Don't let that CPU sit idle: Hardware-aware heterogeneous general matrix multiplication (GEMM)

Johnathan Warawa [1] Sean Chester [1]

[1]University of Victoria, Department of Computer Science

Background

General matrix-matrix multiplication (GEMM) is the multiplication of two matrices (A, B) to produce a third (C), denoted here as $A \times B = C$.

- $O(n^3)$ operation (asymptotically faster algorithms do exist but are not used in practice [2])
- Widely used with many applications; is a fundamental operation in AI applications [2]

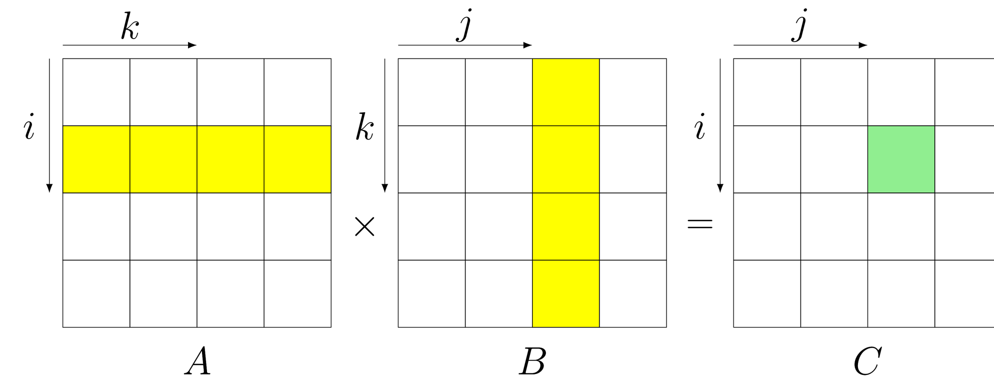

Figure 1. Example calculation of one cell of the result matrix. The dot product of the yellow vectors is taken to produce the green result.

Heterogeneous Computing refers to using multiple different types of processors concurrently. Traditional computer systems will have one or more general-purpose CPU (central processing unit) cores. GPUs (graphics processing units) which are highly specialized processors which excel at specialized highly *parallelizable* tasks (able to be split into many small parts which can be done simultaneously) such as GEMM [1].

Research question

GPUs cannot run on their own - they require a CPU to drive them [1] ("assign" them work). However, this means CPUs are not always fully utilized while the GPU works. The research question is thus: **can we design a GEMM algorithm which leverages both a CPU and a GPU simultaneously to exceed the performance of either individually?**

Accelerating GEMM on the CPU with AVX

Modern CPUs are designed to work most efficiently using ***single instruction, multiple data*** (SIMD) operations which perform operations on contiguous data stored in vectors at once. We use the AVX2/AVX512 instruction sets found on modern Intel and AMD CPUs.

Deep dive: Algorithm design & low-level optimizations

SIMD instructions perform elementwise operations on vectors. In the main loop, we use one vector from the A matrix and one from the B matrix, using FMA (fused multiply-add) instructions to perform the products. For each output cell, we iterate along the k dimension, storing them in a accumulator vector (Figure 2).

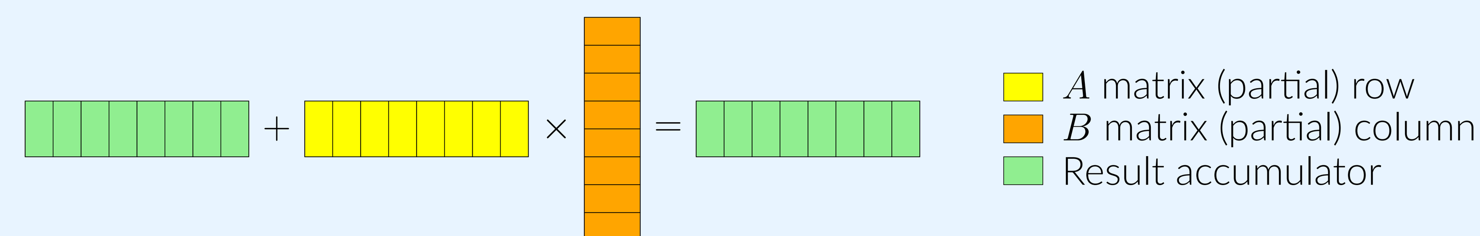

Figure 2. Example of one FMA instruction and application in our algorithm

When we finish iterating, we have 8 (or 16, depending on vector width) elements in a vector which must be summed to retrieve the output value. AVX does not natively support "horizontal" operations like this, so we take 8 or 16 of these result vectors and reduce them at once. Figure 3 demonstrates one step of this process (other steps use similar instructions).

