Contact:

www.cedricnugteren.nl
web@cedricnugteren.nl

Tutorial: OpenCL SGEMM tuning for Kepler

Note: the complete source-code is available at GitHub.
Note: a tuned OpenCL BLAS library based on this tutorial is now available at GitHub.

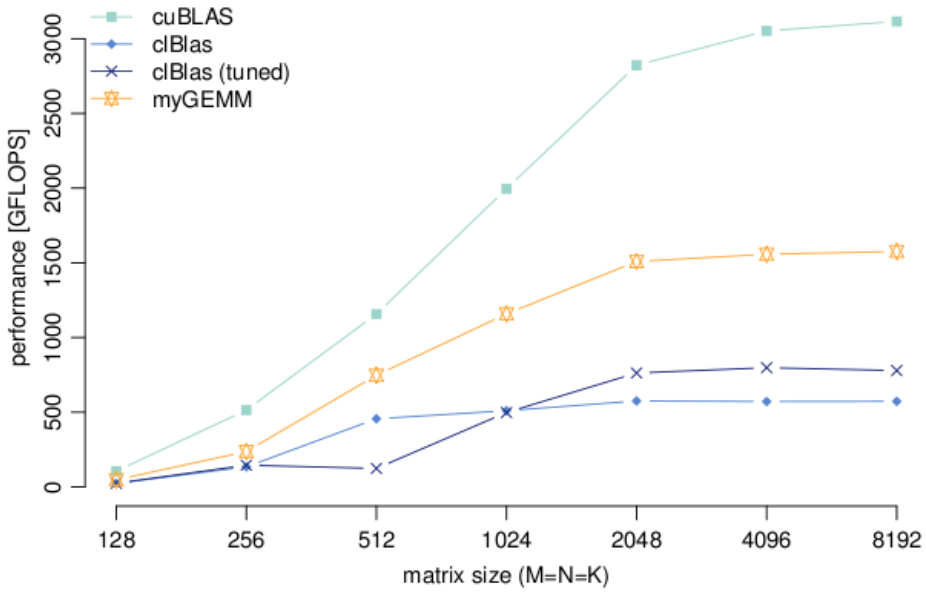
next ->

Introduction

This article describes a GPU OpenCL implementation of single-precision matrix-multiplication (SGEMM) in a step-by-step approach. We'll start with the most basic version, but we'll quickly move on towards more advanced code. Each step introduces a new optimisation - and best of all - working OpenCL code. This means that you'll be able to test it and tune it for your own machine. To make it even easier for you, there is also a GitHub repository online with a benchmarking infrastructure and kernel code for each step.

The first few steps of this article are rather basic, so those familiar with tiling might want to skip ahead. Nonetheless, these first steps can be very useful for those that want to learn about basic OpenCL optimisations (in general) or the design space of parallel matrix-multiplication. The remainder of the article is targeted at those that want to get decent matrix-multiplication performance and are familiar with concepts such as bank conflicts, warps, assembly code, vector operations and instruction latency.

The main reason why I wrote this article - and the code - is the poor performance of the clBlas library on NVIDIA GPUs. I worked on a project that required acceleration of code on an NVIDIA Tesla K40m GPU using OpenCL. Since its main component was a dense single-precision matrix-multiplication, I made a call to the SGEMM routine of clBlas. It turned out that clBlas is roughly a factor 5-6 slower (on my GPU) compared to its CUDA counterpart cuBLAS: clBlas does not get much more than 500 GFLOPS (out-of-the-box) or 700 GFLOPS (tuned), whereas the far superior cuBLAS reaches a little over 3 TFLOPS (~80% of the GPU's peak performance). This is shown in the image below, which also includes my best-case performance so far ("myGEMM").



Like I said, we'll go over the optimisations step-by-step. And remember, they all come with OpenCL code, so please _do_ try this at home. Just a quick heads-up of what the steps are:

1. Naive implementation
2. Tiling in the local memory
3. Increased work per thread
4. Wider data-types (vectors)
5. Transposed input matrix and rectangular tiles
6. 2D register blocking
7. Wider loads with register blocking
8. CUDA and Kepler-specific optimisations
9. Software pre-fetching
10. Incomplete tiles and support for arbitrary matrix-sizes

Technical notes: All tests were performed on a Kepler SM 3.5 GPU, the Tesla K40m. The GPU was configured with ECC enabled. Version 6.5 of the CUDA toolkit was used (including OpenCL).

next ->

Tutorial written by Cedric Nugteren, (c) 2014 SURFsara
Last updated in 2014 - information might be outdated

Contact:

www.cedricnugteren.nl
web@cedricnugteren.nl

Tutorial: OpenCL SGEMM tuning for Kepler

Note: the complete source-code is available at GitHub.
Note: a tuned OpenCL BLAS library based on this tutorial is now available at GitHub.

<- previous next ->

Background: Matrix-multiplication

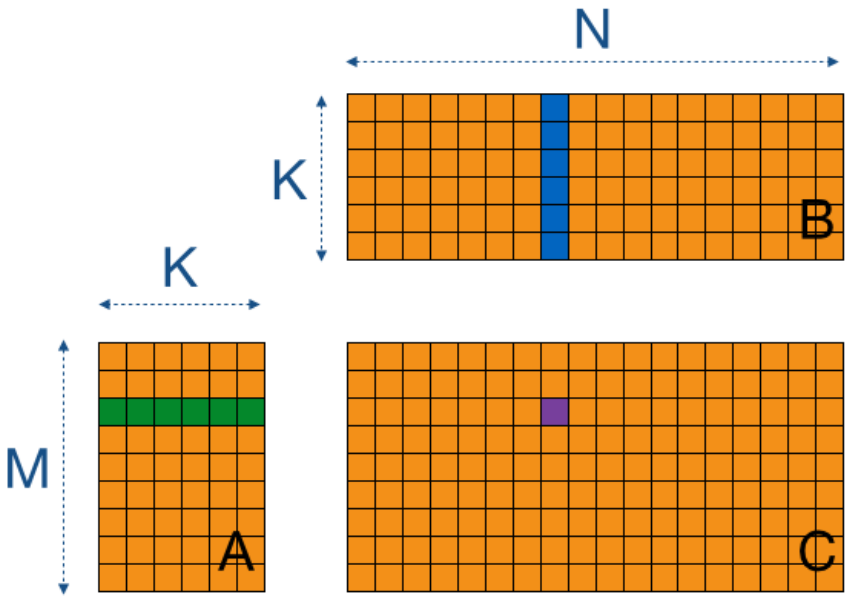
Before we look at OpenCL code, let us define the computation we are about to perform. According to the definition of BLAS libraries, the single-precision general matrix-multiplication (SGEMM) computes the following:

C := alpha * A * B + beta * C

In this equation, A is a K by M input matrix, B is an N by K input matrix, C is the M by N output matrix, and alpha and beta are scalar constants. For simplicity, we assume the common case where alpha is equal to 1 and beta is equal to zero, yielding:

C := A * B

This computation is illustrated in the following image: to compute a single element of C (in purple), we need a row of A (in green) and a column of B (in blue).



This version of SGEMM can be implemented in plain C using 3 nested loops (see below). We assume data to be stored in column-major format (Fortran-style), following cuBLAS's default. If we wanted, we could easily change this to row-major by swapping the A and B matrices and the N and M constants, so this is not a real limitation of our code.

```
1. for (int m=0; m<M; m++) {
2.     for (int n=0; n<N; n++) {
3.         float acc = 0.0f;
4.         for (int k=0; k<K; k++) {
5.             acc += A[k*M + m] * B[n*K + k];
6.         }
7.         C[n*M + m] = acc;
8.     }
9. }
```

To keep things simple for now, we assume that the matrix sizes are nice multiples of 32, such that we can for example create OpenCL workgroups of size 32 by 32 without having to worry about boundary conditions. There will be assumptions along these lines in the next couple of pages, but that's mainly because we want to keep the code simple and the main story-line easy to follow.

<- previous next ->

Contact:

www.cedricnugteren.nl
web@cedricnugteren.nl

Tutorial: OpenCL SGEMM tuning for Kepler

Note: the complete source-code is available at GitHub.
Note: a tuned OpenCL BLAS library based on this tutorial is now available at GitHub.

<- previous next ->

Kernel 1: Naive implementation

Our first implementation is rather straightforward (and slow), but it gives us a good amount of parallelism and a nice starting point for further optimisations. We basically transform the two outer-loops over M and N (see the previous page) into two dimensions of parallel threads (named "work-items" in OpenCL).

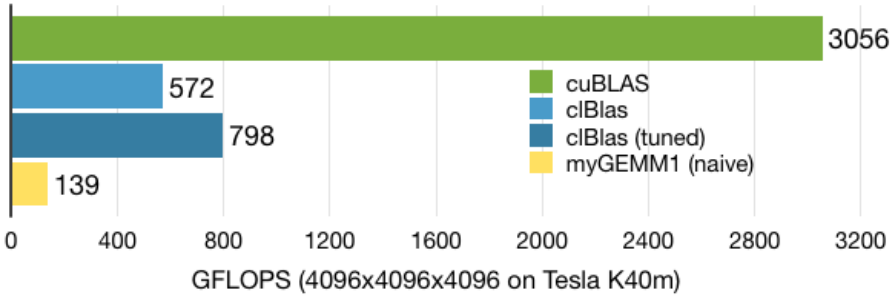
```
// First naive implementation
__kernel void myGEMM1(const int M, const int N, const int K,
                      const __global float* A,
                      const __global float* B,
                      __global float* C) {

    // Thread identifiers
    const int globalRow = get_global_id(0); // Row ID of C (0..M)
    const int globalCol = get_global_id(1); // Col ID of C (0..N)

    // Compute a single element (loop over K)
    float acc = 0.0f;
    for (int k=0; k<K; k++) {
        acc += A[k*M + globalRow] * B[globalCol*K + k];
    }

    // Store the result
    C[globalCol*M + globalRow] = acc;
}
```

As you can see, the indices m and n of the loops over M and N have been replaced with the thread-identifiers globalRow and globalCol. To make this work, we'll have to make sure that the GPU executes this code M * N times. We can do this in the host-code (which runs on the CPU). Assuming you have already initialised OpenCL, created the appropriate buffers and memory copies, create a queue, and compiled the kernel code (see GitHub for an example or use this MWE), this is how you launch the above kernel:

```
kernel = clCreateKernel(program, "myGEMM1", &err)
err = clSetKernelArg(kernel, 0, sizeof(int), (void*)&M);
err = clSetKernelArg(kernel, 1, sizeof(int), (void*)&N);
err = clSetKernelArg(kernel, 2, sizeof(int), (void*)&K);
err = clSetKernelArg(kernel, 3, sizeof(cl_mem), (void*)&A);
err = clSetKernelArg(kernel, 4, sizeof(cl_mem), (void*)&B);
err = clSetKernelArg(kernel, 5, sizeof(cl_mem), (void*)&C);
const int TS = 32;
const size_t local[2] = { TS, TS };
const size_t global[2] = { M, N };
err = clEnqueueNDRangeKernel(queue, kernel, 2, NULL,
                             global, local, 0, NULL, &event);
err = clWaitForEvents(1, &event);
```

The order of M and N (and thus globalRow and globalCol in the kernel) is actually not a random choice. In fact, assigning the parallelism over M to threads in the first dimension (like we did) gives us much better performance. The reason is that subsequent threads (in the first dimension) now access subsequent data-elements of matrices A and C, allowing memory requests to be "coalesced" (grouped together in more efficient bursts). Accesses to B are un-coalesced independent of the order of M and N, since each thread requires subsequent elements of B themselves (an entire column). But don't worry, we'll fix that in the next version.

