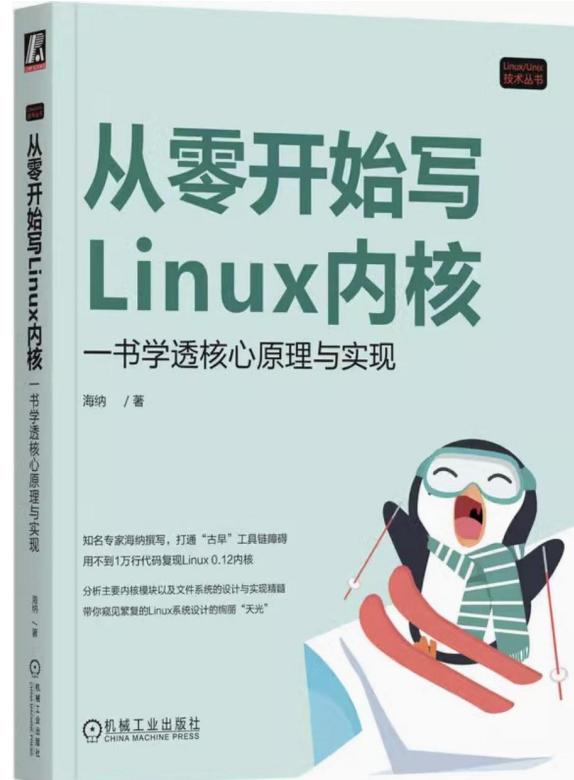
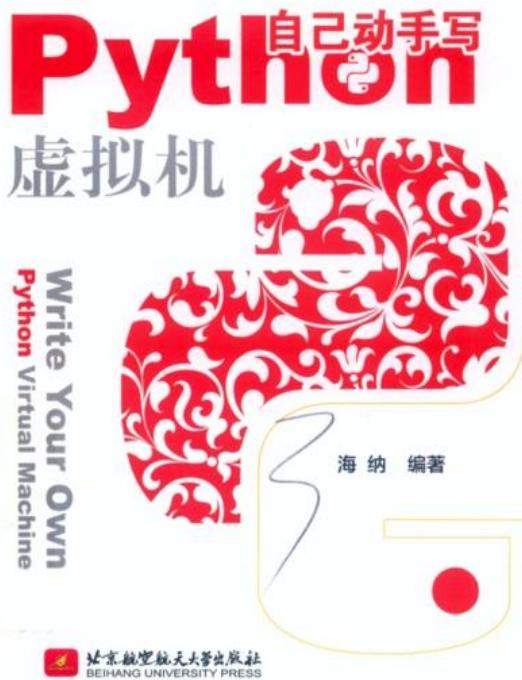


# GPU并发程序编译器

摩尔线程

海纳

# 个人简介



扫一扫上面的二维码图案，加我为朋友。

# 目录

- 并行程序和GPU架构
- Musa C程序的编译器
- AI Compiler
  - MLIR
  - Triton

# 摩尔定律的失效

- 摩尔定律：当价格不变时，单个芯片上可容纳的元器件的数目，约每隔18-24个月便会增加一倍，性能也将提升一倍。
  - 摩尔定律正接近基本的物理极限。随着芯片集成度越来越高，晶体管尺寸越来越小。但是物理元件不可能无限缩小。
  - 为了获得更高的计算能力，人们转向了并发编程
- 
- 课外阅读：<https://zhuanlan.zhihu.com/p/619518496>

# 程序的并行化

- 随着摩尔定律的放缓，单纯依靠晶体管数量增长来提升性能变得困难，因此业界转向多核处理器和并行计算架构。这使得开发高效的并行程序成为继续提升计算性能的关键途径。
- 从频率提升转向多核（并行）：当单纯通过提高处理器时钟频率来提升性能变得不可行（因为功耗和散热问题）时，芯片制造商（如Intel、AMD）转向了在单个芯片上集成多个处理核心。这标志着计算从单核串行向多核并行的转变。因此，编写能够利用多个核心的并行程序变得至关重要。

