have w threads, we load (in parallel among the threads) a row of B' at time. After h iterations, each thread will have accessed each element of it's associated column, sequentially replacing the value in b. **The decreased number of accesses to shared memory gives Volkov's technique an overwhelming advantage.**

Of course, there are tradeoffs. Volkov's algorithm uses more registers per matrix-block than does the naive algorithm (the ratio of registers used is $1 + 1/h$); however, the naive algorithm uses more shared memory. So one choice for block dimensions might work for one but not for the other.

6 Discussion: Thread Divergence and Boundary Blocks

To accommodate matrices of arbitrary size, we must be able to handle partial blocks when the block dimensions don't neatly divide the full matrix. To remedy this issue, we pad partial blocks $A'_{partial}$ and $B'_{partial}$ with zeros and treat them henceforth as full blocks. C' can then be computed in the normal way. Of course, we must be careful when copying the entries of C' back to the full matrix C, since C' may contain a border of zeroes, but this is easily handled.

The presence of different program paths leads to thread divergence. On matrices that evenly divide the block sizes, this isn't much of an issue, since the GPU is smart enough that it detects untaken paths and ignores them completely. Even when the blocks don't tile the input matrices uniformly, we can expect approximately the same performance if the block dimensions are small relative the the entire matrix. This is because only the last block in each block-row or block-column will be truncated. The rest of the blocks in the row/column will still be complete blocks, which skirt by the issue of thread divergence.

Also important to note is that the various program paths to be iterated over do not necessarily have the same execution time. When we load shared B' from global B, the *padding-program-path* is fast, since it does not access global memory. So rather than increasing the execution time for constructing B' by a factor of two, the overhead is much smaller.

Thread divergence manifests itself in the saw-tooth pattern of figure 4. We see local peak performance at $n = 512, 768, 1024$ since these can be uniformly tiled by our blocks of dimension $\{w, h\} = \{128, 32\}$; why the same phenomena is not observed at $n = 128, 256$ is something that we aren't clear on.

Notice how the performance tanks most steeply right after peak performance. This is because for $n = n_{peak} + 1$ we end up with partial blocks that are only 1 row wide, 1 column wide, or both (for the corner). This is a sort of worst case scenario, where we add $2 * n_{peak} + 1$ blocks to each input matrix just for the sake of computing $2 * n_{peak} + 1$ extra elements of C. After this dip, our performance gradually increases until we reach the next local peak: this can be seen in figure 4. We expect the dips to saw-tooth pattern to attenuate as we increase the input matrix sizes to ever larger values. The reason is that the wasted computation for the partial blocks constitutes a progressively smaller fraction of the total arithmetic operations and so has a smaller impact of GFLOPS with is a metric averaged over all arithmetic operations.

Of course, in an attempt to bypass the issue of thread divergence entirely, we could also tune our block sizes to evenly divide the input matrices (unless n's smallest divisor is excessively large). This would be a foolish choice though. As aforementioned, only a minority of the blocks are truncated, so thread divergence is infrequent. It is far more favorable to choose a block size that uses the SMs' resources optimally. Indeed, qualitatively, this is what we found.

7 Results

We were able to achieve up to **450** Gflops/s with the aforementioned implementation.

Block Size (X,Y) \ Input size (NxN)	256x256	512x512	768x768	1024x1024	1280x1280	1536x1536	1792x1792	2048x2048
(16,16)	82	128.74	121.91	130.44	131.62	130.64	131.14	132.47
(32,16)	177.09	241.87	272.43	290.53	284.07	285.32	287.79	292.22
(32,32)	116.79	242.77	275.1	327.01	340.76	341.18	330.28	320.944
(64,16)	189.14	315.58	383.38	392.63	396.73	377.21	373.08	370.41
(64,32)	128.69	426.77	415.77	428.66	433.14	439.14	434.01	423.84
(128,16)	176.35	398.17	393.06	391.75	402.42	399.61	401.1	400.12
(128,32)	134.36	430.36	372.47	445.12	423.63	419.13	421.5	419.27
BLAS with Multicore	9.49	22.5	35.1	41.5	53.2	48.3	56.3	51

Sweet spot : block size (64,32) to (128,32) and input size 1024 to 1792

Figure 1: Matrix multiplication speed for various combinations of block and input sizes

