Porting of GPU Benchmarking Tests to OpenCL

Utkarsh Jha
Department of Electronics and Communication Engineering
R.V. College of Engineering
Bengaluru, India

***Abstract*—GPU computing has become more popular, with applications ranging from cryptocurrency mining to healthcare. Extreme degrees of parallelism are offered by the high number of cores with GPUs that crunch through data-intensive applications like deep learning. The project's goal is to convert CUDA based SGEMM tests, or single precision matrix multiplication tests, to OpenCL, which is more compatible with heterogeneous computing. When GEMM tests are ported, they will be able to execute on AMD GPUs as well, saving the firm resources and levelling the playing field for benchmarking testing as CUDA based SGEMM tests are typically unfairly optimised for particular hardware.**

***Index Terms*—GPU, CUDA, SGEMM, Tiling, Prefetching, GFLOPS**

I. Introduction

Different GEMM test algorithms are implemented using OpenCL, including: breaking them down into smaller matrices, computing smaller matrices, and increasing work per thread to cut the number of local memory accesses in half. Other algorithms include: broadening the data type to have fewer load/store instructions, incorporating rectangular tiling and padding to have more liberty, making the accumulation registers 2-D, and performing some specific optimizations. For smaller matrices, such as 1024x1024, even the most cutting-edge method fell short and could only achieve 30 percent of the max performance from CUDA.

With the introduction of the NVIDIA CUDA environment in 2007, a flurry of activity in the use of the graphics processing unit (GPU) as a programmable device for general-purpose computing started. Therefore, the number of GPU-accelerated programmes using CUDA has significantly increased during the previous four years. However, CUDA is only supported by NVIDIA GPUs. OpenCL was developed by Apple and submitted to the Khronos team to produce an open-source standard in an attempt to decentralise and democratise the use of GPUs for general-purpose computing [1]. OpenCL is a vendor-neutral framework for creating runtime programmes. The term "processors" refers to all types of processors, not only graphics processing units.

II. Basics of GEMM

GEMMs (General Matrix Multiplications) are a fundamental building block for many neural network operations, such as fully-connected layers, recurrent layers like RNNs, LSTMs, or GRUs, and convolutional layers. GEMM is defined as the operation C = AB+C, where A and B are scalar inputs and C is an output that replaces an existing matrix [2]. A straightforward matrix product is a GEMM with values of one and zero, where AB. For instance, in the forward pass of a fully connected layer, the weight matrix would be argument A, incoming activations would be argument B, and and would typically be 1 and 0.

A. Row vs Column Major

Matrix data may be stored in RAM using one of two opposing strategies. A 2D array of values is assumed to be stored similarly to how letters are arranged on a page, moving from top left to bottom right before diving to the bottom at the end of the row. $\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$ For example, if the memory were to store the above matrix in the following order: — 0 — 1 — 2 — 3 — 4 — 5 — 6 — 7 — 8 — This is known as "row major" because each row is stored in close-by sections of memory. If "column major," an alternative storage technique, were used, the same matrix would be organised in memory as follows: — 0 — 3 — 6 — 1 — 4 — 7 — 2 — 5 — 8 — This shows that we are now moving through the columns to find adjacent memory addresses. It's important to realise that both of these are just ways of storing the same matrix in memory; as implementation details, they shouldn't have an effect on the underlying theory or the construction of equations. Understanding the Convention is really necessary.

B. Transpose

It should be noted that when a row major matrix is provided to a function or library that expects a column major or vice versa, the matrix has essentially been transposed. The matrix may be practically transposed by flipping the rows into the columns. $\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$ is transposed to: $\begin{bmatrix} 0 & 3 & 6 \\ 1 & 4 & 7 \\ 2 & 5 & 8 \end{bmatrix}$ Another way to view this is to draw a line from top-left to bottom-right and flip everything along that diagonal. Since apostrophes are usually employed to denote a matrix's transposition, matrix A would be transposed as A'.

C. Argument Ordering

When two integers are multiplied, the result is always the same: A * B = B * A. This is not true for matrix

multiplications! In reality, only when the number of rows in the right-hand argument matches the number of columns on the left-side can two matrices be multiplied together. Even if they both have the same square size and hence could potentially be exchanged, the result will still depend on the sequencing.