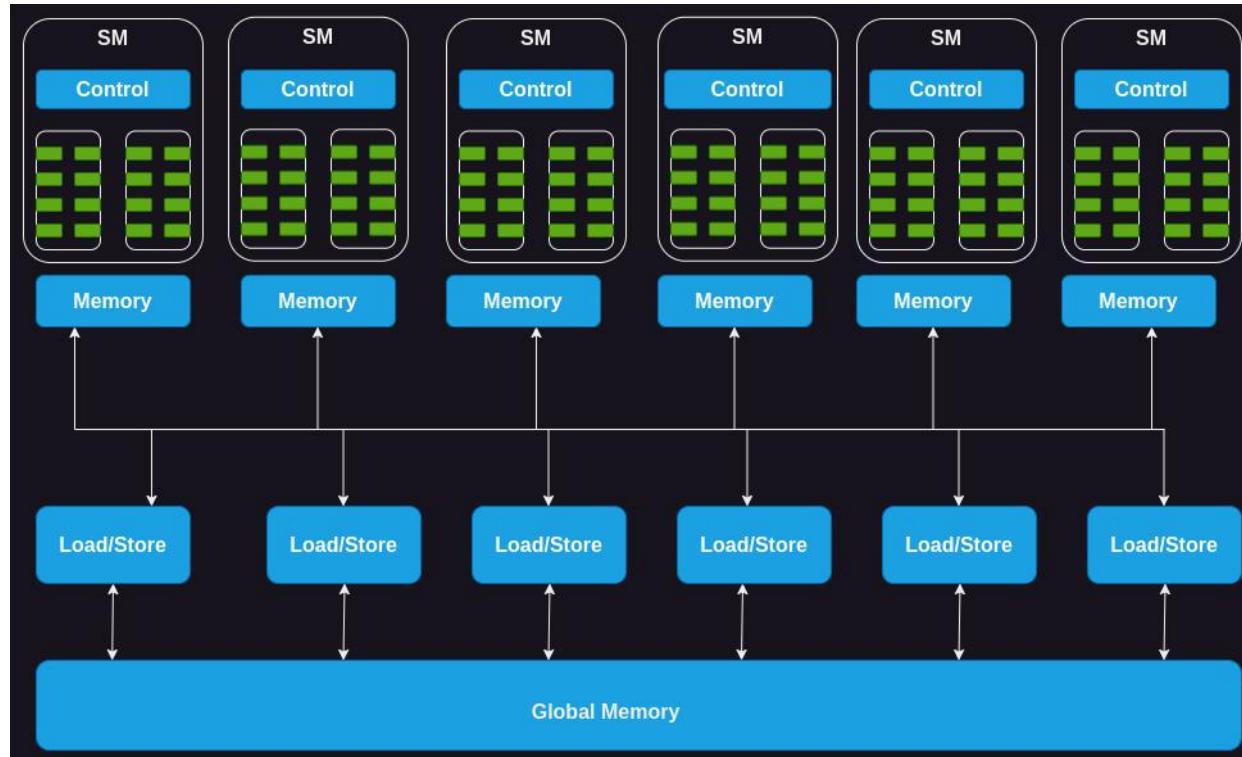
# SIMD vs. SIMT

比较项	SIMD	SIMT
执行方式	一条指令同时操作一个向量中的多个数据元素。	一条指令被多个独立的线程同时执行，每个线程处理自己的私有数据。
硬件结构	<ul style="list-style-type: none"> <li>单个控制单元（CU）</li> <li>宽向量寄存器（如128-bit, 256-bit, 512-bit）</li> <li>多个处理单元（PU）共享一个CU</li> <li>所有PU在CU指挥下同步执行</li> </ul>	<ul style="list-style-type: none"> <li>多核系统（如GPU的流多处理器SM）</li> <li>每个核心有独立的寄存器文件（RF）和计算单元（ALU）</li> <li>统一的指令缓存和解码器，指令广播给所有核心</li> <li>Warp/Wavefront调度器管理线程组</li> </ul>
编程模型	显式并行：程序员需直接操作向量寄存器，使用特定指令（Intrinsics）或依赖编译器自动向量化。需要手动处理数据对齐、打包和分支掩码。	隐式并行：程序员编写描述单个线程行为的标量代码（Kernel）。硬件负责创建、调度成千上万个线程，并将它们组织成Warp进行SIMD-like执行。
条件分支处理	<p>非常困难且低效：</p> <ul style="list-style-type: none"> <li>必须使用掩码（Predication）技术，将if-else转换为向量操作。</li> <li>或串行化执行不同分支路径，导致性能大幅下降。</li> <li>是SIMD的主要性能瓶颈。</li> </ul>	<p>硬件原生支持但有代价：</p> <ul style="list-style-type: none"> <li>可直接使用if/else等标准分支语句。</li> <li>当Warp内线程走向不同分支时，发生分支发散（Branch Divergence），硬件会串行执行各路径，禁用不执行的线程。</li> <li>性能损失严重，优化关键在于避免Warp内的发散。</li> </ul>
优势	<ul style="list-style-type: none"> <li>低延迟，与CPU核心紧耦合。</li> <li>在规则数据、密集计算、简单分支的循环中效率极高。</li> <li>广泛存在于CPU、DSP等多种处理器中。</li> </ul>	<ul style="list-style-type: none"> <li>极致吞吐量，可支持数千至上万线程并行。</li> <li>能有效隐藏延迟（通过Warp切换）。</li> <li>编程模型友好，接近标准多线程。</li> <li>支持复杂的内存层次（共享内存、高速缓存）。</li> <li>易于集成专用硬件（如Tensor Core）。</li> </ul>
劣势	<ul style="list-style-type: none"> <li>编程复杂，需深入了解硬件。</li> <li>灵活性差，难以处理不规则数据和复杂分支。</li> <li>并行度受限于向量宽度。</li> </ul>	<ul style="list-style-type: none"> <li>启动和数据传输开销大，不适合小任务。</li> <li>分支发散是主要性能杀手。</li> <li>单线程延迟高。</li> <li>内存访问模式对性能极其敏感。</li> </ul>