As expected, performance is not too great at 139 GFLOPS. But hey, this was only our first version.



<- previous next ->

Last updated in 2014 - information might be outdated

Contact:

www.cedricnugteren.nl
web@cedricnugteren.nl

Tutorial: OpenCL SGEMM tuning for Kepler

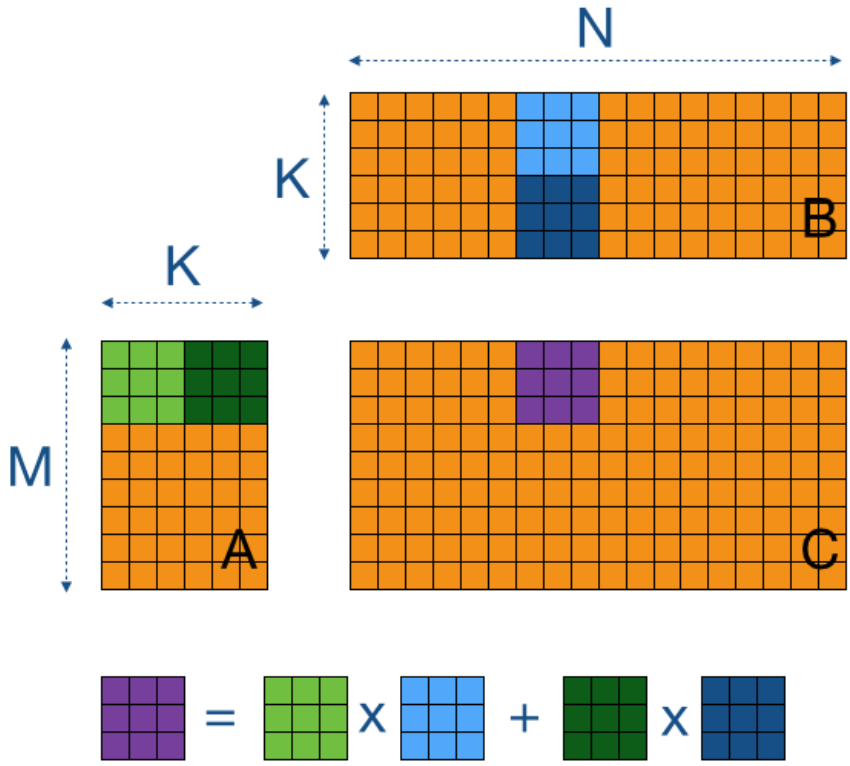
Note: the complete source-code is available at GitHub.
Note: a tuned OpenCL BLAS library based on this tutorial is now available at GitHub.

<- previous next ->

Kernel 2: Tiling in the local memory

The main reason the naive implementation doesn't perform so well is because we are accessing the GPU's off-chip memory way too much. Please count with me: to do the M*N*K multiplications and additions, we need M*N*K*2 loads and M*N stores. Since the multiplications and additions can actually be fused into a single hardware instruction (FMA), the computational intensity of the code is only 0.5 instructions per memory access (search for the "Roofline model" to find out why this is bad). Although the GPU's caches probably will help us out a bit, we can get much more performance by manually caching sub-blocks of the matrices (tiles) in the GPU's on-chip local memory (== shared memory in CUDA).

To compute a sub-block Csub of C (purple tile in the image below), we need A's corresponding rows (in green) and B's corresponding columns (in blue). Now, if we also divide A and B in sub-blocks Asub and Bsub, we can iteratively update the values in Csub by summing up the results of multiplications of Asub times Bsub.



Why is this useful? Well, if we take a closer look at the computation of a single element (in the image below), we see that there is lots of data re-use within a tile. For example, in the 3x3 tiles of the image below, all elements on the same row of the purple tile (Csub) are computed using the same data of the green tiles (Asub).

Let's translate this abstract image into actual OpenCL code. The trick here is to share the data of the Asub and Bsub tiles within a work-group (== threadblock in CUDA) via the local memory. To maximise the benefit of re-use, we'll make these tiles as large as possible.

To implement tiling, we'll leave our host code from the previous naive kernel intact. Note that it uses 2D work-groups of 32 by 32 (or TS x TS). This means that the purple Csub tile is also 32 by 32. For now, let's also take these dimensions for the Asub and Bsub tiles, we'll investigate rectangular tiles later on.

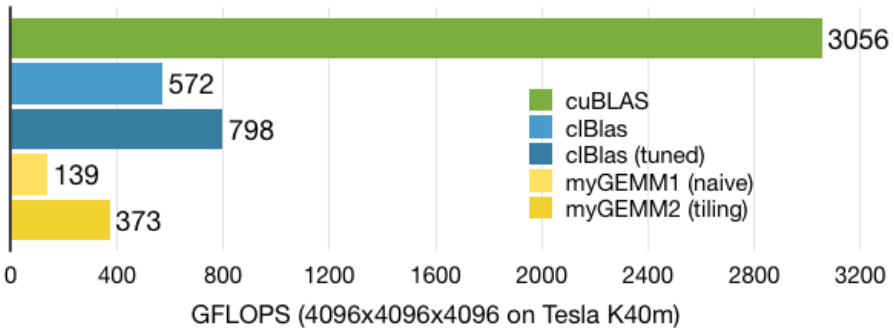
```
1. // Tiled and coalesced version
2. __kernel void myGEMM2(const int M, const int N, const int K,
3.                       const __global float* A,
4.                       const __global float* B,
```

```
                      __global float* C) {

    // Thread identifiers
    const int row = get_local_id(0); // Local row ID (max: TS)
    const int col = get_local_id(1); // Local col ID (max: TS)
    const int globalRow = TS*get_group_id(0) + row; // Row ID of C (0..M)
    const int globalCol = TS*get_group_id(1) + col; // Col ID of C (0..N)

    // Local memory to fit a tile of TS*TS elements of A and B
    __local float Asub[TS][TS];
    __local float Bsub[TS][TS];

    // Initialise the accumulation register
    float acc = 0.0f;

    // Loop over all tiles
    const int numTiles = K/TS;
    for (int t=0; t<numTiles; t++) {

        // Load one tile of A and B into local memory
        const int tiledRow = TS*t + row;
        const int tiledCol = TS*t + col;
        Asub[col][row] = A[tiledCol*M + globalRow];
        Bsub[col][row] = B[globalCol*K + tiledRow];

        // Synchronise to make sure the tile is loaded
        barrier(CLK_LOCAL_MEM_FENCE);

        // Perform the computation for a single tile
        for (int k=0; k<TS; k++) {
            acc += Asub[k][row] * Bsub[col][k];
        }

        // Synchronise before loading the next tile
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // Store the final result in C
    C[globalCol*M + globalRow] = acc;
}
```

If we take a look at the code, we see that the original accumulation loop over K has been split into two new loops: one over all K/TS tiles and one over all TS elements within a tile. Within this loop over the tiles, we can identify two parts which are separated with synchronisation barriers: (1) loading from off-chip memory to local memory, and (2) computation based on local memory data. Note that each thread now performs only two global loads per tile (one element of A and B), whereas this used to be two times the tile-size (a row of A and a column of B). This gives us a reduction of a factor 32 in off-chip memory accesses!

There is one other thing to notice: since we are now loading tiles instead of columns from B, and since we are now sharing them within a work-group, we can replace un-coalesced memory accesses by coalesced counter-parts by loading them per row.

When we look at performance, we see a great improvement over the naive implementation. However, with ~370 GFLOPS we are still not at the level of clBLAS, so there is much more work to!



<- previous

next ->

Tutorial written by Cedric Nugteren, (c) 2014 SURFsara
Last updated in 2014 - information might be outdated

Contact:

www.cedricnugteren.nl
web@cedricnugteren.nl

Tutorial: OpenCL SGEMM tuning for Kepler

Note: the complete source-code is available at GitHub.
Note: a tuned OpenCL BLAS library based on this tutorial is now available at GitHub.

<- previous next ->

Kernel 3: More work per thread

One of the ways in which the previous kernel can be improved is by increasing the amount of work of each thread (and thus reducing the total number of threads). Why is this useful? Well, let's take a moment and inspect the PTX assembly that was generated for the previous kernel. In particular, let's look at the main computational part: the (unrolled) inner k-loop. Below are 2 of the TS iterations:

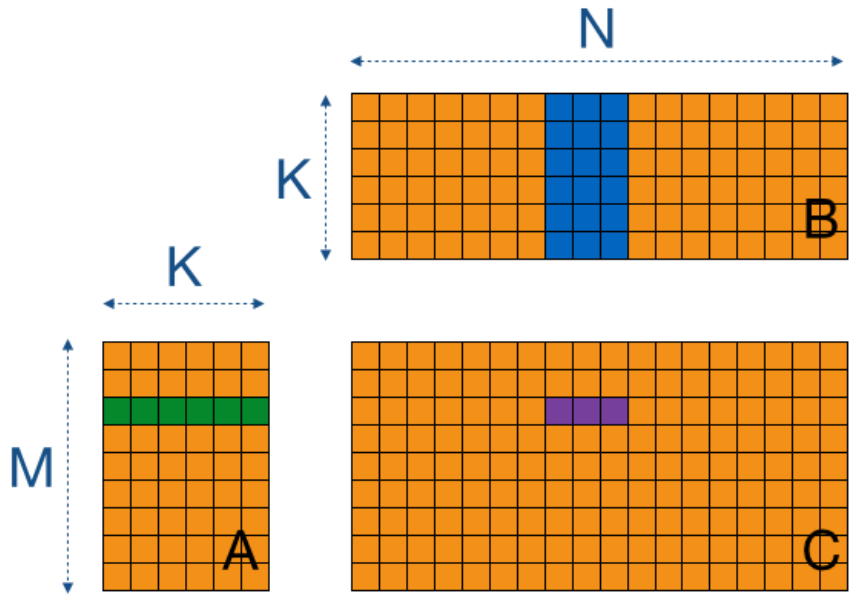
```
1. ld.shared.f32   %f50, [%r18+56];
2. ld.shared.f32   %f51, [%r17+1792];
3. fma.rn.f32  %f52, %f51, %f50, %f49;
4. ld.shared.f32   %f53, [%r18+60];
5. ld.shared.f32   %f54, [%r17+1920];
6. fma.rn.f32  %f55, %f54, %f53, %f52;
```

In each iteration of the k-loop, we see these three instructions:

1. The loading of an Asub element from the local memory into the register file (ld.shared.f32).
2. The loading of a Bsub element from the local memory into the register file (ld.shared.f32).
3. The actual computation in the form of a fused multiply add (fma.rn.f32).