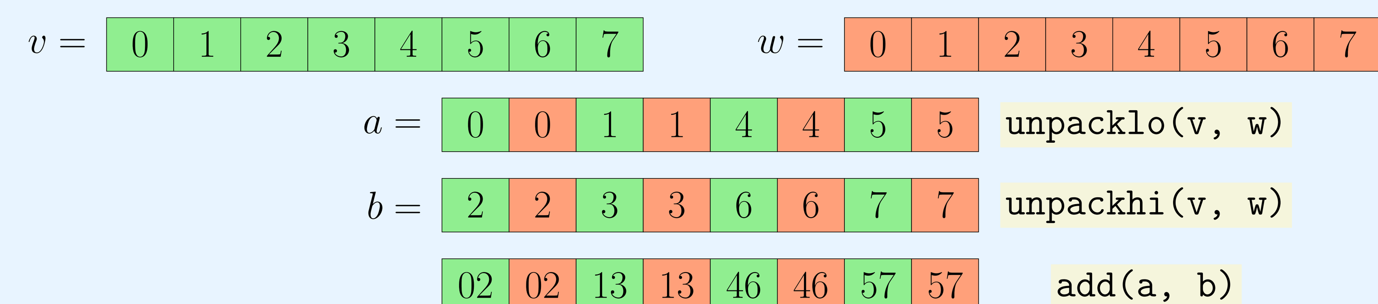

Figure 3. Example first step of the multi-reduction with 8 elements per vector. Note that xy represents $x + y$.

Accelerating GEMM on (Nvidia) GPUs

Modern (2017+ [2]) Nvidia GPUs have *tensor cores* which contain specialized matrix multiplication instructions. These instructions operate on small *matrix fragments* (usually 8×8, 8×16, or 16×16) which are combined to compute an entire product [2].

Deep dive: Driving the tensor cores

The widely used way to program an Nvidia GPU is CUDA, their C++ extension for programming Nvidia GPUs [1]. However, the current and most efficient way to directly program tensor cores is by using PTX [2], an *intermediate language* (similar to assembly, but with a few conveniences such as type checking) developed by Nvidia. NVCC, Nvidia's toolchain (set of compiler tools) compiles CUDA code into PTX, as it acts as a lower-level representation [1].

```
ldmatrix.sync.aligned.m8n8.x4.b16 {af0, af1, af2, af3}, [a_ptr];
ldmatrix.sync.aligned.m8n8.x2.b16 {bf0, bf1}, [b_ptr];
mma.sync.aligned.m16n8k16.row.col.f32.f16.f16.f32 {acc0, acc1, acc2, acc3},
                                                  {af0, af1, af2, af3},
                                                  {bf0, bf1},
                                                  {acc0, acc1, acc2, acc3};
```

Figure 4. Example PTX matrix multiplication commands, using the latest "MMA" instruction set

We programmed our GPU algorithm primarily in CUDA, switching to PTX for the hottest code (the code that runs most frequently) to allow fine-grained control.

Making the two processors work together

As the GEMM algorithm is deterministic in terms of the number of operations needed for any input, we use a *static* scheduler, splitting work on the matrices before beginning the algorithm. The schedule for a particular system is determined through a grid search, splitting on a multiple of a fixed "block size" number of rows (eg, 128 rows). By matching the "block size" to the amount of cache the CPU or GPU had, we ensured that we would not loose CPU or GPU performance.