One mathematical identity with transposes commonly shows up in practise. The standard GEMM equation C = A * B leads to the expression C' = B' * A'. In other words, if both input matrices are transposed and their order is reversed, the outcome will be the transposition.

III. GEMM ON GPU (CUDA)

Prior to beginning, it is helpful to quickly review how a framework grid increase is decided. Think about the two lattices, A and B. Assume that A is a nm framework since it has n lines and m segments. Recognize that B is likewise an m-by-w lattice. The outcome of the augmentation AB (which is unique from BA!) is the framework we refer to as M [3]. Overall, there is no difference between the number of sections in the second grid B and the number of lines in the resulting lattice.

Consider M's cell 1,1 in the main line and first segment. The number within is the total of all component-wise duplications of the characteristics in line 1 of A with the numbers in segment 1 of B after the M=AB activity has been completed. This suggests that the whole collection of whole integers found in the I-th line in An and the j-th segment in B are stored in memory for cell I j of network M.

This concept is intuitively explained in the Figure 1:

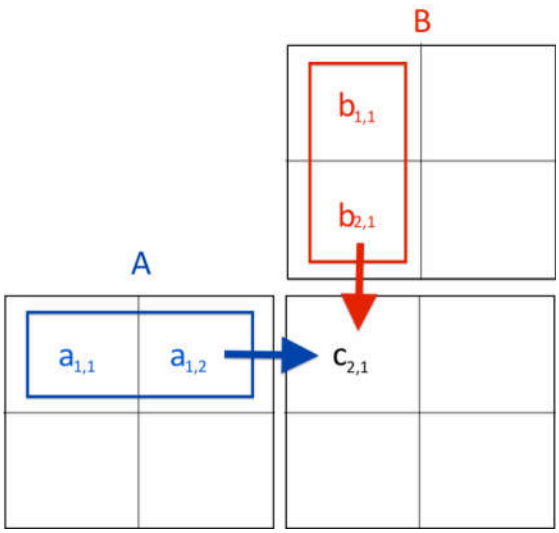

Fig. 1. Matrix Multiplication Illustrated

By now, it should be abundantly clear why matrix-matrix multiplication serves as an excellent example of parallel processing. The independence of each computer component in C enables efficient parallel processing [7]. Additional concepts will be included throughout this article in an attempt to develop a 2D kernel that successfully uses shared memory to optimise operations.

A. Linearise Multidimensional Arrays

Here, we employ 1D exhibits for our networks. However complicated it may appear, the problem is with the programming language. The standard on which CUDA is built needs information on the number of sections prior to purchasing the application [4]. As a result, it cannot be altered or amended in accordance with that aspect of the code.

However, after some contemplation, it becomes clear that this is not a major problem. Currently unable to utilise the natural documentation A[i][j], but this won't be a problem since it is already clear how to appropriately record lines and sections.

In reality, the simplest method to linearize a 2D cluster is to stack each column lengthways from the first to the last. The Figure 2 defines it:

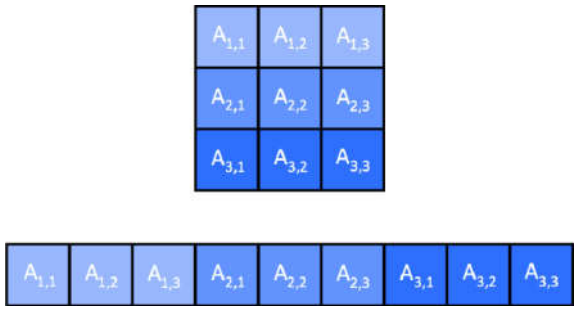

Fig. 2. Stacking Rows Lengthways

B. The Kernel

The task's necessary data has now been collected, making it possible to identify the kernel code. You may use the examples' square NN matrices to keep things simple.Finding the row and column numbers for the x and y axes is the first stage, as was just seen.