Or, in simpler terms, only one of out three instructions are actual (useful) computations! Quite a waste of our precious GFLOPS, don't you think?

We can apply the same trick we did before with the tiling, but now at register-level. If we let one thread compute 8 elements arranged in consecutive columns of C, we can reduce accesses to A by a factor 8 (let's name this constant "WPT"). In this case, this reduction is not in terms of off-chip memory accesses (given that we keep our tiles at constant size), but in a reduced amount of local memory accesses. This is illustrated by the image below in which we (for simplicity) don't consider the tiling and use a WPT factor of 3.



Let's take a look at the new kernel, which brings the following changes:

- A factor of WPT registers are initialised to zero for each thread (lines 17 - 21).
- Each thread now loads WPT values of A and B into the local memory (lines 27 - 33). The tile-size is still the same, as is the amount of local memory used per work-group.
- Each thread performs WPT times TS accumulations per tile (lines 38 - 43). Note that the inner-most loop over WPT doesn't require a new value from Asub each time, saving precious local memory loads.
- Each thread stores WPT values in the resulting C matrix (lines 49 - 52).

```
1.  // Increased the amount of work-per-thread by a factor WPT
2.  __kernel void myGEMM3(const int M, const int N, const int K,
3.                        const __global float* A,
4.                        const __global float* B,
5.                        __global float* C) {
6.
7.      // Thread identifiers
8.      const int row = get_local_id(0); // Local row ID (max: TS)
9.      const int col = get_local_id(1); // Local col ID (max: TS/WPT == RTS)
10.     const int globalRow = TS*get_group_id(0) + row; // Row ID of C (0..M)
```

```
    const int globalCol = TS*get_group_id(1) + col; // Col ID of C (0..N)

    // Local memory to fit a tile of TS*TS elements of A and B
    __local float Asub[TS][TS];
    __local float Bsub[TS][TS];

    // Initialise the accumulation registers
    float acc[WPT];
    for (int w=0; w<WPT; w++) {
        acc[w] = 0.0f;
    }

    // Loop over all tiles
    const int numTiles = K/TS;
    for (int t=0; t<numTiles; t++) {

        // Load one tile of A and B into local memory
        for (int w=0; w<WPT; w++) {
            const int tiledRow = TS*t + row;
            const int tiledCol = TS*t + col;
            Asub[col + w*RTS][row] = A[(tiledCol + w*RTS)*M + globalRow];
            Bsub[col + w*RTS][row] = B[(globalCol + w*RTS)*K + tiledRow];
        }

        // Synchronise to make sure the tile is loaded
        barrier(CLK_LOCAL_MEM_FENCE);

        // Perform the computation for a single tile
        for (int k=0; k<TS; k++) {
            for (int w=0; w<WPT; w++) {
                acc[w] += Asub[k][row] * Bsub[col + w*RTS][k];
            }
        }

        // Synchronise before loading the next tile
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // Store the final results in C
    for (int w=0; w<WPT; w++) {
        C[(globalCol + w*RTS)*M + globalRow] = acc[w];
    }
}
```

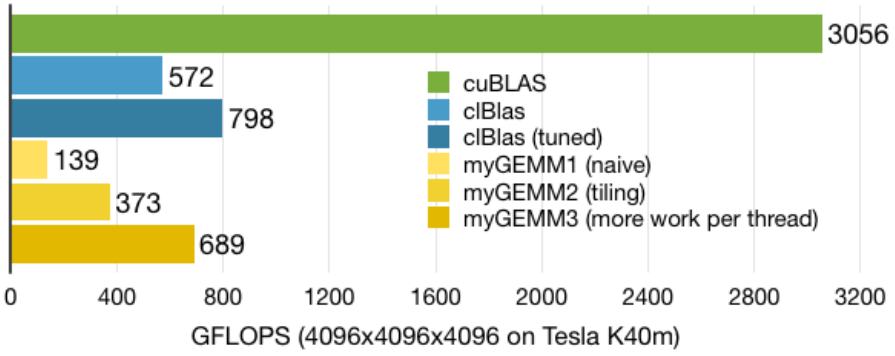
Since we increased the amount of work per thread, let's not forget to reduce the total amount of threads before we launch our kernel:

```
const size_t local[2] = { TS, TS/WPT };
const size_t global[2] = { M, N/WPT };
```

When we look at the new PTX assembly, we can see that we have greatly increased the fraction of FMA-instructions. For example, below are 8 iterations of the computation-loop (instead of the 2 from before), which show that we require only 8+1 loads from the local memory for 8 FMAs (instead of 8+8):

```
ld.shared.f32   %f82, [%r101+4];
ld.shared.f32   %f83, [%r102];
fma.rn.f32  %f91, %f83, %f82, %f67;
ld.shared.f32   %f84, [%r101+516];
fma.rn.f32  %f92, %f83, %f84, %f69;
ld.shared.f32   %f85, [%r101+1028];
fma.rn.f32  %f93, %f83, %f85, %f71;
ld.shared.f32   %f86, [%r101+1540];
fma.rn.f32  %f94, %f83, %f86, %f73;
ld.shared.f32   %f87, [%r101+2052];
fma.rn.f32  %f95, %f83, %f87, %f75;
ld.shared.f32   %f88, [%r101+2564];
fma.rn.f32  %f96, %f83, %f88, %f77;
ld.shared.f32   %f89, [%r101+3076];
fma.rn.f32  %f97, %f83, %f89, %f79;
ld.shared.f32   %f90, [%r101+3588];
fma.rn.f32  %f98, %f83, %f90, %f81;
```

When we look at performance of our new kernel, it turns out we are already ahead of clBlas! Still, there are a couple of other tricks we can do to get more performance, as evidenced by the gap with cuBLAS.



<- previous next ->

Tutorial written by Cedric Nugteren, (c) 2014 SURFsara
Last updated in 2014 - information might be outdated

Contact:

www.cedricnugteren.nl
web@cedricnugteren.nl

Tutorial: OpenCL SGEMM tuning for Kepler

Note: the complete source-code is available at GitHub.
Note: a tuned OpenCL BLAS library based on this tutorial is now available at GitHub.

<- previous next ->

Kernel 4: Wider data-types

In the previous kernel we increased the amount of work in the column-dimension of C. Obviously, we could have also done this in the row-dimension (or in both, but we'll explore this later). Although this has the same advantage of reducing pressure on the local memory, it can have another advantage: wider data-types. Why is this? Well, increasing the work per thread in the row-dimension of C can also be done by considering vector data-types instead of loops over WPT. The NVIDIA GPUs don't support vector operations (such as multiply or add) in hardware, but they do have special wider load and store instructions both for the off-chip and the local memory. Let's see if using them gives us better performance.

First, we set the launch-parameters of our new kernel considering that we'll use data-types of width 'WIDTH':

```
1. const size_t local[2] = { TS/WIDTH, TS };
2. const size_t global[2] = { M/WIDTH, N };
```

OpenCL already defines floating-point vector types, which makes using them quite convenient. And we also don't have to transform our matrices into vector types: we can simply cast the pointers in our kernel. To remain flexible, we'll create a new data-type to be able to configure the WIDTH parameter with a pre-processor define:

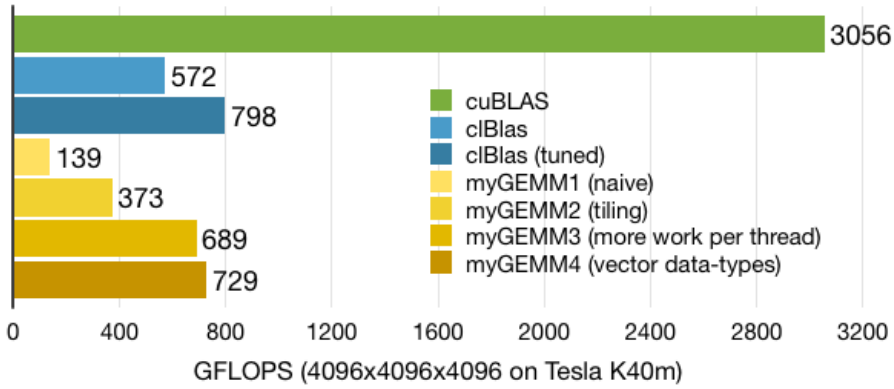
```
1. #define WIDTH 4
2. #if WIDTH == 1
3.     typedef float floatX;
4. #elif WIDTH == 2
5.     typedef float2 floatX;
6. #elif WIDTH == 4
7.     typedef float4 floatX;
8. #endif
```

