CSE-234: Data Systems for Machine Learning, Winter 2025

6: GPU Matmul and Operator Compilation

Lecturer: Hao Zhang

Scribe: Lucas Lee, Mahesh Kannan, Haisheng Chen, Casey Hild, Vaidehi Sinha, Ganesh Bannur, Annie Xu, Aksay Ghosh, Andrew Xi, Leonard Zhang, Baizhou Zhang, Shijie Wang

1 Recap

1.1 GPU Execution Model & Thread Hierarchy

A central idea introduced was the GPU's thread hierarchy, which is conceptually divided into:

1. Grids – a collection of blocks that execute the same kernel
2. Blocks – a group of threads that share memory
3. Threads – smallest units to process a chunk of data

When we write a GPU kernel in CUDA, we run it by launching many threads according to this hierarchy.

1.2 Distributed Address Space

GPUs maintain a separate device address space (often called global or device memory, physically implemented as High Bandwidth Memory, HBM).

To enable effective data transfer between CPU and GPU, CUDA provides built-in memory allocation and transfer APIs, such as:

- cudamalloc (allocation of memory)
- cudamemcpy (transfer of memory)

Pinned memory Memory that is reserved on the host for GPUs to optimize data transfer between CPU/GPU and there are also APIs for this.

1.3 GPU Internal Memory Hierarchy

Within the GPU, there are also other memory hierarchies besides the HBM.

- per thread - private memory for local variables

- per block - shared memory (SRAM) can be accessed for threads in the same block using the __shared__ keyword in CUDA
- per device - global memory (HBM) can be accessed by any thread in any block

1.4 Topics during last lecture that wasn't covered during this lecture's recap

- Barrier synchronization primitive for threads in a block
- Barrier synchronization primitive between CPU and GPU
- First GPU program (window averaging/conv1D)

2 Learning Goal

For the first half of the lecture we discuss about the implementation of matmul on GPU. Once the GPU programming components are covered we then discuss alternative methods to make the operations faster such as compiler optimization and wrap up the operation augmentation layer and introduce graph optimization.

3 CUDA Thought Process

The way we program in CUDA, we need to keep in mind that it is different from conventional programming for other languages as we need to leverage the fact that CUDA programming is SIMD (Single Instruction Multiple Data). In order to do so we need to do the following 3 steps:

- We should identify the work that can be parallelized.
- We then need to partition the task and identify the subset of data that is associated with each partition and bind pair to a CUDA thread.
- Implement the logic in the CUDA kernel and the GPU will launch the code into multiple threads under the hood.
- In order to improve execution time, we follow the techniques used in the conv 1D example seen previously in class.

For maximum efficiency, we need to ensure that

- **Oversubscription:** Creating a large number of partitions of work and push them onto the limited number of GPU workers.
- **Mitigate Straggler:** We need to balance the workload across the threads as equal as possible such that each thread has same amount of workload. We need to avoid If-else conditional statements at all costs.
- **Minimize "Communication":** Reduce the amount of data amount between the memory hierarchies (shared memory $\leftrightarrow$ global memory) by using shared memory as much as possible keeping in mind the constraints of the same.

4 GPU Matmul v1

1. One Thread per C-Element
2. Multiply two square matrices A and B of size N×N to get the result matrix C.

4.1 Kernel Code Logic

A simplified version of the kernel (GPU function) might look like this:

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  float result = 0.0f;
  for (int k = 0; k < N; ++k) {
    result += A[x][k] * B[k][y];
  }
  C[x][y] = result;
}
```

Thread and block dimensions:

```
    int N = 1024;
    dim3 threadsPerBlock(32, 32, 1);
    dim3 numBlocks(N/32, N/32, 1);
    mm<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

threadsPerBlock(32, 32, 1) specifies a block of 32×32=1024 threads.

numBlocks(N/32, N/32, 1) indicates how many blocks cover the full N×N matrix. For N=1024, this is (32,32) blocks. The total number of threads is (32×32)×(32×32)=1024×1024=1,048,576

5 Global Memory Access and Performance Analysis

5.1 Global Memory Reads per Thread

A single thread needs to fetch the entire x-th row of A (size N) and the entire y-th column of B (also size N) to compute the dot product. This equates to $N + N = 2N$ floating-point reads per thread.