# 基本术语

名称	英文名称	详细解释
网格	Grid	核函数（Kernel）启动时创建的所有线程的集合
线程块	Thread Block	Thread Block 是 Grid 的子集，是程序员可以直接管理的最小调度单元。它是逻辑上的分组。
线程束	Warp	Warp 是 GPU 硬件执行的基本单位。它是物理层面的调度和执行单元。
流式多处理器	Stream Multi-processor, SM	GPU 的核心处理单元，可以看作是 GPU 内部的一个“小型多核 CPU”
流式处理器	Stream Processor, SP	SP 是 SM 内部的基本计算核心，也常被称为 Shader Core。它是真正执行算术和逻辑运算（如加法、乘法）的物理单元。
共享内存	Shared Memory	共享内存是位于 SM 内部的一块高速、可软件管理的 SRAM。它是 SM 提供给其上运行的线程块使用的共享存储资源。

# GPU的基本架构



1. GPU由多个SM组成
2. 每个SM由多个SP组成
3. 开发者通过Grid定义任务
4. Grid由多个Thread Block组成
5. 一个Thread Block会被调度到一个SM上执行
6. 调度时Thread Block会被进一步拆分成Warp
7. SM执行时以Warp为最小调度单位
8. shared memory 为一个Thread Block共享
9. 寄存器是每个线程独占的

# AI计算中的编译技术

- C++扩展使用Clang + LLVM（例如CUDA, MUSA）
- 虚拟指令集（PTX, MTX）
- 自动融合框架（IREE, MLIR-based, TVM）
- 算子自动生成（AKG, Inductor）
- 算子开发DSL（Tilelang, Triton）

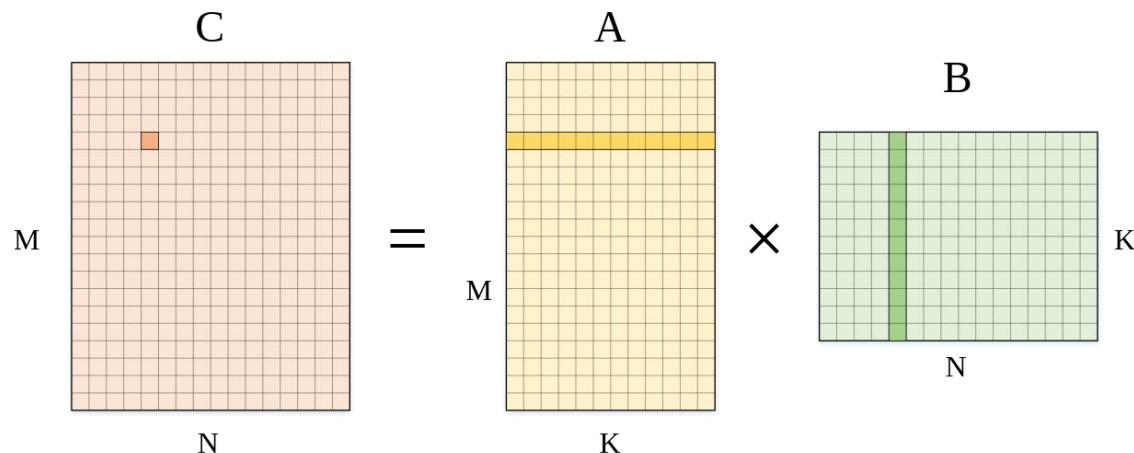
# GPU体系结构及编程优化

# 背景知识

- 通用矩阵乘法 (GEMM)