The new kernel is given below. We have left out support for the float8 data-type to make the code more readable, although WIDTH is actually set to 8 for our benchmark, as this gives the best performance. The kernel is based on the same concept that we used for the previous kernel, but there are a few changes: The increased amount of work per thread is now found in dimension 0 (rows of C) rather than 1 (columns of C). Since the data-types are now vectors, all indices have to be divided by a factor WIDTH. The inner-loop computation has to be written out, as it is not a vector FMA operation. An alternative solution is to use a scalar pointer to local memory and a regular loop.

```
1. // Use wider data types
2. __kernel void myGEMM4(const int M, const int N, const int K,
3.                       const __global floatX* A,
4.                       const __global floatX* B,
5.                       __global floatX* C) {
6.
7.     // Thread identifiers
8.     const int row = get_local_id(0); // Local row ID (max: TS/WIDTH)
9.     const int col = get_local_id(1); // Local col ID (max: TS)
10.    const int globalRow = (TS/WIDTH)*get_group_id(0) + row; // 0..M/WIDTH
11.    const int globalCol = TS*get_group_id(1) + col; // 0..N
12.
13.    // Local memory to fit a tile of TS*TS elements of A and B
14.    __local floatX Asub[TS][TS/WIDTH];
15.    __local floatX Bsub[TS][TS/WIDTH];
16.
17.    // Initialise the accumulation registers
18.    #if WIDTH == 1
19.        floatX acc = 0.0f;
20.    #elif WIDTH == 2
21.        floatX acc = { 0.0f, 0.0f };
22.    #elif WIDTH == 4
23.        floatX acc = { 0.0f, 0.0f, 0.0f, 0.0f };
24.    #endif
25.
26.    // Loop over all tiles
27.    const int numTiles = K/TS;
28.    for (int t=0; t<numTiles; t++) {
29.
30.        // Load one tile of A and B into local memory
31.        const int tiledRow = (TS/WIDTH)*tile + row;
32.        const int tiledCol = TS*tile + col;
33.        Asub[col][row] = A[tiledCol*(M/WIDTH) + globalRow];
34.        Bsub[col][row] = B[globalCol*(K/WIDTH) + tiledRow];
35.
36.        // Synchronise to make sure the tile is loaded
37.        barrier(CLK_LOCAL_MEM_FENCE);
38.
39.        // Perform the computation for a single tile
40.        floatX vecA, vecB;
41.        float valB;
42.        for (int k=0; k<TS/WIDTH; k++) {
43.            vecB = Bsub[col][k];
44.            for (int w=0; w<WIDTH; w++) {
45.                vecA = Asub[WIDTH*k + w][row];
46.                #if WIDTH == 1
47.                    valB = vecB;
48.                    acc += vecA * valB;
49.                #elif WIDTH == 2
```

```
                    switch (w) {
                        case 0: valB = vecB.x; break;
                        case 1: valB = vecB.y; break;
                    }
                    acc.x += vecA.x * valB;
                    acc.y += vecA.y * valB;
                #elif WIDTH == 4
                    switch (w) {
                        case 0: valB = vecB.x; break;
                        case 1: valB = vecB.y; break;
                        case 2: valB = vecB.z; break;
                        case 3: valB = vecB.w; break;
                    }
                    acc.x += vecA.x * valB;
                    acc.y += vecA.y * valB;
                    acc.z += vecA.z * valB;
                    acc.w += vecA.w * valB;
                #endif
            }
        }

        // Synchronise before loading the next tile
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // Store the final results in C
    C[globalCol*(M/WIDTH) + globalRow] = acc;
}
```

This kernel performs a bit better than the previous kernel due to its wider data-types and thus fewer load/store instructions. But the real benefit will be apparent later on when we will combine these two techniques.



<- previous next ->

Tutorial written by Cedric Nugteren, (c) 2014 SURFsara
Last updated in 2014 - information might be outdated

Contact:

www.cedricnugteren.nl
web@cedricnugteren.nl

Tutorial: OpenCL SGEMM tuning for Kepler

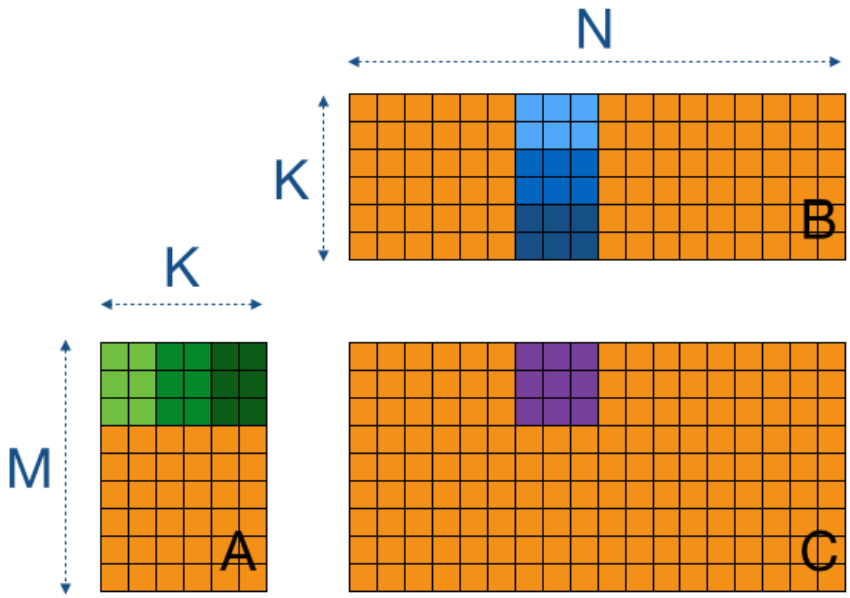
Note: the complete source-code is available at GitHub.
Note: a tuned OpenCL BLAS library based on this tutorial is now available at GitHub.

<- previous next ->

Kernel 5: Transposed input matrix and rectangular tiles

Our first tiled version showed that a large tile size can greatly reduce off-chip memory accesses and can thus improve performance. However, if we want to go to even larger tiles (say 64 by 64) we'll run out of resources. The Tesla K40 GPU which we use has 48KB of local memory per SM, on which multiple work-groups can run. For a 32 by 32 tile, this means we consume 2*32*32*4 = 8KB per work-group, so there is some headroom left.

To get more flexibility and to be able to fine-tune the parameters, we'll want to go for rectangular tiles. Since both matrix A and B share the dimension K, we can actually create rectangular tiles, as is shown in the image below. In the example picture, we'll have to iterate over 3 tiles of 3x2 each, while we are still computing a 3x3 tile in the end.



To get this idea implemented, we'll want to transpose one of the input matrices before starting the matrix-multiplication. This will save us a lot of trouble computing indices, as the K-sized dimension (which A and B share) will be the same dimension. We choose to transpose the B matrix.

The transpose kernel itself is actually quite trivial and is negligible in terms of computational cost compared to the main matrix-multiplication kernel. Here is the transpose kernel which we used:

```
// Simple transpose kernel for a P * Q matrix
__kernel void transpose(const int P, const int Q,
                        const __global float* input,
                        __global float* output) {

    // Thread identifiers
    const int tx = get_local_id(0);
    const int ty = get_local_id(1);
    const int ID0 = get_group_id(0)*TRANSPOSEX + tx; // 0..P
    const int ID1 = get_group_id(1)*TRANSPOSEY + ty; // 0..Q

    // Set-up the local memory for shuffling
    __local float buffer[TRANSPOSEX][TRANSPOSEY];

    // Swap the x and y coordinates to perform the rotation (coalesced)
    if (ID0 < P && ID1 < Q) {
        buffer[ty][tx] = input[ID1*P + ID0];
    }

    // Synchronise all threads
    barrier(CLK_LOCAL_MEM_FENCE);

    // We don't have to swap the x and y thread indices here,
    // because that's already done in the local memory
    const int newID0 = get_group_id(1)*TRANSPOSEY + tx;
    const int newID1 = get_group_id(0)*TRANSPOSEX + ty;

    // Store the transposed result (coalesced)
    if (newID0 < Q && newID1 < P) {
        output[newID1*Q + newID0] = buffer[tx][ty];
    }
}
```

Now, let's see what the new matrix-multiplication kernel looks like. We'll take our earlier kernel with an increased work per thread count (WPT) as a starting point. First, let us define some new constants:

```
#define TSM 64                  // The tile-size in dimension M
#define TSN 64                  // The tile-size in dimension N
#define TSK 32                  // The tile-size in dimension K
#define WPTN 8                  // The work-per-thread in dimension N
#define RTSN (TSN/WPTN)         // The reduced tile-size in dimension N
#define LPT ((TSK*TSM)/(RTSM*RTSN)) // The loads-per-thread for a tile
```

Note that, although we do define separate tile-sizes in the M and N dimensions, we assume for now they are equal. As before, we use the WPT setting to reduce the amount of threads in the second thread dimension. Our launch-parameters are:

```
const size_t local[2] = { TSM, TSN/WPTN }; // Or { TSM, RTSN };
const size_t global[2] = { M, N/WPTN };
```

Our kernel looks a lot like our 3rd version. The most notable differences are:

- Since matrix B is transposed, reading the inputs A and B is now very similar. The main difference between the two is that outside of a work-group, work-group IDs determine which tile to load.
- Because matrix B is transposed, we'll have to un-transpose data in the local memory. There are two options:
    1. We could access the Bsub matrix in the inner computational loop as [k,col] instead of [col,k]. However, that would mean that subsequent iterations of the k-loop no longer require subsequent data from the local memory. Is this bad? Actually, yes, since the compiler can replace 2 32-bit loads from the local memory (LDS) with one 64-bit load (LDS.64) as long as data is well positioned and aligned. These 64-bit (or 128-bit) loads not only save instructions, but are also the only way to benefit from the full bandwidth of the GPU's local memory (and we might actually saturate the bandwidth!).
    2. Since we don't want to change the access pattern to the local memory (see point 1), we'll have to change the way data is stored into the local memory (otherwise results will simply be incorrect). This can be done by changing the original [col,row] pattern (as still is used for matrix A) into a [row,col] storing pattern (see the new code below). This has a new drawback: bank conflicts. We'll discuss that later.
- The constant TS has been replaced with either TSK, TSM, or TSN. Special care has to be taken with respect to the amount of loads per tile, as this is no longer equal to the WPT setting. In fact, the tile-size changes with TSK, whereas the number of threads in a work-group changes with TSM and TSN. Note that the code poses some constraints on the settings, for example LPT has to be an integer. If not, code will grow complicated very quickly and performance will drop due to branches.

```
// Pre-transpose the input matrix B and use rectangular tiles
__kernel void myGEMM5(const int M, const int N, const int K,
                      const __global float* A,
                      const __global float* B,
                      __global float* C) {

    // Thread identifiers
    const int row = get_local_id(0); // Local row ID (max: TSM)
    const int col = get_local_id(1); // Local col ID (max: TSN/WPTN)
    const int globalRow = TSM*get_group_id(0) + row; // 0..M
    const int globalCol = TSN*get_group_id(1) + col; // 0..N

    // Local memory to fit a tile of A and B
    __local float Asub[TSK][TSM];
    __local float Bsub[TSN][TSK];

    // Initialise the accumulation registers
    float acc[WPTN];
    for (int w=0; w<WPTN; w++) {
        acc[w] = 0.0f;
    }

    // Loop over all tiles
    int numTiles = K/TSK;
    for (int t=0; t<numTiles; t++) {

        // Load one tile of A and B into local memory
        for (int l=0; l<LPT; l++) {
            int tiledIndex = TSK*t + col + l*RTSN;
            int indexA = tiledIndex*M + TSM*get_group_id(0) + row;
            int indexB = tiledIndex*N + TSN*get_group_id(1) + row;
            Asub[col + l*RTSN][row] = A[indexA];
            Bsub[row][col + l*RTSN] = B[indexB];
        }

        // Synchronise to make sure the tile is loaded
        barrier(CLK_LOCAL_MEM_FENCE);

        // Perform the computation for a single tile
        for (int k=0; k<TSK; k++) {
            for (int w=0; w<WPTN; w++) {
                acc[w] += Asub[k][row] * Bsub[col + w*RTSN][k];
            }
        }

        // Synchronise before loading the next tile
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // Store the final results in C
    for (int w=0; w<WPTN; w++) {
        C[(globalCol + w*RTSN)*M + globalRow] = acc[w];
    }
}
```