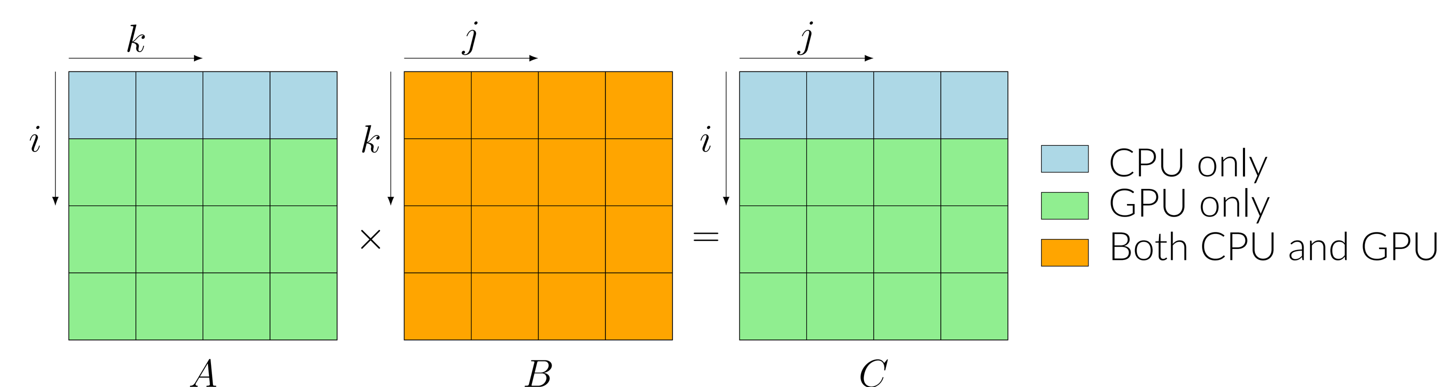

Figure 5. Example work schedule

Deep dive: Connecting multiple programming languages

We foudn that toolchains expect developers to write performant CPU or GPU code, *but not both at the same time*. Our build process, described in Figure 6, involved several steps.

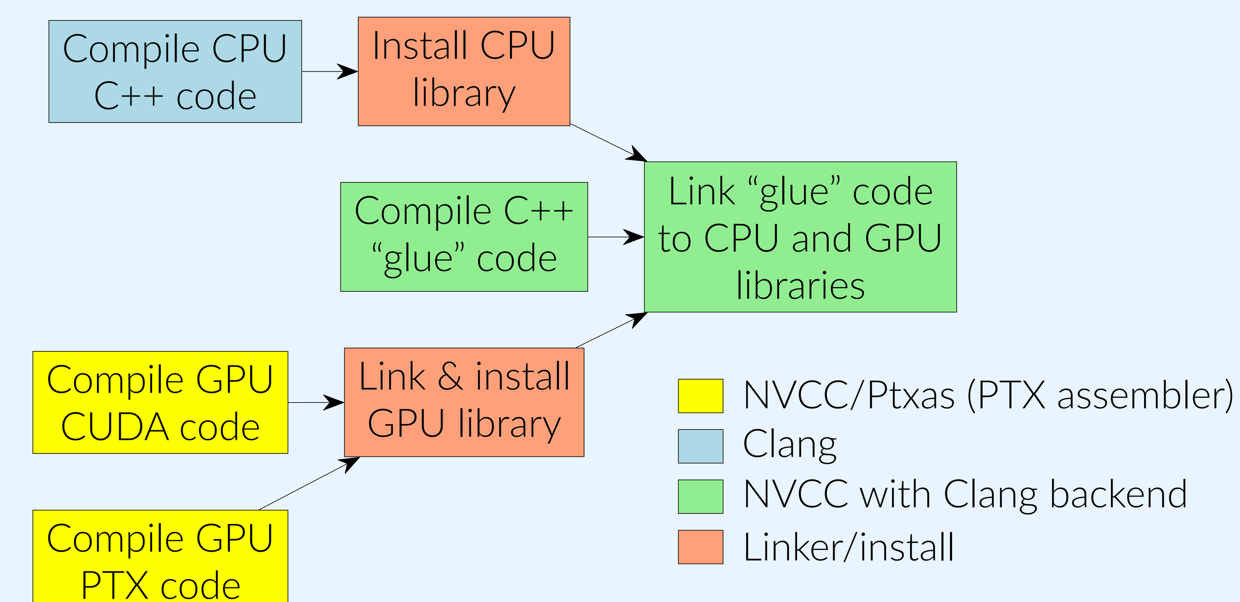

Figure 6. Build process workflow

Results

We measure the end-to-end latency of our algorithm using FLOPS (floating point operations per second), by taking the number of operations required for the GEMM operation and dividing by the time taken. Because our objective is to show that a heterogeneous algorithm outperforms a purely GPU- (or CPU-) algorithm, we use speedup to measure our results:

$$\textit{speedup} = \frac{\text{heterogeneous algorithm GLFOPS}}{\text{max(GPU algorithm GLFOPS, GPU algorithm GFLOPS)}}$$

A speedup greater than 1 shows that our heterogeneous algorithm outperforms the alternative.

We ran our experiments on a system with a AMD Ryzen 9 3900X (the CPU algorithm was given 8 CPU cores) and a Nvidia RTX 3060, taking an average of 10 runs. Figure 7 shows our results at different matrix sizes. We achieved about 33-50% of the maximum theoretical speedup.