5.2 Total Global Memory Reads

There are N^2 threads (one for each C-element). Total read operations can be around $N^2 \times 2N = 2N^3$ which is very large for bigger N

5.3 Memory Footprint

Storing A, B, and C each requires N^2 floats in global memory.

6 Memory Hierarchy and Register Tilling

In modern GPUs memory is organized in a hierarchy that balances speed and capacity. Thread can take advantage of fast, "thread-local" registers to reduce slower memory accesses.

6.1 Grid, Blocks, and Threads

- Grid: A grid is the collection of all thread blocks running a particular GPU kernel (kernel = a function you run on the GPU)
- Thread Block: Each block is a set of threads that can cooperate via fast, on-chip shared memory and can synchronize with each other
- Thread: A single execution unit within a block. Each thread has its own set of registers

7 GPU Matmul v1.5: Thread Tiling

7.1 Previous Method (GPU Matmul v1)

In GPU Matmul v1, we used a straight forward approach. We computed the dot product between a row of the first matrix and a column of the second matrix on each thread.

Thus, for a matrix of size N x N, this method required $2N$ memory reads per thread to compute the dot product and required N^2 threads, one for each row, column pair. As a result, the total global memory access required was $2N^3$ memory reads. However, the local memory required was only 1 float per thread to store each result.

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;

  result = 0;
  for (int k = 0; k < N; ++k) {
    result += A[x][k] * B[k][y];
  }
  C[x][y] = result;
}
```

7.2 Improved Method (GPU Matmul v1.5)

The goal of this improved method is to have each thread compute an entire V x V submatrix instead of only 1 entry.

Essentially this means that we divide the original N x N resulting matrix into a grid of N/V x N/V blocks. Thus, each block is of size V x V. Within each of these blocks, we then compute each of these V^2 entries.

The goal is to have each of these blocks of size V x V be computed by one thread.

Thus, for the overall matrix of size N x N, this method requires $(N/V)^2 = N^2/V^2$ threads and each thread requires $NV + NV^2$ memory reads. This number of memory reads is reduced from the expected $2NV^2$ because computing this V x V block all at once allows us to reuse the row from the first matrix throughout the each entire iteration of the outer loop (i.e. each row is reused V times). As a result of this reuse, we only read from the first matrix NV times while we read from the second matrix NV^2 times. Thus, the total global memory access requires is $(N^2/V^2)(NV + NV^2) = N^3/V + N^3$ memory reads. However, to compute the values of the whole V x V block, the local memory required must increase to $V^2 + 2N$ floats per thread as can be seen in the code below.

```
__global__ void mm(float A[N][N], float B[N][N], float C[N][N]) {
  int ybase = blockIdx.y * blockDim.y + threadIdx.y;
  int xbase = blockIdx.x * blockDim.x + threadIdx.x;