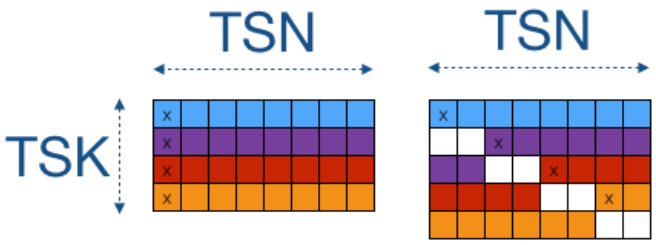
Using the same settings as before (squared tiles of 32 by 32 and a WPT of 8) we achieve only 460 GFLOPS, less than the 689 of the earlier kernel. The decreased performance is caused by bank-conflicts in the local memory in the following line:

```
Bsub[row][col + w*RTSN] = B[indexB];
```

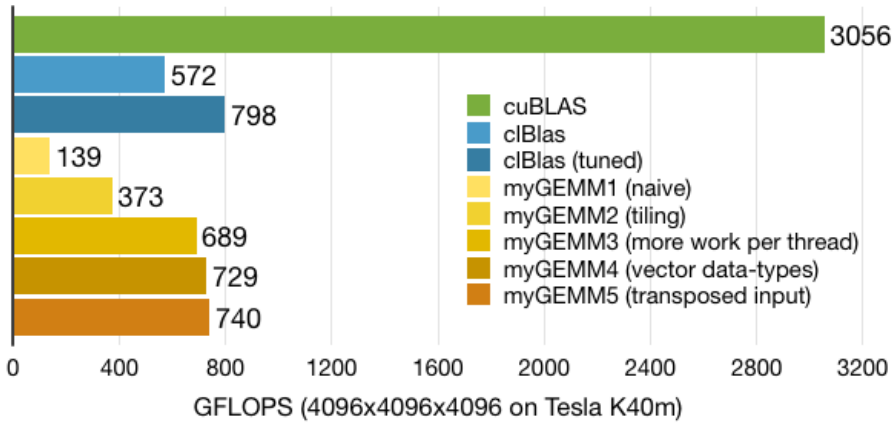
The bank-conflicts arise because threads within a warp (same 'col' value, but different 'row' value) access the same (or many of the same) local memory banks. The image below on the left visualises this: threads in a warp (4 for this toy-example) access data in a column-wise fashion. For example, the first warp will access all elements marked with an 'x'. On our Kepler GPU, local memory is arranged in 32 banks. If TSN is equal to 32, then every column in the picture represents a single bank (let's not consider the different addressing modes for simplicity). To repair this we have to make sure that threads in a warp do not access the same column. A well-known technique is to use padding:

```
__local float Bsub[TSN][TSK+2];
```

Padding by 2 will result in the picture on the right, in which we can see that bank conflicts no longer occur (or at least a lot less than before). Note that we pad the memory by 2 rather than 1 to align data to 64-bit (two floats) so that we can still benefit from 64-bit loads from local memory. You probably noted that the padding does consume extra local memory, so this technique might deteriorate performance if occupancy is reduced and latency can no longer be hidden.



Now that we have all of this working, we have the freedom to select 64 by 32 blocks for example. This gives us a little bit of extra performance over the earlier kernel's 32 by 32 blocks, giving us 740 GFLOPS.



<- previous next ->

Tutorial written by Cedric Nugteren, (c) 2014 SURFsara
Last updated in 2014 - information might be outdated

Contact:

www.cedricnugteren.nl
web@cedricnugteren.nl

Tutorial: OpenCL SGEMM tuning for Kepler

Note: the complete source-code is available at GitHub.
Note: a tuned OpenCL BLAS library based on this tutorial is now available at GitHub.

<- previous next ->

Kernel 6: 2D register blocking

In one of our earlier kernels we increased the work per thread by a factor of 8. The motivation was that this would require less accesses to local memory, yielding a higher percentage of FMA instructions in the main computational loop. Let's take this idea one step further: we'll increase the work per thread in both dimensions to form so-called 2D 'register blocking'. This trick is very similar to what we've done for 2D tiling, but at a different memory level (from local memory to registers rather than from off-chip to local memory).

To get this working, we'll have to make a few changes. Let's start off with some fresh defines:

```
#define TSM 128                 // The tile-size in dimension M
#define TSN 128                 // The tile-size in dimension N
#define TSK 16                  // The tile-size in dimension K
#define WPTM 8                  // The work-per-thread in dimension M
#define WPTN 8                  // The work-per-thread in dimension N
#define RTSM (TSM/WPTM)         // The reduced tile-size in dimension M
#define RTSN (TSN/WPTN)         // The reduced tile-size in dimension N
#define LPTA ((TSK*TSM)/(RTSM*RTSN)) // Loads-per-thread for A
#define LPTB ((TSK*TSN)/(RTSM*RTSN)) // Loads-per-thread for B
```

We now have two work-per-thread variables, WPTM and WPTN. Both are set to 8, giving us work-groups of 16 by 16 (RTSM by RTSN) for tile-sizes in the M and N dimension of 128. We still consider TSM and TSN to be equal, making loading data a bit more concise since LPTA is equal to LPTB in that case.

We'll use the new WPTM and WPTN constants to compute the launch-parameters:

```
const size_t local[2] = { TSM/WPTM, TSN/WPTN }; // Or { RTSM, RTSN };
const size_t global[2] = { M/WPTM, N/WPTN };
```

The new kernel takes its basis from the previous kernel (myGEMM5). The following changes are made:

- The accumulation registers are now 2D: they are initialised using a double-loop. The results are also stored to the C matrix using a similar double-loop.
- Loading data has become even more complicated (but its worth it). We now put all the threads (in first and second dimension) onto one big pile, and add a loop over the amount of loads per threads (LPTA = LPTB) on top of that. This gives us the variable 'id', which we split by modulo and integer division to obtain the row and column IDs.
- In previous versions of the kernel we could cache a single value of Asub into a register and re-use it in the loop of WPT. Now, we make this explicit (using Areg) and do a similar trick for Bsub. However, for Breg we do need WPTN temporary registers, as its re-use is not in the inner-most loop.

```
// Use 2D register blocking (further increase in work per thread)
__kernel void myGEMM6(const int M, const int N, const int K,
                      const __global float* A,
                      const __global float* B,
                      __global float* C) {

    // Thread identifiers
    const int tidm = get_local_id(0); // Local row ID (max: TSM/WPTM)
    const int tidn = get_local_id(1); // Local col ID (max: TSN/WPTN)
    const int offsetM = TSM*get_group_id(0); // Work-group offset
    const int offsetN = TSN*get_group_id(1); // Work-group offset

    // Local memory to fit a tile of A and B
    __local float Asub[TSK][TSM];
    __local float Bsub[TSN][TSK+2];

    // Allocate register space
    float Areg;
    float Breg[WPTN];
    float acc[WPTM][WPTN];

    // Initialise the accumulation registers
    for (int wm=0; wm<WPTM; wm++) {
        for (int wn=0; wn<WPTN; wn++) {
            acc[wm][wn] = 0.0f;
        }
    }

    // Loop over all tiles
    int numTiles = K/TSK;
    for (int t=0; t<numTiles; t++) {

        // Load one tile of A and B into local memory
        for (int la=0; la<LPTA; la++) {
            int tid = tidn*RTSM + tidm;
            int id = la*RTSN*RTSM + tid;
            int row = id % TSM;
            int col = id / TSM;
            int tiledIndex = TSK*t + col;
            Asub[col][row] = A[tiledIndex*M + offsetM + row];
            Bsub[row][col] = B[tiledIndex*N + offsetN + row];
        }

```

```
44.         // Synchronise to make sure the tile is loaded
45.         barrier(CLK_LOCAL_MEM_FENCE);
46.
47.         // Loop over the values of a single tile
48.         for (int k=0; k<TSK; k++) {
49.
50.             // Cache the values of Bsub in registers
51.             for (int wn=0; wn<WPTN; wn++) {
52.                 int col = tidn + wn*RTSN;
53.                 Breg[wn] = Bsub[col][k];
54.             }
55.
56.             // Perform the computation
57.             for (int wm=0; wm<WPTM; wm++) {
58.                 int row = tidm + wm*RTSM;
59.                 Areg = Asub[k][row];
60.                 for (int wn=0; wn<WPTN; wn++) {
61.                     acc[wm][wn] += Areg * Breg[wn];
62.                 }
63.             }
64.         }
65.
66.         // Synchronise before loading the next tile
67.         barrier(CLK_LOCAL_MEM_FENCE);
68.     }
69.
70.     // Store the final results in C
71.     for (int wm=0; wm<WPTM; wm++) {
72.         int globalRow = offsetM + tidm + wm*RTSM;
73.         for (int wn=0; wn<WPTN; wn++) {
74.             int globalCol = offsetN + tidn + wn*RTSN;
75.             C[globalCol*M + globalRow] = acc[wm][wn];
76.         }
77.     }
78. }
```

This new kernel is quite resource consuming and its performance can fluctuate quite a bit. Let's take a look at local memory usage first:

local_memory_bytes = 4 * TSK * TSM + 4 * (TSK + 2) * TSN

With the current settings (TSK = 16, TSM = TSN = 128), we use 16KB plus 1KB padding. With a total of 48KB local memory available, our best-case scenario is to get two work-groups running on a single SM at a time. With a work-group size of 256 (16 by 16) we'll have a total of 512 active threads to hide latencies.