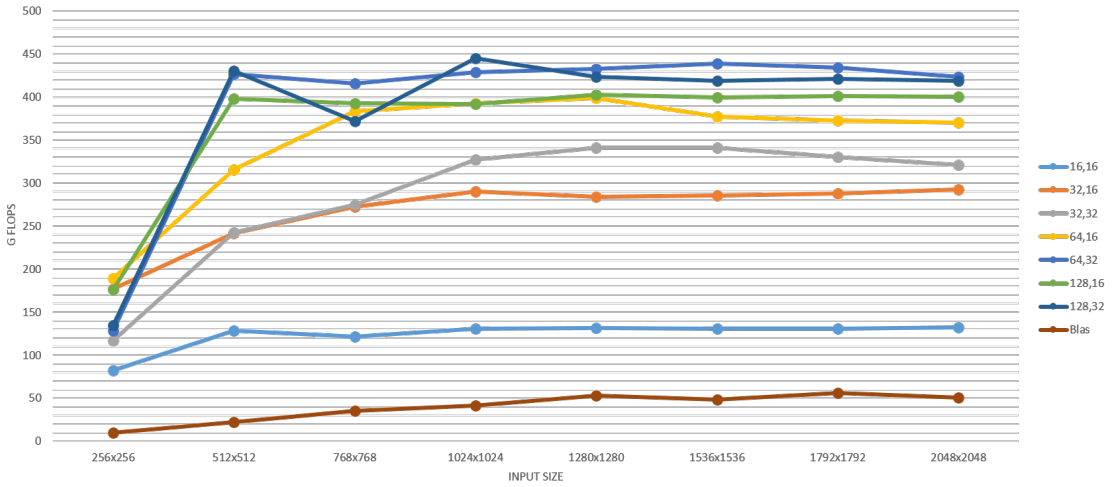

Figure 2: Matrix multiplication speed for various combinations of block and input sizes

Input size (NxN) / Block size (X,Y)	99	111	128	137	150	178	190	212	237	270	300	350	400	450	550	600	650	700	750	800	850	900	950	1000
(128, 32)	17.73	24.44	37.43	37.22	49.92	71.66	85.74	99.98	108.8	139.7	173	244	294.2	310.2	342.1	290.4	287.6	332.1	350.5	403.2	345.2	331	375.2	411.9

Figure 3: Matrix multiplication Performance for various input sizes (see plot in figure 4)

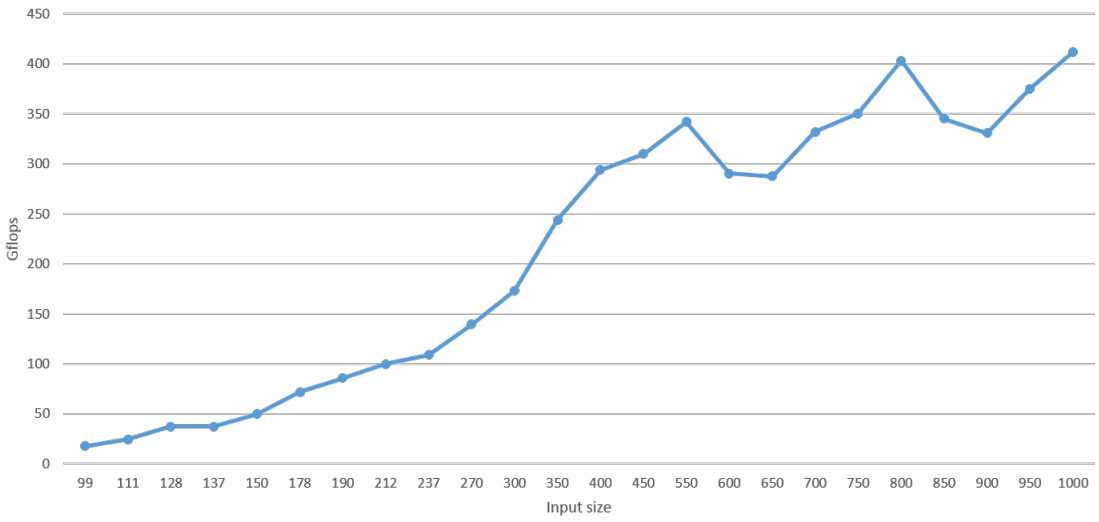

Figure 4: Matrix-multiplication speed for various input sizes. The block dimensions are $\{w, h\} = \{128, 32\}$