  float c[V][V] = {0};
  float a[N], b[N];
  for (int x = 0; x < V; ++x) {
    a[:] = A[xbase * V + x, :];
    for (int y = 0; y < V; ++y) {
      b[:] = B[:, ybase * V + y]
      for (int k = 0; k < N; ++k)
        c[x][y] += a[k] * b[k];
      }
    }
  }
  C[xbase * V: xbase*V + V, ybase * V: ybase*V + V] = c[:];
}
```

8 GPU Matmul v2: Can we do better?

To get the resultant matrix, we can either use the traditional row-by-column multiplication or break the matrices into smaller submatrices. Instead of computing each element individually, we partition the matrices, dividing X into $X_1, X_2, \ldots$ and Y into $Y_1, Y_2, \ldots$.

Each small region of the resultant matrix is computed as $X_1Y_1 + X_2Y_2$. The key advantage of this method is that each required region is read only once, whereas in the previous version, the same data had to be accessed multiple times. This reduces redundant memory reads and makes the computation more efficient by improving memory usage.

Kernel Implementation

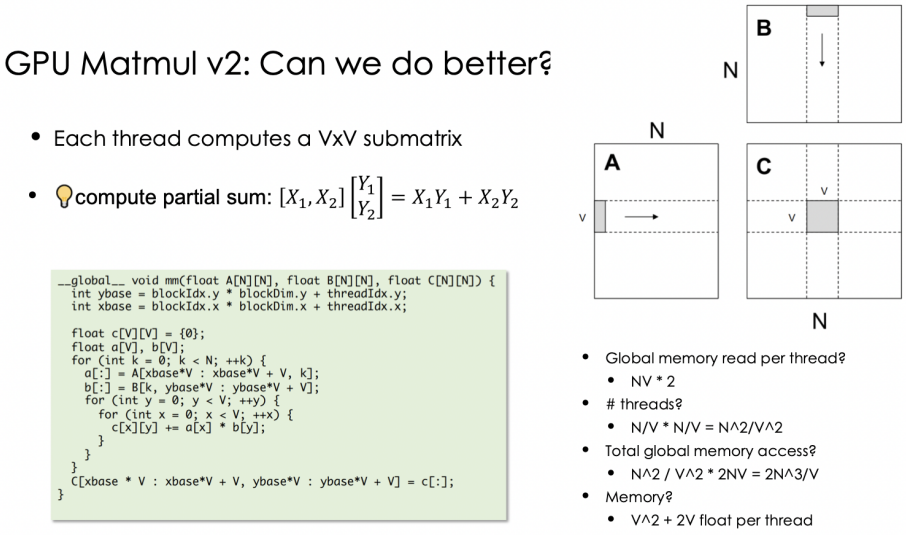


Figure 1: GPU Matmul v2

The kernel follows this approach and modifies the previous implementation. The key steps are:

- **Locating the region:** Uses `blockIdx` and `threadIdx` to determine the position of each $V \times V$ region in the output matrix.
- **Register allocation:** Once the region is identified, the kernel allocates the required registers.
- **Looping over matrix dimensions:** The outer loop (k) iterates over the **columns of** A and **rows of** B, ensuring proper matrix multiplication.
- **Loading elements:** The necessary elements from A and B are loaded into registers.
- **Performing computation:** Accumulates **partial sums** to compute the submatrix.
- **Storing the result:** The computed $V \times V$ submatrix is stored back into **Matrix** C.

Memory Access and Performance Analysis

- **Global memory reads per thread:** $NV \times 2$ (each thread loads V elements from A and V elements from B).
- **Number of threads:** $\frac{N^2}{V^2}$ (each thread computes a $V \times V$ submatrix).
- **Total global memory access:** $\frac{N^2}{V^2} \times 2NV = \frac{2N^3}{V}$.
- **Memory requirement per thread:** $V^2 + 2V$ floats, covering the registers used for computation.