But do we actually get to run 512 threads? The Kepler GPU has 64K 32-bit registers per SM, which gives us 128 registers per thread if we divide them over 512 threads. With our register-tiling technique, we use 8 by 8 accumulation registers, another 8 for B, and 1 for A. This gives us a lower limit of 73 registers, well below 128. Unfortunately, the compiler will use a lot more registers, making this 128 limit a real constraint. Some of these extra registers are used because of 'optimisations' to reduce the amount of instructions. For example, let's take a look at the following snippet from our kernel:

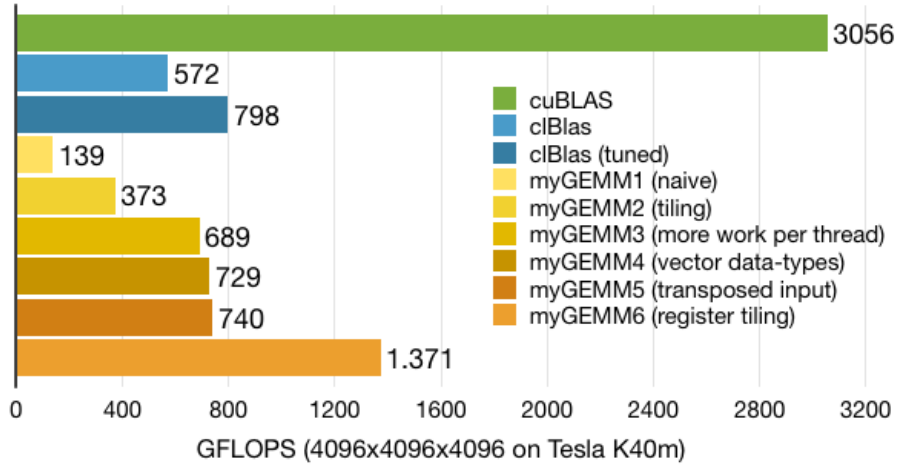
```
1.     // Loop over all tiles
2.     int numTiles = K/TSK;
3.     for (int t=0; t<numTiles; t++) {
4.
5.         // Load one tile of A and B into local memory
6.         for (int la=0; la<LPTA; la++) {
7.             int tid = tidn*RTSM + tidm;
8.             int id = la*RTSN*RTSM + tid;
9.             int row = id % TSM;
10.            int col = id / TSM;
11.            (...)
```

The compiler will rightly conclude that each tile (iterations of the t-loop) requires the same index computations for 'row' and 'col' (based on 'id' and 'tid'). Therefore, to reduce the amount of instructions executed, the compiler will compute all LPTA values of row and col once before the tile-loop starts. Although this does save instructions, it also consumes 2*LPTA registers! Luckily we can tell the compiler to re-compute the values every time with the volatile-keyword:

```
volatile int id = la*RTSN*RTSM + tid;
```

For our kernel, running 256 or 512 active threads makes quite a difference. This makes our code very fragile and sensitive to compiler or small code changes (such as the volatile keyword above). Even compiler-options such as -cl-nv-maxrregcount=127 do not always help, as the compiler might decide to spill the extra registers it 'needs' to memory.

The best-case performance that we get from this kernel is well over 1.3 TFLOPS, more than twice the clBlas score! However, changing small things in the code (like using the volatile keyword or unrolling a particular loop) can bring us anywhere between 800 and 1300 GFLOPS. All in all, a significant improvement over previous kernels.



<- previous next ->

Tutorial written by Cedric Nugteren, (c) 2014 SURFsara
Last updated in 2014 - information might be outdated

Contact:

www.cedricnugteren.nl
web@cedricnugteren.nl

Tutorial: OpenCL SGEMM tuning for Kepler

Note: the complete source-code is available at GitHub.
Note: a tuned OpenCL BLAS library based on this tutorial is now available at GitHub.

<- previous next ->

Kernel 7: Wider loads with register blocking

In one of our earlier kernels we used wider data-types to get 64-bit or 128-bit loads and stores. Let's re-integrate that in our latest version with 2D register blocking. We'll do this for loads only.

The launch configuration and the constants remain the same. The kernel is also quite similar, with the main exceptions in the part where we are loading from off-chip memory:

- We now load float2 or float4 values from off-chip memory. Therefore, the loop over the amount of loads per thread has been reduced by a factor WIDTH. The same holds for the indexing into the vector arrays.
- Since we now have a slightly different access pattern when storing data into Asub and Bsub, we choose to swap the column and row indices of Bsub back to how they are for Asub. This does require more local memory loads (LDS instead of LDS.64) in the computational loop, but since this is now cached using register tiling, its impact is not that big. The main new advantage is that we can now also store data into Bsub using vector data-types. Note that since Asub and Bsub are not declared as vectors, the stores into Asub and Bsub look scalar from a coding-perspective, but they become 64-bit or 128-bit stores (STS.128) when compiled. We can now also remove the padding, giving us that extra bit of local memory back.

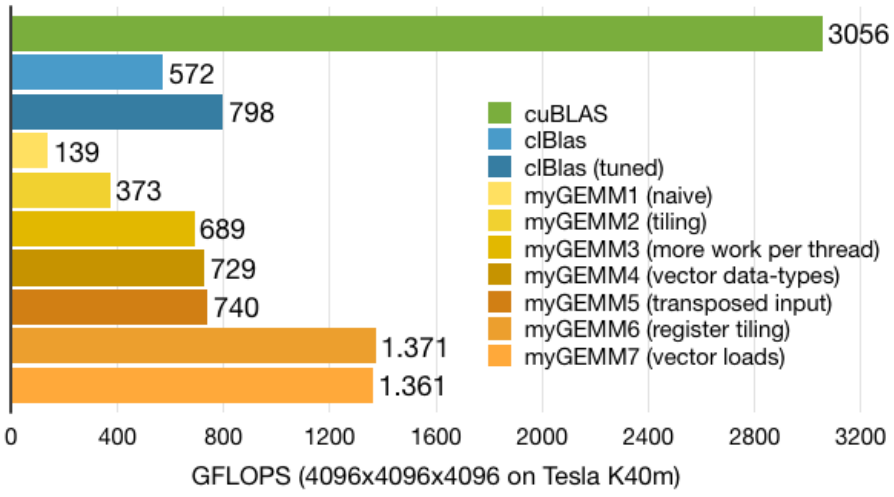
Note that we still have the restriction that TSM has to be equal to TSN. This restriction can be removed by creating two loops when loading, one over LPTA with indices constrained to TSM and one over LPTB with indices constrained to TSN. We'll implement this in our final version.

```
// Wider loads combined with 2D register blocking
__kernel void myGEMM7(const int M, const int N, const int K,
                      const __global floatX* A,
                      const __global floatX* B,
                      __global float* C) {

    // Thread identifiers
    const int tidm = get_local_id(0); // Local row ID (max: TSM/WPTM)
    const int tidn = get_local_id(1); // Local col ID (max: TSN/WPTN)
    const int offsetM = TSM*get_group_id(0); // Work-group offset
    const int offsetN = TSN*get_group_id(1); // Work-group offset

    // Local memory to fit a tile of A and B
    __local float Asub[TSK][TSM];
    __local float Bsub[TSK][TSN];

    // Allocate register space
    float Areg;
    float Breg[WPTN];
    float acc[WPTM][WPTN];

    // Initialise the accumulation registers
    for (int wm=0; wm<WPTM; wm++) {
        for (int wn=0; wn<WPTN; wn++) {
            acc[wm][wn] = 0.0f;
        }
    }

    // Loop over all tiles
    int numTiles = K/TSK;
    for (int t=0; t<numTiles; t++) {

        // Load one tile of A and B into local memory
        for (int la=0; la<LPTA/WIDTH; la++) {
            int tid = tidn*RTSM + tidm;
            int id = la*RTSN*RTSM + tid;
            int row = id % (TSM/WIDTH);
            int col = id / (TSM/WIDTH);

            // Load the values (wide vector load)
            int tiledIndex = TSK*t + col;
            floatX vecA = A[tiledIndex*(M/WIDTH) + offsetM/WIDTH + row];
            floatX vecB = B[tiledIndex*(N/WIDTH) + offsetN/WIDTH + row];

            // Store the loaded vectors into local memory
            #if WIDTH == 1
                Asub[col][row] = vecA;
                Asub[col][row] = vecA;
            #elif WIDTH == 2
                Asub[col][WIDTH*row + 0] = vecA.x;
                Asub[col][WIDTH*row + 1] = vecA.y;
            #elif WIDTH == 4
                Asub[col][WIDTH*row + 0] = vecA.x;
                Asub[col][WIDTH*row + 1] = vecA.y;
                Asub[col][WIDTH*row + 2] = vecA.z;
                Asub[col][WIDTH*row + 3] = vecA.w;
            #endif
            #if WIDTH == 1
                Bsub[col][row] = vecB;
                Bsub[col][row] = vecB;
            #elif WIDTH == 2
                Bsub[col][WIDTH*row + 0] = vecB.x;
                Bsub[col][WIDTH*row + 1] = vecB.y;
            #elif WIDTH == 4
```

```
                Bsub[col][WIDTH*row + 0] = vecB.x;
                Bsub[col][WIDTH*row + 1] = vecB.y;
                Bsub[col][WIDTH*row + 2] = vecB.z;
                Bsub[col][WIDTH*row + 3] = vecB.w;
            #endif
        }

        // Synchronise to make sure the tile is loaded
        barrier(CLK_LOCAL_MEM_FENCE);

        // Loop over the values of a single tile
        for (int k=0; k<TSK; k++) {

            // Cache the values of Bsub in registers
            for (int wn=0; wn<WPTN; wn++) {
                int col = tidn + wn*RTSN;
                Breg[wn] = Bsub[k][col];
            }

            // Perform the computation
            for (int wm=0; wm<WPTM; wm++) {
                int row = tidm + wm*RTSM;
                Areg = Asub[k][row];
                for (int wn=0; wn<WPTN; wn++) {
                    acc[wm][wn] += Areg * Breg[wn];
                }
            }
        }

        // Synchronise before loading the next tile
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // Store the final results in C
    for (int wm=0; wm<WPTM; wm++) {
        int globalRow = offsetM + tidm + wm*RTSM;
        for (int wn=0; wn<WPTN; wn++) {
            int globalCol = offsetN + tidn + wn*RTSN;
            C[globalCol*M + globalRow] = acc[wm][wn];
        }
    }
}
```

After inspection of the assembly code, we see that we went from 16 32-bit loads (LD) and local stores (STS) to 4 128-bit loads (LD.128) and local stores (STS.128) for a vector width of 4 floats.

Performance with WIDTH equal to 4 is slightly worse compared to our previous kernel. As before, this is because everything is very sensitive to register pressure and compiler optimisations. Luckily these wider loads will be paying off for the next couple of kernels, so we'll definitely keep them in.



<- previous

next ->

Tutorial written by Cedric Nugteren, (c) 2014 SURFsara
Last updated in 2014 - information might be outdated

Contact:

www.cedricnugteren.nl
web@cedricnugteren.nl

Tutorial: OpenCL SGEMM tuning for Kepler

Note: the complete source-code is available at GitHub.
Note: a tuned OpenCL BLAS library based on this tutorial is now available at GitHub.

<- previous next ->

Kernel 8: CUDA and Kepler-specific optimisations

Although we are a lot faster than clBlas, we are still more than twice as slow as cuBLAS. Is this simply because our choice for OpenCL rather than the superior (?) CUDA? Let's find out!