The total number of rows and columns must then be checked to ensure that it does not exceed the actual number of rows and columns in the matrices. We must do this to prevent other threads from doing operations on our matrices since they will access the memory in an arbitrary order [5]b3. In other words, if N is not a multiple of the size of the block or grid being produced, then more threads than required will be present:

```
if (ROW < N && COL < N)
```

Although there won't be any problems since square matrices are being utilised, it's typically a good idea to keep this in mind. The cells in the selected row and column must then be added together using the temporary variable tmp sum, which must first be initialised [6]. It is often a good practise to include the decimal points and the f even when the numbers are zeros.

```
float tmp sum = 0 . 0 f ;
```

In the CPU code, Total can be computed using a for loop and then store the result in the relevant C cell. Given the if condition in the kernel, some set N integers which are not

necessarily multiples of the block size. Additionally, the Dev array library will be used.

There are other possibilities, including completing the matrices, creating functions for user input, or using random numbers. In this case, a loop was used to fill the cells with the trigonometric values of the indices:

```
for(int i=0;i<N;i++){
    for (int j=0; j<N; j++){h
        A[i*N+j] = sin(i);h
        B[i*N+j]=cos(j);
    }
}
```

We now need to set the dim3 variables for the dimensions of both grids and blocks, keeping in mind the information from the "Grids and Blocks" section. Since the grid is just a BLOCK SIZE BLOCK SIZE grid, we can write:

```
dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE)
```

Since not just matrices with sizes multiples of BLOCK SIZE are used, the CEIL instruction must be used to retrieve the following integer value as the size. The following is an example [8]:

```
int nblocks=ceil(N/BLOCK SIZE);
dim3 blocksPerGrid(n_blocks,nblocks)
```

This approach will use more threads than required, but the if condition we added to the kernel will prevent them from working on matrices.

This is the comprehensive response that addresses all questions, while kernel.cu has a more potent version of it across the whole code.

Now that the device arrays have been constructed, all that is left to do is allocate memory to each device, start the kernel, and get a parallel matrix multiplication application. using a dev array:

```
dev_array<float> dA(SIZE);d
ev_array<float> dB(SIZE);de
v array<float> dC(SIZE);
```

IV. MPLEMENTATION OF SGEMM IN OPENCL

Different SGEMM (Single Precision General Matrix Multiplication) performance-enhancing algorithms have been developed for OpenCL, beginning with a simple implementation and progressing via prefetching, tiling, padding, 2-D Register filing, and Kepler-specific optimizations. The 2-D Register file was the breakthrough in improving performance, but each version aided in doing so. Transposed input matrices that have padding added for effective memory use improved performance as well.

A. Kernel 1-Basic Matrix Multiplication

Although the first version is sluggish and simple, it offers a place to start and a good amount of parallelism. In this case, two-dimensional threads are produced; in OpenCL, these are

```
__kernel void myGEMM1(const int P, const int Q, const int R,
                      const __global float* S,
                      const __global float* T,
                      __global float* U) {

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

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

// Store the result
U[gCol*M + glRow] = sum
```

known as work items. These threads are equivalent to the sum of M and N.

The thread-identifiers giRow and giCol have been used in place of the indices mi and ni in the loops over P and Q. To make this work, it must be assured that the GPU runs this code P * Q times. This is made possible via the host-code (which runs on the CPU). If OpenCL has been initialised, the proper buffers and memory copies have been produced, a queue has been built, and the kernel code has been compiled, then this is how the aforementioned kernel is run.

B. Kernel-2 Tiling in the local memory

The off-chip memory of the GPU plays a crucial role in the appalling performance of the innocent execution. To implement the Mi*Ni*Ki augmentations and increases, Mi*Ni*Ki*2 loads and Mi*Ni stores are used. Since the additions and duplications may really be combined into a single equipment advice, the computational force of the code is just 0.5 recommendations per memory access (FMA). Even while the GPU's reserves will probably be used to some extent, as seen in Figure 3 even greater speeds can be physically achieved by storing individual networks (tiles) in the GPU's on-chip local memory (otherwise called shared memory in CUDA).

When one looks more closely at the computation of a single component, it should be obvious that a tile contains a tonne of information reuse. For instance as shown in Figure 4, all elements on a certain column of the purple tile (in the 3x3 tiles of the image below) are decided using the same data as the green tiles.