Memory Hierarchy and Register tiling

To optimize performance, we need to take advantage of shared memory within the memory hierarchy. Rather than relying solely on global memory, which has higher latency, data is first moved to shared memory and then to registers, ensuring faster access and minimizing redundant memory operations.

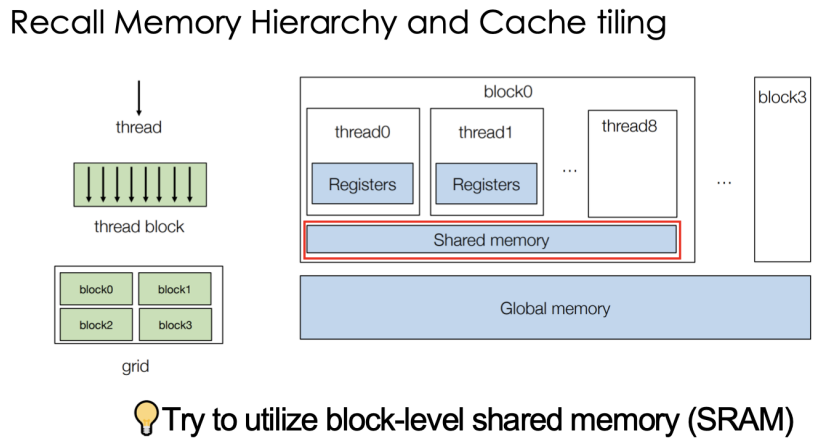


Figure 2: Memory Hierarchy and Register Tiling

9 GPU Matmul v3: SRAM Tiling

9.1 Method

In matmul v3 we make use of the shared memory available to a block. Recall that each block has shared memory which is accessible to all the threads in that block. When performing matmul using two level tiling the following divisions occur. Figure 3 shows the matrices we are considering.

At the first level we let each block compute an $L \times L$ sub-region of the output (let's call it C_{block}). Then at the second level we let each thread in the block compute a $V \times V$ sub-region of C_{block} (let's call it C_{thread}). To compute C_{thread} we need to multiply two sub-regions of A and B of sizes $V \times N$ and $N \times V$. Adjacent threads will have an overlap in the sub-regions of either A or B that they load. For example if a thread is calculating `C[0:5, 0:5]` and another thread is calculating `C[0: 5, 5:10]` then they will both load `A[0: 5, 0: N]`. So if we are able to load the sub-regions into shared memory once and use it in all the threads, tiling will speed up matmul because of memory re-use.

How this is done is first regions of size $S \times L$ (gray blocks in A and B in Figure 3) are loaded into shared memory. Each thread will use the corresponding part of this which has size $S \times V$ (let's call these A_i and B_i). Now the thread will perform $C_{thread} \leftarrow C_{thread} + A_i \times B_i$. In the next iteration the next $S \times L$ region is loaded and the threads repeat their computations. Finally each thread computes it's $V \times V$ region $C_{thread} = \sum_{i=0}^{\frac{N}{S}} A_i \times B_i$.

9.2 CUDA Kernel

Listing 1 gives the CUDA kernel which performs matmul v3.

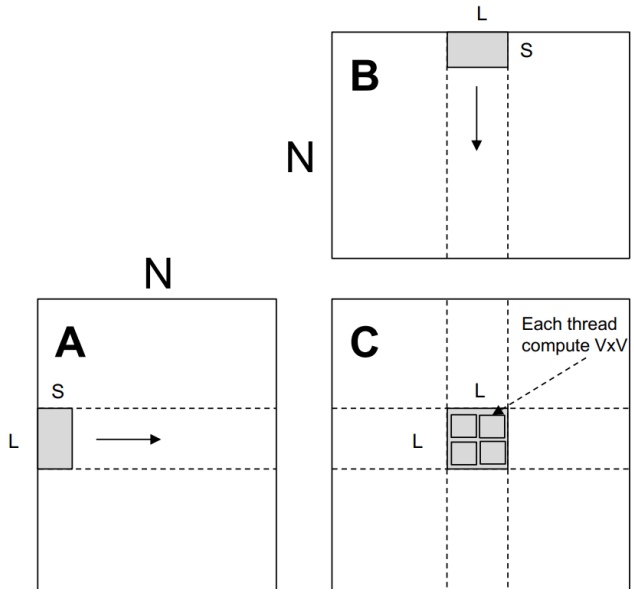


Figure 3: Matrices being considered

Listing 1: matmul v3 CUDA kernel