There are definitely some things that you can do in CUDA that you cannot do with OpenCL. But for starters, let's see what the exact same kernel would do if it were CUDA. To do so, we constructed a crude but functional (at least for our kernels) conversion of OpenCL kernel code to CUDA. This can simply be included as a header file just before including the OpenCL kernel code:

```
// Replace the OpenCL keywords with CUDA equivalent
#define __kernel __placeholder__
#define __global
#define __placeholder__ __global__
#define __local __shared__

// Replace OpenCL synchronisation with CUDA synchronisation
#define barrier(x) __syncthreads()

// Replace the OpenCL get_xxx_ID with CUDA equivalents
__device__ int get_local_id(int x) {
    return (x == 0) ? threadIdx.x : threadIdx.y;
}
__device__ int get_group_id(int x) {
    return (x == 0) ? blockIdx.x : blockIdx.y;
}
__device__ int get_global_id(int x) {
    return (x == 0) ? blockIdx.x*blockDim.x + threadIdx.x :
                      blockIdx.y*blockDim.y + threadIdx.y;
}

// Add the float8 data-type which is not available natively under CUDA
typedef struct { float s0; float s1; float s2; float s3;
                 float s4; float s5; float s6; float s7; } float8;
```

Running the same kernel code through the CUDA toolchain increases performance from 1338 to 1467 GFLOPS. There are a couple of differences:

- Although both paths (CUDA and OpenCL) use the same PTX to binary assembler, they use a different front-end compiler.
- The CUDA toolchain (version 6.5) supports targeting the SM 3.5 architecture and the PTX 4.1 ISA, whereas the OpenCL toolchain (also version 6.5) does not support the native architecture (only up to SM 3.0) and PTX 3.0 at most.
- The CUDA toolchain generates 64-bit PTX on a 64-bit host machine, whereas the OpenCL toolchain always generates 32-bit PTX. The latter is advantageous in our register-heavy case, as pointers to off-chip and local memory become twice as small. The CUDA compiler has the option to generate 32-bit PTX, but only in combination with 32-bit host code.

Now, let's see if we can improve performance further by using SM 3.5 (Kepler) specific optimisations. First of all, we replace our loads with LDG instructions (see below). By simply passing the address we want to load to the __ldg intrinsic, we enable caching into the GPU's L1 texture cache. This gives us a boost from 1467 to 1563 GFLOPS.

```
#ifdef USE_LDG
    floatX vecA = __ldg(&A[indexA]);
    floatX vecB = __ldg(&B[indexB]);
#else
    floatX vecA = A[indexA];
    floatX vecB = B[indexB];
#endif
```

Another SM 3.5 specific optimisation is to use the warp-shuffle instructions to reduce local memory operations. The __shfl intrinsic allows threads within a warp (32) to share data amongst each other without going through the local memory. This can be used for example when loading values of Bsub from the local memory into the register. In the current kernel code, each thread in the first dimension (tidm) loads a set of identical values. We can replace this by letting each of them load a single unique value and share it through warp-shuffles:

```
// Cache the values of Bsub in registers
#ifdef USE_SHUFFLE
    int col = tidn + (tidm % WPTN)*RTSN;
    float val = Bsub[k][col];
    for (int wn=0; wn<WPTN; wn++) {
        Breg[wn] = __shfl(val, wn, WPTN);
    }
#else
    for (int wn=0; wn<WPTN; wn++) {
        int col = tidn + wn*RTSN;
        Breg[wn] = Bsub[k][col];
    }
#endif
```

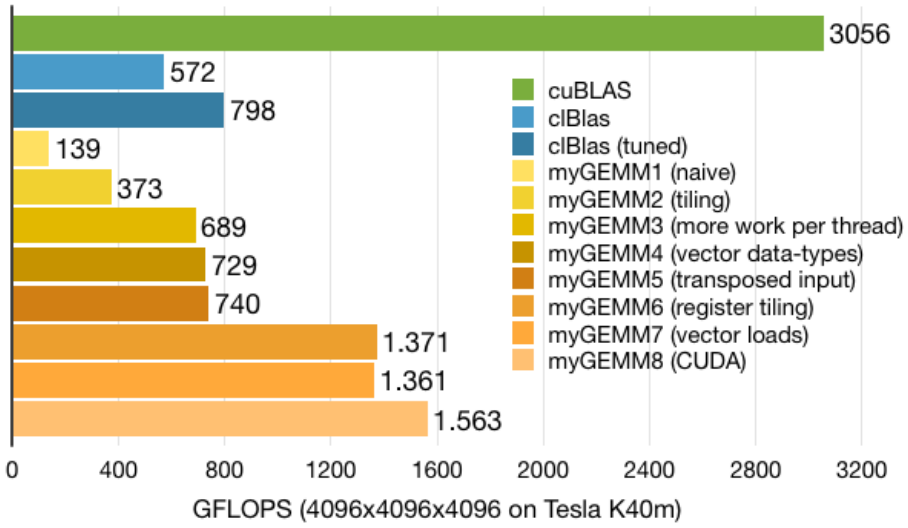
We can do something similar when loading data from A, but is this slightly more complicated and involves

the use of __shfl_up and __shfl_down. In any case, warp-shuffle instructions do not improve performance in our case. In fact, performance is slightly reduced because of the increased number of instructions.

CUDA also gives us a bit more freedom with respect to the L1 cache configuration, but going from a 48KB/16KB configuration to a 32KB/32KB configuration does not give us much extra. The same holds for setting the local memory's bank size to either 4 or 8 bytes:

```
cudaDeviceSetSharedMemConfig(cudaSharedMemBankSizeEightByte);
cudaDeviceSetCacheConfig(cudaFuncCachePreferEqual);
```

Finally, we could try to generate 32-bit PTX code to save precious registers. Since this option (nvcc -m32) also generates 32-bit x86 code, it is not trivial to get this working on our test system. We leave it up to you to test.



<- previous next ->

Tutorial written by Cedric Nugteren, (c) 2014 SURFsara
Last updated in 2014 - information might be outdated

Contact:

www.cedricnugteren.nl
web@cedricnugteren.nl

Tutorial: OpenCL SGEMM tuning for Kepler

Note: the complete source-code is available at GitHub.
Note: a tuned OpenCL BLAS library based on this tutorial is now available at GitHub.

<- previous next ->

Kernel 9: Pre-fetching

Still not at the performance level of cuBLAS, so what's next? Are we using the same techniques as cuBLAS? Inspection using the CUDA visual profiler tells us that cuBLAS is running only a single work-group of 256 threads per SM, that each work-group uses 32KB local memory, and that each thread consumes the maximum of 255 registers. Let's try to mimic that configuration.

First of all, if we go for only 256 threads per SM, we'll suffer from the latency of off-chip memory accesses. There is a well-known technique to overcome that: software pre-fetching. In the case of matrix-multiplication, that means that we are going to load the next tile while we are computing the current tile, such that the latency of the loads can be hidden by computations.

To implement this, we'll need twice as much local memory, automatically limiting us to one work-group per SM (and giving us the same 32KB as cuBLAS). Since we cannot have 3D arrays, we'll just flatten one dimension:

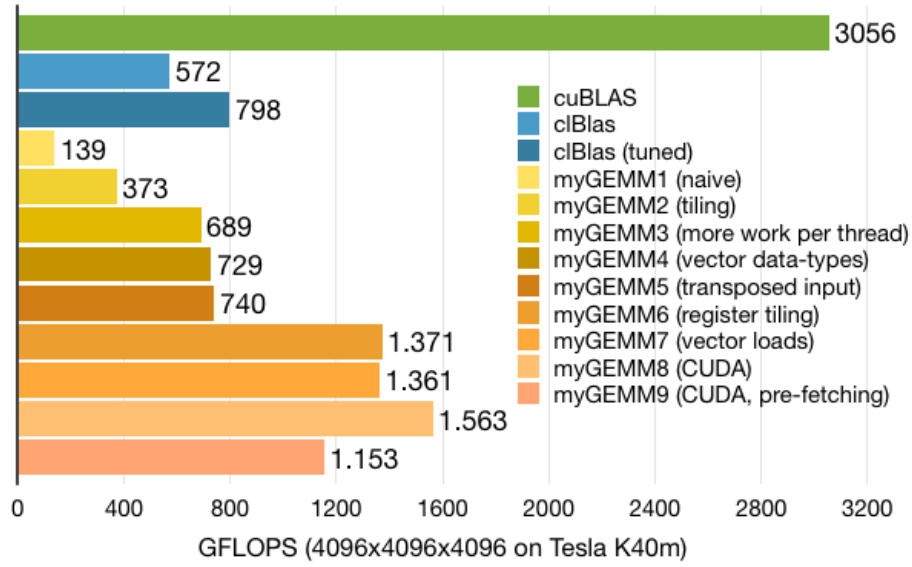
```
1. // Local memory to fit two tiles of A and B
2. __local float Asub[2][TSK*TSM];
3. __local float Bsub[2][TSK*TSN];
```

Since the kernel code has become quite large, we'll take a look at a shorter pseudo-code version. The code below replaces the previous loop over the tiles. Things to notice:

- Before starting the loop, we already load the first tile (tile 0) into Asub[0,*] and Bsub[0,*]. After that, we have to synchronise because there is no other barrier before the start of the computation.
- Within the loop we already load the next tile. Note that we need an extra if-statement for when we arrive at the last tile.
- The computation is as before, but now alternates between the two tiles of Asub and Bsub. With pre-fetching, we no longer need to synchronise directly before computation, allowing overlap between loads and FMAs.

```
1. // Load the first tile of A and B into local memory
2. load(tile=0, Asub[0][*], Bsub[0][*];
3.
4. // Loop over all tiles
5. int numTiles = K/TSK;
6. for (int t=0; t<numTiles; t++) {
7.
8.     // Synchronise
9.     barrier(CLK_LOCAL_MEM_FENCE);
10.
11.    // Load the next tile of A and B into local memory
12.    int tt = t + 1;
13.    if (tt<numTiles) {
14.        load(tile=tt, Asub[tt%2][*], Bsub[tt%2][*]);
15.    }
16.
17.    // Perform the computation
18.    compute(tile=t, Asub[t%2][*], Bsub[t%2][*]);
19. }
```

Since pre-fetching uses twice as much local memory, we end up with only 256 active threads. Although we do pre-fetch, this still harms performance significantly, ending up with ~400 GFLOPS less than the previous kernel. However, we no longer have to care about the 128 registers per thread limit to get 512 threads running, so we have headroom for further optimisations. On to the next kernel!