$$\mathbf{C} = \mathbf{AB}; \mathbf{A}, \mathbf{B}, \mathbf{C} \in R^{n \times n}$$

$$C_{m,n} = \sum_{k=1}^K A_{m,k} B_{k,n}; m, n, k \in R^n$$



- 通用矩阵乘法的CPU C代码实现

```

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

- 计算量

最内层循环体 1 次迭代有 1 次乘法，1 次加法，共 2 次浮点运算。

$$\text{总计算量} = 2 \times M \times N \times K \text{ (FLOP)}$$

- 访存量

假设矩阵元素没有缓存，每次访问都从主存读写。最内层 2 次读，内层结束 1 次写  $\mathbf{C}$ 。

$$\text{总内存访问} = 2MNK + MN \text{ (以元素为单位)}$$

# 背景知识

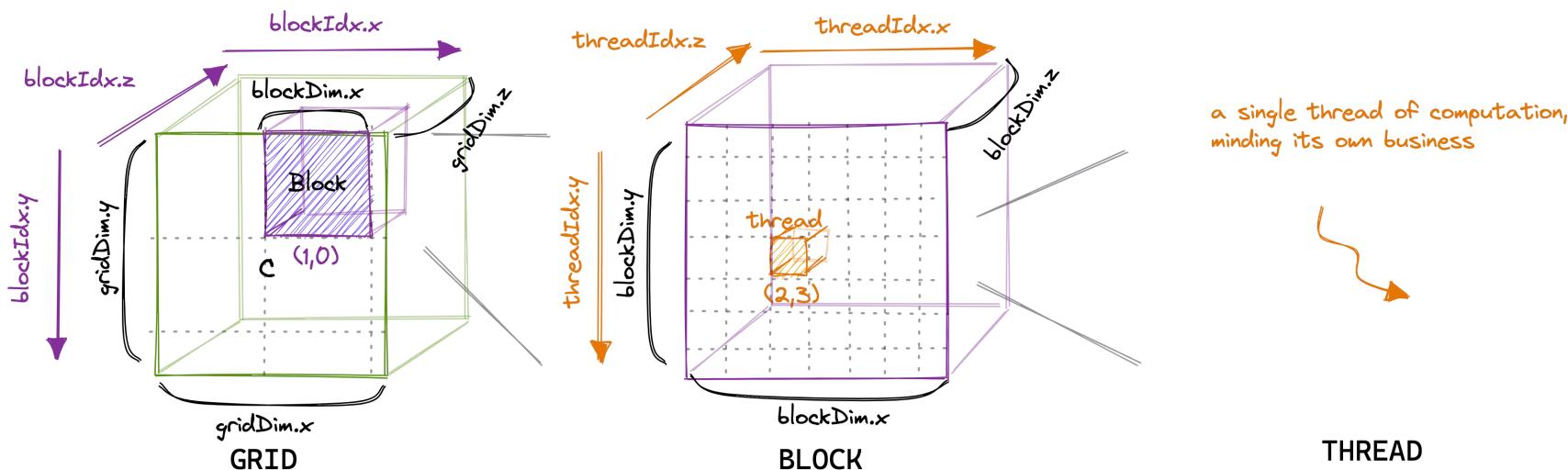
- MUSA编程模型

MUSA编程模型的三级层次结构: grid, block, thread。每次调用MUSA内核都会创建一个新的网格(grid), 该网格由多个块(block)组成。每个块由blockDim (含x、y、z三个维度) 指定线程数量最多可以包含1024个独立线程(thread)。网格规模由gridDim控制。

```

1 // create as many blocks as necessary to map all of C
2 dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32), 1);
3 // 32 * 32 = 1024 thread per block
4 dim3 blockDim(32, 32, 1);
5 // launch the asynchronous execution of the kernel on the device
6 kernel<<<gridDim, blockDim>>>(M, N, K, A, B, C);

```



# Kernel-1 (Naive)

- GPU上的矩阵乘法几乎占据了模型训练和推理过程中绝大部分的FLOPs。编写一个高性能的 SGEMM (Single precision General Matrix Multiplication) 是重要的，其执行 $C = \alpha AB + \beta C$ ，其中C/A/B都是矩阵， $\alpha/\beta$ 是标量。
- 接下来从一个简单的kernel开始，逐步优化。