```
1    __shared__ float sA[S][L], sB[S][L];
2    float c[V][V] = {0};
3    float a[V], b[V];
4    int yblock = blockIdx.y;
5    int xblock = blockIdx.x;
6    for (int ko = 0; ko < N; ko += S) {
7        __syncthreads();
8        // needs to be implemented by thread cooperative fetching
9        sA[:, :] = A[ko : ko + S, yblock * L : yblock * L + L];
10       sB[:, :] = B[ko : ko + S, xblock * L : xblock * L + L];
11       __syncthreads();
12       for (int ki = 0; ki < S; ++ ki) {
13           a[:] = sA[ki, threadIdx.y * V : threadIdx.y * V + V];
14           b[:] = sA[ki, threadIdx.x * V : threadIdx.x * V + V];
15           for (int y = 0; y < V; ++y) {
16               for (int x = 0; x < V; ++x) {
17                   c[y][x] += a[y] * b[x];
18               }
19           }
20       }
21   }
22   int ybase = blockIdx.y * blockDim.y + threadIdx.y;
23   int xbase = blockIdx.x * blockDim.x + threadIdx.x;
24   C[ybase * V : ybase*V + V, xbase*V : xbase*V + V] = c[:];
25 }
```

Line 2 in the code uses the `__shared__` keyword to declare `sA` and `sB` in shared memory. `sA` and `sB` will

hold the $S \times L$ regions (gray blocks in A and B in Figure 3) which are loaded in each iteration. In this code `sB` is declared as an $S \times L$ array even though the region is $L \times S$. This is done to simplify the code and we can assume the matrix `sB` has been transposed. In lines 5, 6 we use the `blockIdx` to determine which $L \times L$ sub-region (C_{block}) is assigned to the block. The outermost for loop in line 7 goes over $S \times L$ regions one at a time. This corresponds to loading $S \times L$ regions from left to right in A and top to bottom in B. The lines 10, 11 load the current $S \times L$ regions in A and B into shared memory. However the way `sA` and `sB` are loaded here is not correct. In this implementation every thread will load the entire $S \times L$ block. In reality we need to partition the loading between threads so that each thread only loads one part. This is called co-operative fetching. We keep the incorrect implementation for simplicity. The `__syncthreads` in line 12 is needed so that all threads have finished loading their part (remember the actual implementation is through co-operative fetching). Since we are loading into shared memory that can be accessed by all threads, we need to ensure that loading has finished or else we run the risk of getting undetermined behaviour. The `__syncthreads` in line 8 is needed in case a thread which finishes the remainder of the loop faster than others and starts loading the regions for the next iteration. We want to make sure that all the threads have finished accessing the current region before we start loading the next. Now that we have the regions needed by all the threads in shared memory, the threads can perform the operation $C_{thread} \leftarrow C_{thread} + A_i \times B_i$ mentioned before. This is implemented by the next three loops. The loop at line 13 goes over every column of A_i ($V \times 1$) and row of B_i ($1 \times V$) and multiplies them to give a $V \times V$ matrix which is added to C. This is different from the way we think about normal matrix multiplication (in which we multiply rows of the first matrix with columns of the second). But it gives the same result.

Compared to previous version of matmul we just added the part which loads $S \times L$ regions into shared memory. Next we can calculate the memory overhead of this implementation. This includes the number of global and shared memory accesses. Global memory is only accessed in lines 10, 11. We load regions of sizes $S \times L$ but ultimately we end up loading the entire $N \times L$ region which is needed to compute C_{block} (of size $L \times L$). We load one $N \times L$ region each from A and B so the total global memory access for one block is $2NL$. There are $\frac{N^2}{L^2}$ blocks and so the total global memory access is $2NL \times \frac{N^2}{L^2} = \frac{2N^3}{L}$. Next we calculate shared memory accesses. Only lines 14, 15 perform shared memory accesses. Each thread loads two $S \times V$ regions in each iteration of the loop. But ultimately the thread needs to load the entire region of size $N \times V$ in both A and B in order to calculate C_{thread} (of size $V \times V$). So the memory access for a thread is $2VN$. There are $\frac{N^2}{V^2}$ threads and so the total shared memory access is $2VN \times \frac{N^2}{V^2} = \frac{2N^3}{V}$.

10 Other GPU Optimization

Cooperative fetching is when all threads fetch memory from one source area. When implementing cooperative fetching, the developer determines what subregion a thread is responsible for. Developers need to calculate position and offset so that a thread can fetch and write things into shared memory (SM).

There are many other optimizations that make GPU kernels run faster.

- One way is having global memory continuous read. Reading off of offsets is slow, so we want to always do continuous read.
- Avoiding shared memory bank conflict. If all threads read from an area without offset, we could run into conflicts (this is called bank conflict).