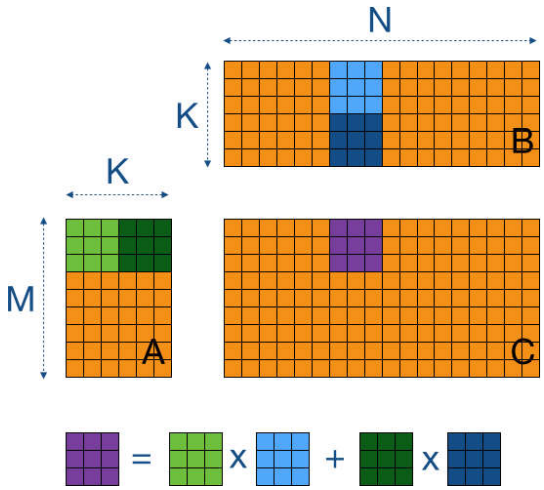

Fig. 3. Tiling matrices

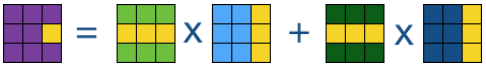

Fig. 4. Computing from Tiles

C. Kernel-3 More Work per Thread

Increasing the amount of work that each thread does is one technique to enhance the previous kernel (and thus reducing the total number of threads). Examine the PTX assembly produced by the previous kernel. focusing in particular on the (unrolled) inner k-loop, which is the primary computational componenent.

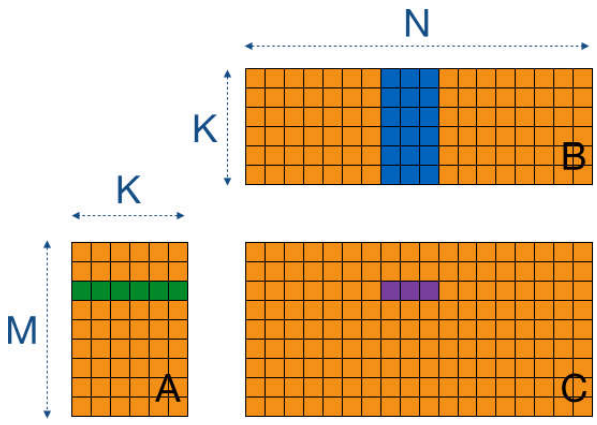

Fig. 5. WPT factor of 3

We squander time since just one of our three instructions really performs a useful calculation. At the register-level, the same approach that was previously utilised with tiling may be used. If one thread is permitted to calculate eight items that are arranged in successive columns of C, accesses to A may be reduced by an order of eight. Given that a constant tile size is maintained, the amount of local memory accesses rather than off-chip memory accesses is reduced in this case. This is shown in the Figure 5, when tiling is disregarded and a WPT of 3 is applied.

D. Kernel-4 Wider data-types

The prior kernel required more work in the C column-dimension. That this might have been done in the row-dimension as well is obvious. Additionally, this has the advantage of reducing the demand on local memory, but it also has the ability to support more diverse data types. However, it is also feasible to increase the work per thread in the row-dimension of C by considering vector data-types rather than loops over WPT. NVIDIA GPUs do not enable vector operations like addition and multiplication in They do, however, have special broader load and store instructions that work with off-chip and local memory. The effectiveness of hiring them on our performance was assessed.

Firstly, define the launch parameters for the new kernel while keeping in mind that data types with the prefix ”Width” will be used: Since they are already defined by OpenCL, using floating-point vector types is quite convenient. Instead of turning the o matrices into vector types, just cast the pointers in the kernel. To keep our flexibility, create a new data type that enables pre-processor define setting of the WIDTH parameter. Support for the float8 data-type has been omitted from the code in order to improve readability, despite the fact that WIDTH is actually set to 8 for the test since it offers the best performance.

E. Kernel-5 Transposed input matrix and rectangular Tiles

By dramatically lowering the frequency of off-chip memory accesses, the early tiling implementation showed how performance might be increased. However, if bigger tiles, such as 64 × 64, were to be employed, the system would run out of resources. On the Tesla K40 GPU that is used, the 48KB of local memory per SM enables the operation of many work-groups. This shows that for a 32 by 32 tile, 2*32*32*4 = 8KB per work-group is required, giving considerable headroom.