<- previous next ->

Tutorial written by Cedric Nugteren, (c) 2014 SURFsara
Last updated in 2014 - information might be outdated

Contact:

www.cedricnugteren.nl
web@cedricnugteren.nl

Tutorial: OpenCL SGEMM tuning for Kepler

Note: the complete source-code is available at GitHub.
Note: a tuned OpenCL BLAS library based on this tutorial is now available at GitHub.

<- previous next ->

Kernel 10: Incomplete tiles and arbitrary matrix sizes

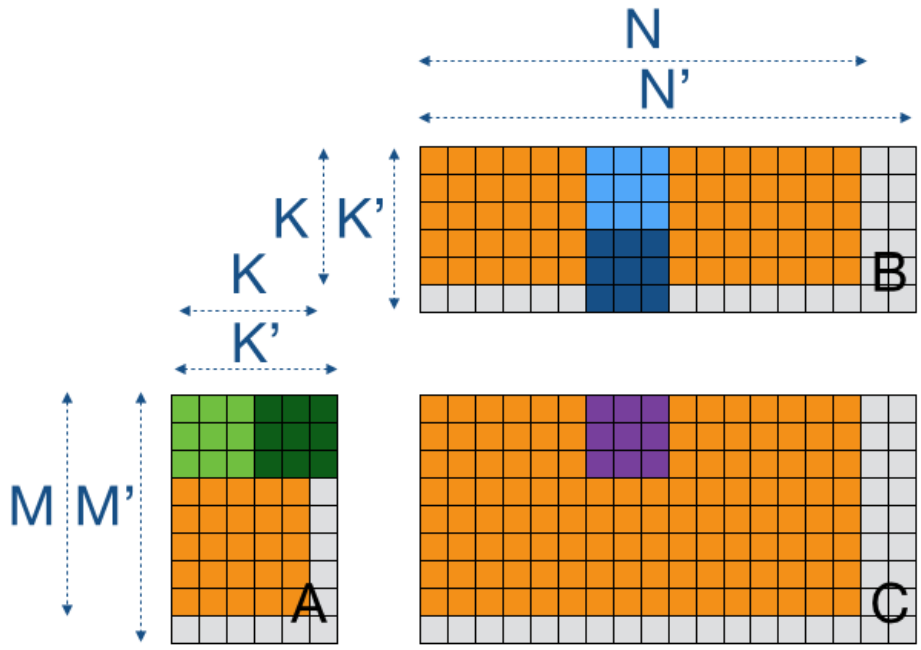
2D register tiling has given us the biggest performance boost (and an increased amount of work per thread before that). So, maybe we can take this one step further, using the extra registers that are now available to us with 256 active threads. The obvious thing to do is increase the register blocking factors WPTN and WPTM from 8 by 8 to 16 by 16. However, that would already consume 256 registers for the accumulation values alone. So, a rectangular or a non-power-of-2 solution would be a more realistic solution.

Before we do anything else, we'll change our load code a bit, allowing us to change TSM and TSN independent of each other. This can cost a bit of performance, but it gives us some generality:

```
// Load one tile of A into local memory
for (int la=0; la<LPTA/WIDTH; la++) {
    int tid = tidn*RTSM + tidm;
    int id = la*RTSN*RTSM + tid;
    int row = id % (TSM/WIDTH);
    int col = id / (TSM/WIDTH);
    (...)
}
// Load one tile of B into local memory
for (int lb=0; lb<LPTB/WIDTH; lb++) {
    int tid = tidn*RTSM + tidm;
    int id = lb*RTSN*RTSM + tid;
    int row = id % (TSN/WIDTH);
    int col = id / (TSN/WIDTH);
    (...)
}
```

Now, if we want to apply 10 by 10 register blocking (or local memory tiling for that matter) to a 4096 by 4096 matrix for example, we'll end up with partially filled tiles. Since we don't have a 1-to-1 correspondence of thread IDs to data loads, we cannot simply add a big if-statement around the kernel to guard partial-tiles. What we can do is pad the input matrices with zeroes to create complete tiles and remove them from the output matrix afterwards.

Padding is done as shown below. We first extend the dimensions K, M, and N into multiples of the tile-sizes TSK, TSM, and TSN to obtain K', M', and N' respectively. Once we have the new arrays, we can proceed as for the earlier kernels, but now we multiply by zero in some cases.



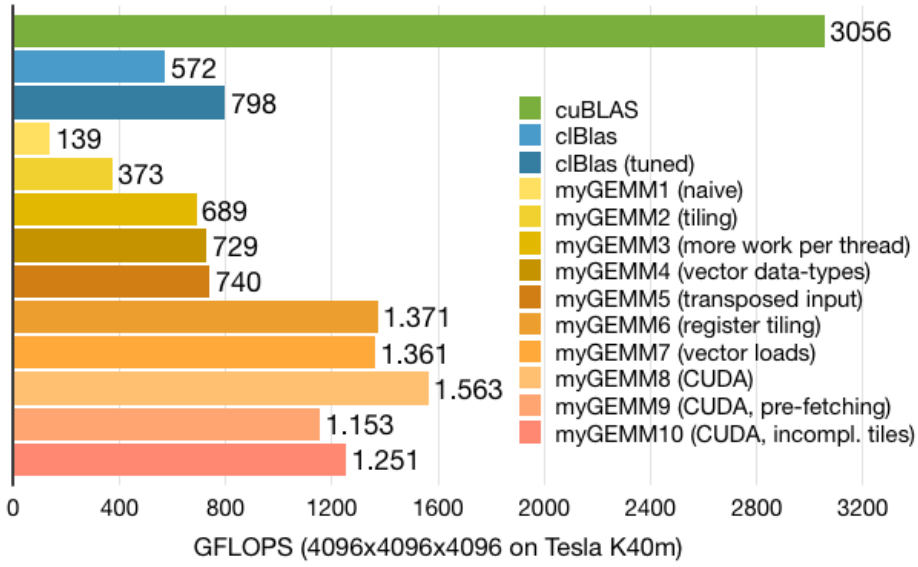
We use simple kernels to add padding to the input matrices A and B, and a similar kernel to remove padding from the output matrix C. The kernel to add padding is shown below:

```
// Pad the P * Q matrix with zeroes to form a P_XL * Q_XL matrix
__kernel void paddingAddZeroes(const int P, const int Q,
                               const __global float* input,
                               const int P_XL, const int Q_XL,
                               __global float* output) {

    // Thread identifiers
    const int tx = get_group_id(0)*PADDINGX + get_local_id(0); // 0..P_XL
    const int ty = get_group_id(1)*PADDINGY + get_local_id(1); // 0..Q_XL
```

```
10.
11.     // Check whether we are within bounds of the XL matrix
12.     if (tx < P_XL && ty < Q_XL) {
13.
14.         // Copy the input or pad a zero
15.         float value;
16.         if (tx < P && ty < Q) {
17.             value = input[ty*P + tx];
18.         }
19.         else {
20.             value = 0.0f;
21.         }
22.
23.         // Store the result
24.         output[ty*P_XL + tx] = value;
25.     }
26. }
```

Note that we now run a total of 5 kernels: 2 to pad the input matrices, one to transpose the B matrix, one to perform the matrix-multiplication, and finally one to remove the padding from the output matrix. The 4 supporting kernels cost around 3% of the total execution time: not very significant, but it might be beneficial to combine transposing and padding or integrate these kernels into the remainder of your application.

Running with tiles of 160 by 16 (matrix A), 16 by 160 (matrix B), and 160 by 160 (matrix C) enables 10 by 10 register blocking while running 16 by 16 threads with 64-bit vector loads. With this configuration, we make use of some of the additional available registers, increasing performance a bit over the previous version: from 1153 to 1251 GFLOPS. Unfortunately, this is not as good as our best version yet (running 512 instead of 256 active threads). On the positive side, the padding adds support for arbitrary matrix sizes, something that could very well be a requirement for real-life applications.



<- previous

next ->

Tutorial written by Cedric Nugteren, (c) 2014 SURFsara
Last updated in 2014 - information might be outdated

Contact:

www.cedricnugteren.nl
web@cedricnugteren.nl

Tutorial: OpenCL SGEMM tuning for Kepler

Note: the complete source-code is available at GitHub.
Note: a tuned OpenCL BLAS library based on this tutorial is now available at GitHub.

<- previous next ->

What's next?

Our last kernel tried to mimic behaviour of cuBLAS by running fewer threads. Unfortunately, it did not even come close in terms of performance. Our best kernel so far was one with 512 active threads, achieving half of the performance of cuBLAS. On the bright side, we've beaten clBlas by a factor 2 to 3, so our work was not a waste. Especially those who are working in OpenCL on NVIDIA GPUs will be thankful.

But why haven't we come close to cuBLAS? Why can it perform so well with only 256 active threads? Let's take a step back and consider whether we really need those extra threads. With 8 warps (256 threads) per SM we can keep Kepler's 192 cores fully occupied. Since there are only 4 warp schedulers per SM and 192 cores, each warp scheduler can schedule up to 2 instructions per warp, given that there is instruction-level parallelism (ILP). So, to hide latency and to get maximum performance, we'll need to have many independent instructions following each other. With pre-fetching we've made sure that off-chip memory latency is no longer an issue, but we don't have much control over the scheduling of the other instructions: this happens in the assembler.

Another issue is the throughput of the register-file. As detailed in the articles listed below, we can only achieve 75% of the peak performance (~3 TFLOPS) because of a limited bandwidth to the register file. If register allocation is not done correctly, we quickly lose a factor of 2 or 3 as we might get register-bank conflicts.

In other words, to get more performance, we'll have to go to assembly level. This will allow us to: (1) schedule instructions for maximum ILP, (2) save precious registers to increase register tiling, (3) use 32-bit addresses, and (4) ensure that there are no register bank-conflicts. With extra registers, we can further increase the tile-sizes and get better performance. But we can't do all of this in OpenCL nor in CUDA: our optimisation story ends here. There are however community-built assemblers for the Fermi architecture and the Maxwell architecture (see below), but there is none for the Kepler architecture.

Further reading:

1. **Fast Implementation of DGEMM on Fermi GPU.** G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao and N. Sun. In: *SC '11.* ACM. Get PDF.
2. **Performance Upper Bound Analysis and Optimisation of SGEMM on Fermi and Kepler GPUs.** J. Lai and A. Seznec. In: *CGO '13.* IEEE. Get PDF.
3. NVIDIA devtalk forums: Is 3-address FFMA faster than 4-address FFMA?
4. Fermi assembler.
5. Maxwell assembler and (very interesting) SGEMM walkthrough.

<- previous next ->

Tutorial written by Cedric Nugteren, (c) 2014 SURFsara
Last updated in 2014 - information might be outdated