- We can pipeline threads so that some threads do reading and some threads so computation. This way, we save time by overlapping memory IO and computation time, and not doing things sequentially.

- Tensor Core is a specialized core for mat-mul operations, so using Tensor Core will speed up mat-mul computations.

- Lower precision will be covered in next week's lecture

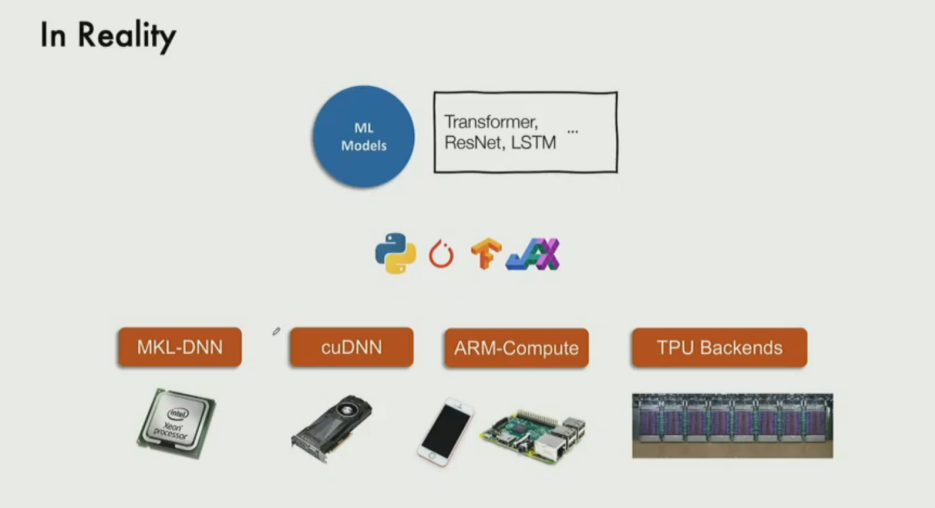
11 Core Problems of CUDA

Upon considering the approach described in GPU Matmul v3, several follow up questions arise:

How should one choose L (block sub-matrix size) and V (matrix computed by thread)?

The answer to this question depends on the number of threads, number of registers, and the amount of SRAM available on the system. We cannot write a CUDA program that requires more resources than are available on the execution hardware. The process of determining the optimal hyperparameters for a kernel is called kernel tuning; when PyTorch or Tensorflow is loaded, it automatically determines (profiles your system for) the optimal hyperparameters based on your system.

Kernel tuning is an especially difficult problem since there is such a wide variety of models, execution devices, network architectures, and operations. Therefore, it is hard to determine which settings are optimal for different hardware. One could enumerate over all possible hardware configurations and tune the performance for each config, but this is an extremely expensive and labor intensive approach:



The second approach is to develop machine learning based compilers that take the compiler, code, and devices and automatically determine the optimized implementation for operations. The machine learning approach greatly reduces labor costs and automates this work.

12 Introducing Compilers

12.1 Primary Goal

The primary goal of machine learning (ML) compilers is to automatically generate optimal configurations and kernel code from high-level code (e.g., TensorFlow, PyTorch, Python) for target hardware. This approach eliminates manual labor by transforming user-written code into a highly efficient version that can be directly executed on hardware.

12.2 Comparison: Traditional vs. ML Compiler

12.2.1 Traditional Compiler:

1. **Input**: Human-written code focusing on general-purpose programming languages (e.g., C++)
2. **Process**: Compilers eliminates inefficient and unnecessary code by transforming it into machine-readable instructions optimized for specific hardware.
3. **Output**: Machine instructions executed by hardware

12.2.2 ML Compiler

1. **Input:** User-defined ML models in frameworks like TensorFlow or PyTorch, represented as dataflow graphs.
2. **Process:** Compilers automatically transform dataflow graphs into an optimized version and generates efficient kernel code for various operators used within the graph. Additionally, they use existing compilers, such as CUDA, to produce hardware-specific machine code
3. **Output:** Optimized machine code for efficient execution of ML workloads.

ML compilers can be seen as higher-level compared to traditional compilers. Having a successful ML compiler means that most machine learning system problems will be solved.

12.3 Problems

1. **Programming-level issues:** Using arbitrary, imperative user code to generate compile-able code can be very challenging due to the difficulties in transforming highly dynamic user-code into static dataflow graphs. It is an active area of research conducted by people focusing on programming languages
2. **Graph-level issues:** Optimizing and transforming dataflow graphs automatically to make it faster
3. **Operator-level issues:** Generating efficient kernel code for individual operators on diverse hardware

12.4 Notable ML Compilers

12.4.1 XLA

Developed by Google and released with TensorFlow in 2016, XLA was the first compiler specifically designed for machine learning. Recognizing the inefficiency of manually developing 200+ operators for

multiple hardwares (CPU, GPU, TPU), Google created XLA. Today, most TensorFlow backends rely on XLA, and its scope has been extended to PyTorch and other frameworks. Additionally, since XLA serves as the best backend for running on TPUs, it has been integrating PyTorch into it.

12.4.2 TVM

TVM, Tensor Virtual Machine, is a famous, open-source compiler project led by TQ Chen from Princeton University. It primarily focuses on compiling for inference rather than training, as it does not generate backward passes. TVM has been highly successful in academia, and its researchers and students created the startup OctoML, which has raised nearly $200 million before being acquired by Nvidia.

12.4.3 PyTorch 2.0

PyTorch's compiler, torch.compile, was introduced as part of PyTorch 2.0. PyTorch started with an imperative programming model and gradually integrated compilation into its framework. It gained immense popularity due to its ease of use and continued to add new features for its user base. Torch.compile has become a key feature for optimizing PyTorch workflows, and is also integrating XLA to enable efficient execution on TPUs.

12.4.4 Modular