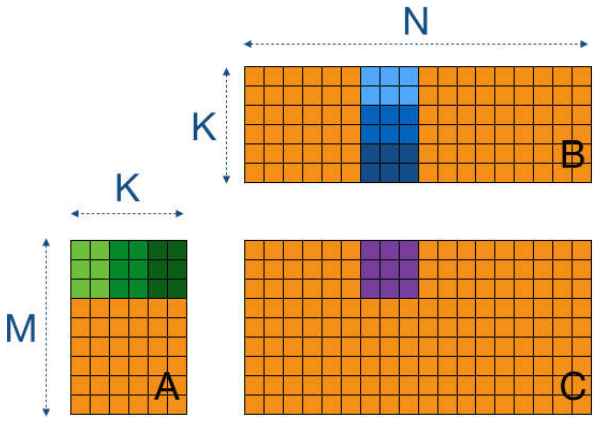

Fig. 6. Computing 3x3 tile

Tiles that are rectangular should be used so that parameters may be changed and have greater flexibility. Due to the fact that both matrix A and matrix B have the size K, the Figure 6 illustrates how rectangular tiles are made. Following three iterations of three 3x2 tiles each, the 3x3 tiles in the example picture need to be calculated.

```
/ 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];
}
}
```

F. Kernel-6 2D Register Tiling

The previous kernel (myGEMM5) is built on top of it, with the alterations listed below: Now 2D, the accumulation registers are initialised using a double-loop. The outcomes are also kept in the C matrix using a similar double-loop. The difficulty of loading data has increased Then, a loop over the number of loads per thread (LPTA = LPTB) is put on top of the thread stack (in the first and second dimensions). The variable ”id” is present and will be divided into the row and column IDs using integer division and modulo.

An older version of the kernel allowed for the storage of one value of Asub and its subsequent reuse in the WPT loop. It has been established (using Areg) and pull off a similar trick using Bsub. Breg needs WPTN temporary registers, despite the fact that its reuse is not in the innermost loop. This new kernel has a significant performance variance and makes heavy use of resources. Let’s start by examining the use of the local RAM: local memory bytes equal 4 * (TSK + 2) * TSM + 4 * TSK * TSK * TSM.

G. Kernel-7 Wider Loads with register blocking

To accomplish 64-bit or 128-bit loads and stores, an earlier kernel required wider data types. Include it again in the most recent 2D register blocking iteration. Only loads will be utilised with it. Constants and launch parameters don’t change. The kernel is likewise quite similar, with the main differences occurring in the area where data is loaded from off-chip memory: Float2 or float4 data should now be loaded from off-chip memory.

As a consequence, the loop has been multiplied by the quantity of loads per thread using a factor called WIDTH. The same guidelines apply to indexing into vector arrays. As a result of the somewhat altered access pattern when saving data into Asub and Bsub, change the column and row indices of Bsub back to the way they were for Asub. The computational loop must now load LDS more often than LDS.64, but as this memory is now cached via register tiling, the impact will be less noticeable.

The fact that vector data-types may now be utilised to store data into Bsub is the main new advantage. Since Asub and Bsub are not defined as vectors, the stores into them seem to be scalar from the standpoint of coding, but they really become 64-bit or 128-bit stores. After compilation, (STS.128) We now have that additional amount of local RAM back once the padding was removed.

H. Kernel-8 CUDA and Kepler Specific Optimisations

Undoubtedly, certain CUDA-only features are not supported by OpenCL. First, let’s look at what the same kernel might do in a CUDA environment. This was accomplished via a simple yet efficient (at least for our kernels) translation of the OpenCL kernel code to CUDA. This may be accomplished quickly by placing it just before the OpenCL kernel code in a header file:

When using the CUDA toolchain, the identical kernel code performs better, going from 1338 to 1467 GFLOPS. There are several differences: The front-end compilers for CUDA and OpenCL are separate, despite the fact that they both use the same PTX to binary assembler. The CUDA toolchain (version 6.5) enables targeting the SM 3.5 architecture and the PTX 4.1 ISA, but the OpenCL toolchain (also version 6.5) only supports the native architecture (up to SM 3.0) and PTX 3.0 at most.