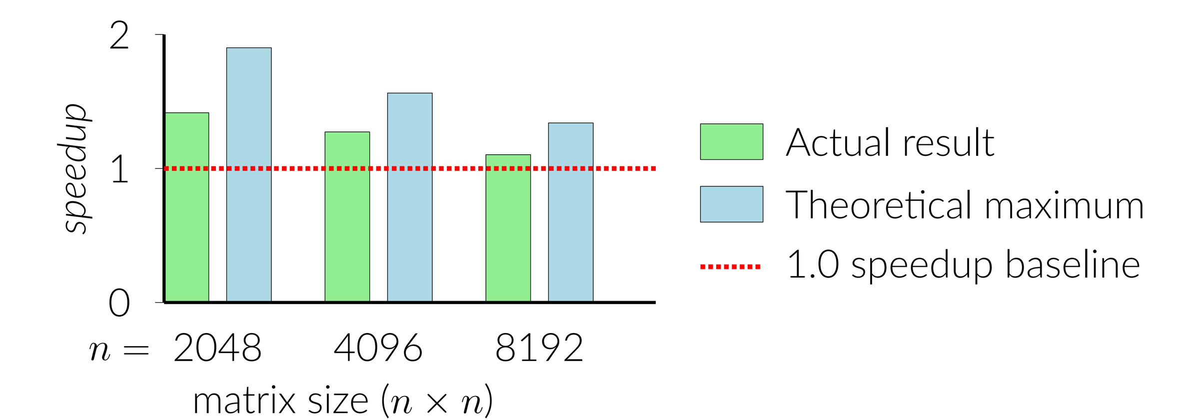

Figure 7. Performance analysis of our algorithm on various sizes of $n \times n$ matrices.

In each case, we are still can achieve positive results even with a suboptimal schedule. Figure 8 shows results with various schedules for $n = 8192$.

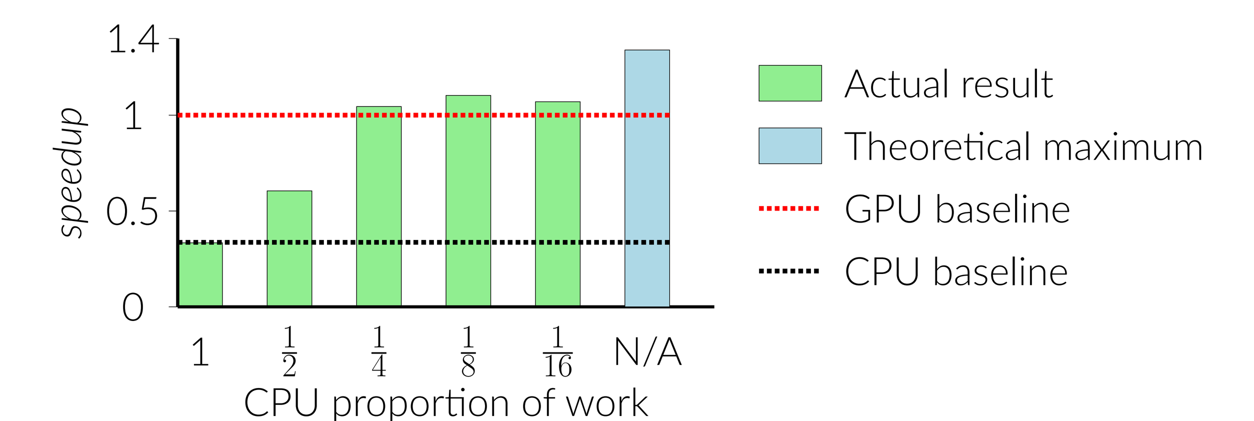

Figure 8. Performance analysis of our algorithm on various schedules with 8192×8192 matrix sizes.

Conclusion

We have demonstrated that simply saying "Speed up your GEMM operations by running them on a GPU" does not tell the full story - CPUs, while not as powerful as GPUs, are still powerful, and a better way to fully take advantage of available hardware is to use a heterogeneous algorithm which leverages both processors at once. Driving a GPU does not require so much CPU time that the CPU should be discounted entirely, as even a suboptimal schedule will improve performance.

Deep dive: Future work

Many current AI technologies leverage matrices with *structured sparsity*, often 2:4 sparsity, where every 2 out of 4 elements are 0. **The next step would be to apply our algorithm to multiplication of a matrix with a 2:4 sparse matrix.**

- We have already developed a CPU 2:4 sparsity algorithm which achieves 1.35-1.4x speedup compared to our GEMM algorithm and 1.2x speedup as compared to the fastest CPU GEMM (and 2:4 sparsity) algorithms.
- Recent Nvidia GPUs (2020+) contain hardware support for 2:4 sparsity, which can be programmed in PTX. These can achieve 1.7x speedup in benchmark tests. [2]

Because both algorithms achieve similar speedup when using structured sparsity, we believe that we would be able to achieve a similar ratio of speedup with 2:4 sparsity.

References

[1] Nvidia. CUDA Programming Guide, December 2025.

[2] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):246–261, January 2023.

This research was supported by the Jamie Cassels Undergraduate Research Awards, University of Victoria. Supervised by Sean Chester.

johnnyw3@uvic.ca