Modular is another compiler founded by Chris Lattner, the creator of LLVM, a highly sucessful compiler framework for C++ and other languages. After leaving Google and Apple, he started Modular. One to two years, ago, he claimed to achieve performance 20 times faster than PyTorch, sparking controversy and debate in the community. The company has raised nearly $300 million and remains a significant player in the compiler field.

13 Operator Compilation

13.1 Loop Splitting

Given user-defined high-level program, how compiler converts high-level program into low-level hardware-efficient codes. The loop splitting task of compiler is to automatically find the loops that works best with your hardware. The hardware refers to specifications of CPUs and GPUs like the number of registers, the size of caches, and the size of shared memory etc.

13.2 Problem Definition

- Possibilities: How to represent the search space and all possibilities?
- Searching: How to find the closest-to-optimal value in the search space?
- Acceleration: How to reduce the search space for efficiency?
- Generalizability: How to generalize the search on different hardwares?

13.3 Case Study: AutoTVM

Components of AutoTVM framework (Fig. 4):

- Search Space: Search Space is pre-defined.
- Search Planner: Enumerate through the space.
- Code Generator: Compile kernel codes based on user's code and hardware specifications.
- Cost Model: Apply the codes on target hardware and model its performance, e.g. using neural networks to generalize and predict the performance, taking loop parameters as the input of the function and performance as the output of the function.

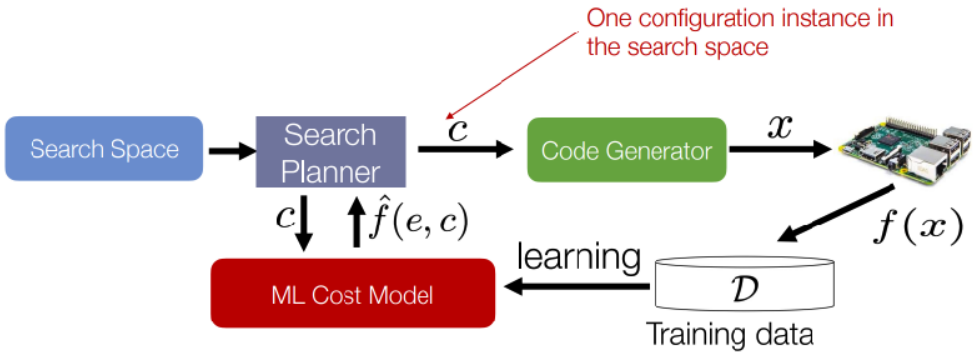


Figure 4: AutoTVM framework

Template-based Search Space requires experts to write a template, i.e. writing the skeleton codes and leaving the looping factors as parameters for compiler optimization.

Searching Strategies:

- Early Pruning the search space: pruning the bad performance candidates in the early stages of searching.
- Cost Modeling: using historical data to predict the performance of loops on target hardware.

13.4 Elements of Automated ML Compilers

- Program Abstraction: Representation the program/optimization of interest.
- Build Search Space: Good coverage of common optimizations like tiling.
- Effective Search: Accurate cost models and transferability.

14 Triton

14.1 Motivation

In the last chapter, we have discussed about ML compilers that can automatically generate codes on target hardware. ML compilers promise to generate optimal configurations on hardware, however they didn't fully delivered this promise. To show this, the famous Flash Attention kernel used by most language models is invented in 2022, which is later than the invention of ML compilers. Rather than discovered by compilers, Flash Attention is written as highly optimized handcrafted Cuda kernels.

Since manual written kernels are still highly in need, Triton came out as a high-level DSL(domain specific language) for Cuda.

14.2 Device-specific DSL vs. Compiler

For device-specific DSLs like Cuda, the advantage is users have high flexibility when developing kernels. They can use whatever data-structures and low-level techniques that can squeeze the last bits of performance. The disadvantage is that the development process requires deep expertise, and it's really hard and time-consuming to write correct and fast kernels.

For compilers like XLA or TVM, the advantage is users can write kernels quickly and prototype ideas at high iteration speed. The disadvantage is the compilers cannot represent certain types of ideas involving in-operator control flow and custom data structures (eg. FlashAttention). Also, they rely heavily on templates and pattern matching, which cause lots of performance cliffs.

Triton tries to strike a balance between DSLs and compilers:

- On development difficulty, Triton is Python based and simpler than Cuda. While Triton is more complicated than graph compilers to deliver higher expressing ability.
- On expressing ability, Triton is less expressive than Cuda, but more expressive than graph compilers. Users can craft some delicate algorithms like Flash Attention with Triton.

14.3 Triton Programming Model

The programming model of Triton is based on several ideas:

- Triton is embedded in Python, and kernels are defined with triton.jit decorator.
- Users construct tensors of pointers in SRAM, and modify them elementwise with torch-like primitives.
- The tensors must have power-of-two number of elements along each dimension, otherwise Triton will do the padding.

Figure 5 is an example of implementing elementwise addition using Triton. The kernel is mapped to a single block (SM) of threads, and users are responsible for the mapping of multiple blocks. The size of block is parametrized as BLOCK variable, so Triton can better handle tiling and avoid manipulating loops.

Inside the kernel, primitives like tl.arange and tl.load are used to vectorize the loading of elements. tl.program_id varaiable indicates the position of current block, and the position of each thread is represented by offsets. The mask parameter is used for bound checks of array size.