The CUDA toolchain creates 64-bit PTX on a 64-bit host machine, but the OpenCL toolchain always generates 32-bit PTX. The latter is desirable in register-heavy scenarios because references to off-chip and local memory become twice as small. only when the CUDA compiler’s option to create 32-bit PTX is combined with 32-bit host code.

V. RESULTS AND DISCUSSIONS

Beginning with the most basic algorithm, this project sought to get performance from OpenCL SGEMM that was comparable to that of CUDA SGEMM. Gradually, as algorithmic complexity rose, the Gflops began to rise. Finally, the algorithm attempted to imitate CuBlas SGEMM, a library function made available by CUDA.

A. Kernel-1 Results

Since this was naive implementation Gflops Performance was only 139, compared to 3000+ for Cublas as shown in Figure 7.

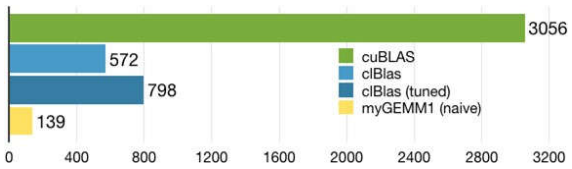

Fig. 7. Gflop Performance Comparison

B. Kernel-2 Results

In kernel-2 tiling was implemented which is breaking down the matrix to smaller matrices to increase the parallelism, two components can be distinguished that are divided by synchronisation barriers within this loop across the tiles: (1) loading from off-chip memory to local memory, and (2) calculation based on local memory data. The number of global loads performed by each thread per tile (one each for elements of A and B) has decreased from the previous value of two times the tile size (a row of A and a column of B). This results in a 32-fold reduction in off-chip memory accesses!Gflops saw a sharp increase when compared with naive implementation. As shown in Figure 8, still the Performance is much behind CLblas and Cublas.

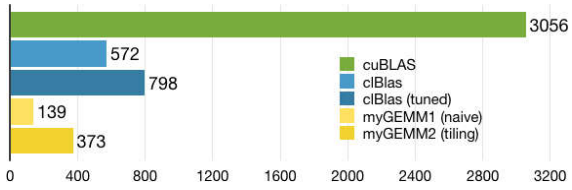

Fig. 8. Gflop Performance Comparison

C. Kernel-3 Results

Looking at the updated PTX assembly, it can be observed that the proportion of FMA-instructions has significantly increased. For instance, the computation-loop is shown below in 8 iterations (instead of the previous 2 iterations), demonstrating that we only need 8+1 loads from the local memory for 8 FMAs (instead of 8+8). Looking at new kernel's performance, it appears that its already outperforming clBlas! Nevertheless, as seen in Figure 9 by the performance gap compared to cuBLAS, there are a few other techniques that can be used to boost performance.

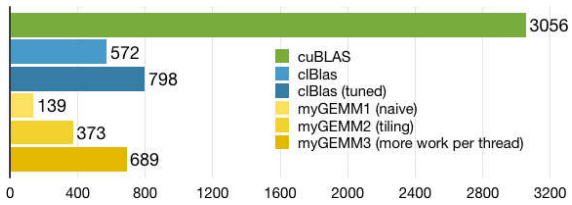

Fig. 9. Gflop Performance Comparison

D. Kernel-4 Results

In order to make the code more legible, support for float8 data type has been left out , even though WIDTH is really set to 8 for benchmark because it provides the best performance. The kernel is based on the same idea as the previous kernel, however some changes have been made: Instead of dimension 1, which now corresponds to rows of C, the extra work per thread is found in dimension 0. (columns of C). As a result of the data kinds Now, for vectors, each index must be split by a factor called WIDTH. Since it is not a vector FMA operation, the inner-loop computation must be written out. A regular loop plus a scalar pointer to local memory can be used as an alternate approach. Due to its wider data types and thus fewer load/store instructions, as seen in Figure 10 this kernel performs a little bit better than the prior kernel. But when we combine these two methods, the true advantage will become clear.

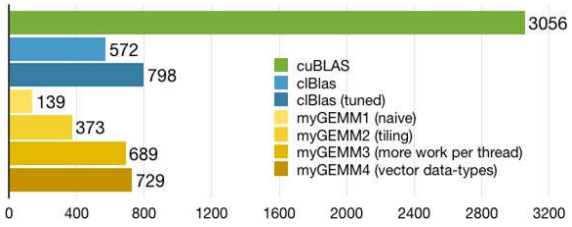

Fig. 10. Gflop Performance Comparison

E. Kernel-5 Results

Using the same settings as before (squared tiles of 32 by 32 and a WPT of 8) only 460 GFLOPS is achieved, less than the 689 of the earlier kernel. The decreased performance is caused by bank-conflicts in the local memory. When everything is working and padding was included, their is freedom to select 64 by 32 blocks for example. This gives a little bit of extra performance over the earlier kernel's 32 by 32 blocks, as seen in Figure 11 giving 740 GFLOPS.

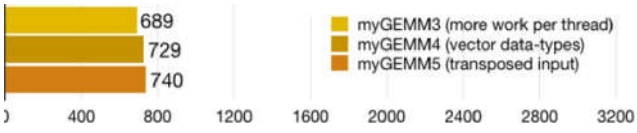

Fig. 11. Gflop Performance Comparison

F. Kernel-6 Results

Running 256 or 512 active threads has a significant impact on our kernel. Due to this, our code is extremely vulnerable to compiler errors or minor code modifications (such as the volatile keyword above). Even compiler options like -cl-nv-maxrregcount=127 may not always be helpful because the compiler may decide to spill the additional registers it 'needed' to RAM. This kernel's best-case performance is well over 1.3 TFLOPS, which is more than twice the clBlas rating. However, making little changes to the code (such as unrolling a specific loop or utilising the volatile keyword) can increaseour performance by anywhere between 800 and 1300 GFLOPS. Overall, a big improvement over earlier kernels.

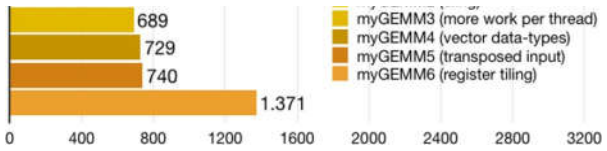

Fig. 12. Gflop Performance Comparison

G. Kernel-7 Results

Inspecting the assembly code reveals that for a vector width of 4 floats, we switched from 16 32-bit loads (LD) and local stores (STS) to 4 128-bit loads (LD.128) and local stores (STS.128). When WIDTH is set to 4, performance is marginally poorer than with our previous kernel as can be seen in Figure 13. This is due to how vulnerable everything is to compiler optimizations and register pressure. We will absolutely keep these wider loads in since they will fortunately pay off for the following several kernels.

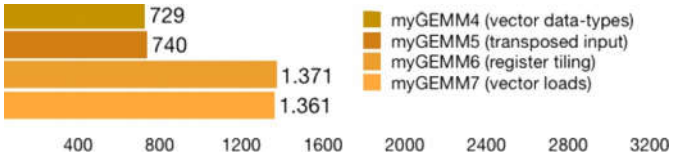

Fig. 13. Gflop Performance Comparison

H. Kernel-8 Results

When it comes to the L1 cache design, CUDA also provides us a little more latitude, although the difference between a 48KB/16KB configuration and a 32KB/32KB configuration is not very significant, as shown in Figure 14. The same is true for changing the bank size of the local memory to either 4 or 8 bytes: 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.

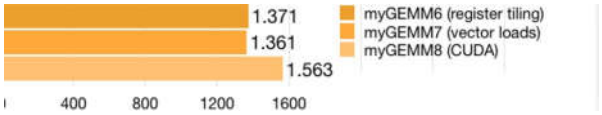

Fig. 14. Gflop Performance Comparison

I. Modified Kernel 8

The previous Results were obtained while working with matrices smaller in size, when kernel-8 was modified for matrices with large size and kept under similar loop coniditions as the intial SGEMM code for CUDA it gave very similar results for Gflops, as shown in Figure 15, although the power consumption was only one third of what CUDA was fetching.