```
import triton.language as tl
Import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N, BLOCK: tl.constexpr):
  # same as torch.arrange
  offsets = tl.arange(0, BLOCK)
  offsets += tl.program_id(0)*BLOCK
  # create 1024 pointers to X, Y, Z
  x_ptrs = x_ptr + offsets
  y_ptrs = y_ptr + offsets
  z_ptrs = z_ptr + offsets
  # load 1024 elements of X, Y, Z
  x = tl.load(x_ptrs, mask=offset<N)
  y = tl.load(y_ptrs, mask=offset<N)
  # do computations
  z = x + y
  # write-back 1024 elements of X, Y, Z
  tl.store(z_ptrs, z)


N = 192311
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
grid = lambda args: (triton.cdiv(N, args['BLOCK']), )
_add[grid](z, x, y, N)
```

Figure 5: Elementwise Add Kernel in Triton

Contributions

The following individuals contributed to this scribe note:

- **Lucas Lee:** Added and revised Section 1: Recap
- **Mahesh Kannan:** Added and revised Section 2: Learning Goal, and Section 3: CUDA Thought Process
- **Haisheng Chen:** Added and revised Section 4: GPU Matmul v1, Section 5: Global Memory Access and Performance Analysis, Section 6: Memory Hierarchy and Register Tiling
- **Casey Hild:** Added and revised section 7: GPU Matmul v1.5 - Thread Tiling
- **Vaidehi Sinha:** Added and revised Section 8: GPU Matmul v2 - Can we do better
- **Ganesh Bannur:** Added and revised Section 9: GPU Matmul v3 - SRAM Tiling
- **Annie Xu:** Added and revised content in Section 10: Other GPU Optimization
- **Akshay Ghosh:** Added and revised content in Section 11: Core Problems of CUDA
- **Andrew Xi:** Added and revised content in Section 12: Introducing Compilers
- **Leonard Zhang:** Added and revised content in Section 13: Operator Compilation
- **Baizhou Zhang:** Added and revised content in Section 14: Triton
- **Shijie Wang:** Compiled, proofread, and submitted the scribe note.