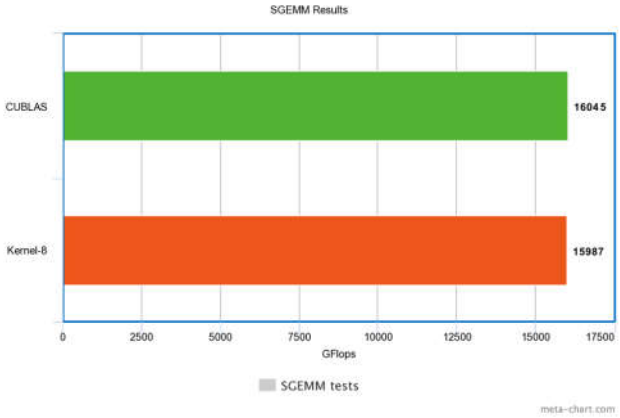

Fig. 15. Gflop Performance Comparison

VI. Conclusion

This project's initial goal was to convert SGEMM code written in CUDA to OpenCL while still obtaining comparable Gflops to those seen in the Code. Only Nvidia-compatible devices could execute the SGEMM code in CUDA.Snippets of the original code were converted first, in the original code, Cublas SGEMM function received initialised matrices. Finding an algorithm and methods to replace Cublas' SGEMM function was the goal of this effort. Starting with the crude matrix multiplication implementation, we next move on to tiling and prefetching methods. The finished kernel was included into the original converted code.

In contrast, the Naive implementation could only provide 3 percent of the 3056 Gflops produced by the Cublas library for a 256x256 matrix. Prefetching made it possible for the final kernel 8 to populate 2-D registers at a rate of 1563 Glops!, a significant increase. With the same restrictions and fewer loops than the original code, Kernel-8 produced 15987 Gflops for a matrix of size 10240x6192, which is almost as fast as CuBlas 16045. As a result, the code's goals were achieved, and conversion was successful.

ACKNOWLEDGMENT

The work described in the paper was supported by the department of Electronics and Communication (ECE), R.V. College of Engineering located in Bengaluru, Karnataka, India. The guide, Dr. Govinda Raju M., Assistant Professor and head of the department supported the project in all the phases of the project, thus would like to express gratitude for their guidance.

REFERENCES

[1] Sang Ik Lee, Troy Johnson, and Rudolf Eigenmann. "Cetus – An Extensible Compiler Infrastructure for Source-to-Source Transformation". In: vol. 2958. May 2018, pp. 539–553. isbn: 978-3-540-21199-0. doi: 10.1007/978-3-540-24644-2-35

[2] Paul Sathre, Mark Gardner, and Wu Feng. "Lost in Translation: Challenges in Automating CUDA-to-OpenCL Translation". In: Sept. 2018, pp. 89–96. isbn: 978-1-4673-2509-7. doi: 10.1109/ICPPW.2012.15.

[3] John Stratton, Sam Stone, and Wen-mei Hwu. "MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs". In: vol. 5335. July 2016, pp. 16–30. isbn: 978-3-540-89739-2. doi: 10.1007/978-3-540-89740-8-2.

[4] Kazuhiko Komatsu et al. "Evaluating performance and portability of OpenCL programs". In: (Jan. 2016).

[5] John Stone, David Gohara, and Guochun Shi. "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems". In: Computing in science engineering 12 (May 2015), pp. 66–72. doi: 10.1109/MCSE.2010.69.

[6] Rodrigo Dominguez, Dana Schaa, and David Kaeli. "Caracal: Dynamic translation of runtime environments for GPUs". In: Jan. 2014, p. 5. doi: 10.1145/1964179.1964186.

[7] Gregory Diamos et al. "Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems". In: 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT). 2010, pp. 353–364.

[8] Shuai Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: 2009 IEEE International Symposium on Workload Characterization (IISWC). 2009, pp. 44–54. doi: 10.1109/IISWC.2009.5306797.