```

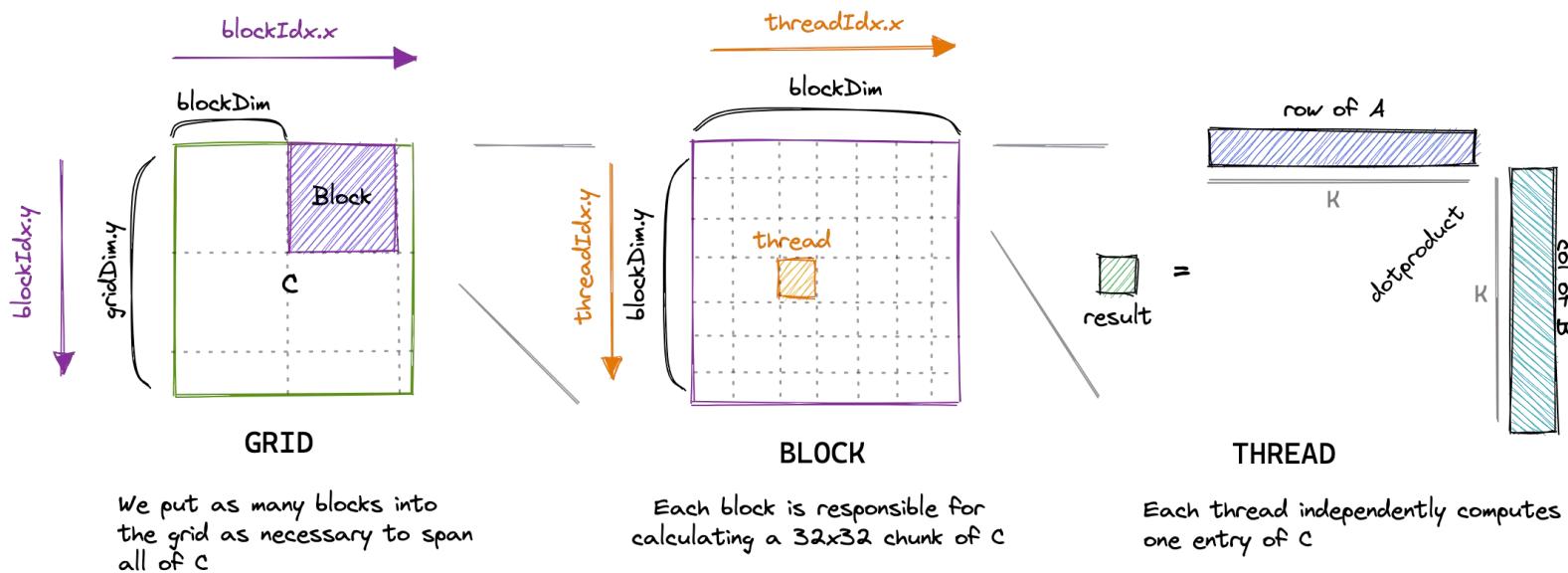
1 dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32), 1);
2 dim3 blockDim(32, 32, 1);
3 // launch the asynchronous execution of the kernel on the device
4 sgemm_naive<<<gridDim, blockDim>>>(M, N, K, alpha, A, B, beta, C);
5
6 __global__ void sgemm_naive(int M, int N, int K, float alpha, const float *A,
7                             const float *B, float beta, float *C) {
8     // compute position in C that this thread is responsible for
9     const uint x = blockIdx.x * blockDim.x + threadIdx.x;
10    const uint y = blockIdx.y * blockDim.y + threadIdx.y;
11
12    // `if` condition is necessary for when M or N aren't multiples of 32.
13    if (x < M && y < N) {
14        float tmp = 0.0;
15        for (int i = 0; i < K; ++i) {
16            tmp += A[x * K + i] * B[i * N + y];
17        }
18        // C = α*(A@B)+β*C
19        C[x * N + y] = alpha * tmp + beta * C[x * N + y];
20    }
21 }
```

- MUSA内核以单线程视角编写，通过内置变量 `threadIdx` 和 `blockIdx` 区分不同线程。  
`threadIdx.x/y` 在块内从0到31变化，`blockIdx.x/y` 在网格内从0到`CEIL_DIV(N,32)`或`CEIL_DIV(M,32)`变化，共同确定线程的唯一位置。

# Kernel-1 (Naive)

- sgemm\_naive kernel

在某实验虚拟GPU上（峰值计算性能30TFLOPs/s，全局内存带宽768GB/s），假设A/B/C大小均 $4092^2$ （ $4092 \times 4092$ ），元素类型为fp32，这个矩阵乘法大约需要 0.5 秒。



运算  $C = \alpha AB + \beta C$  分析：

计算量：137 GFLOPs

最小数据移动：268 MB

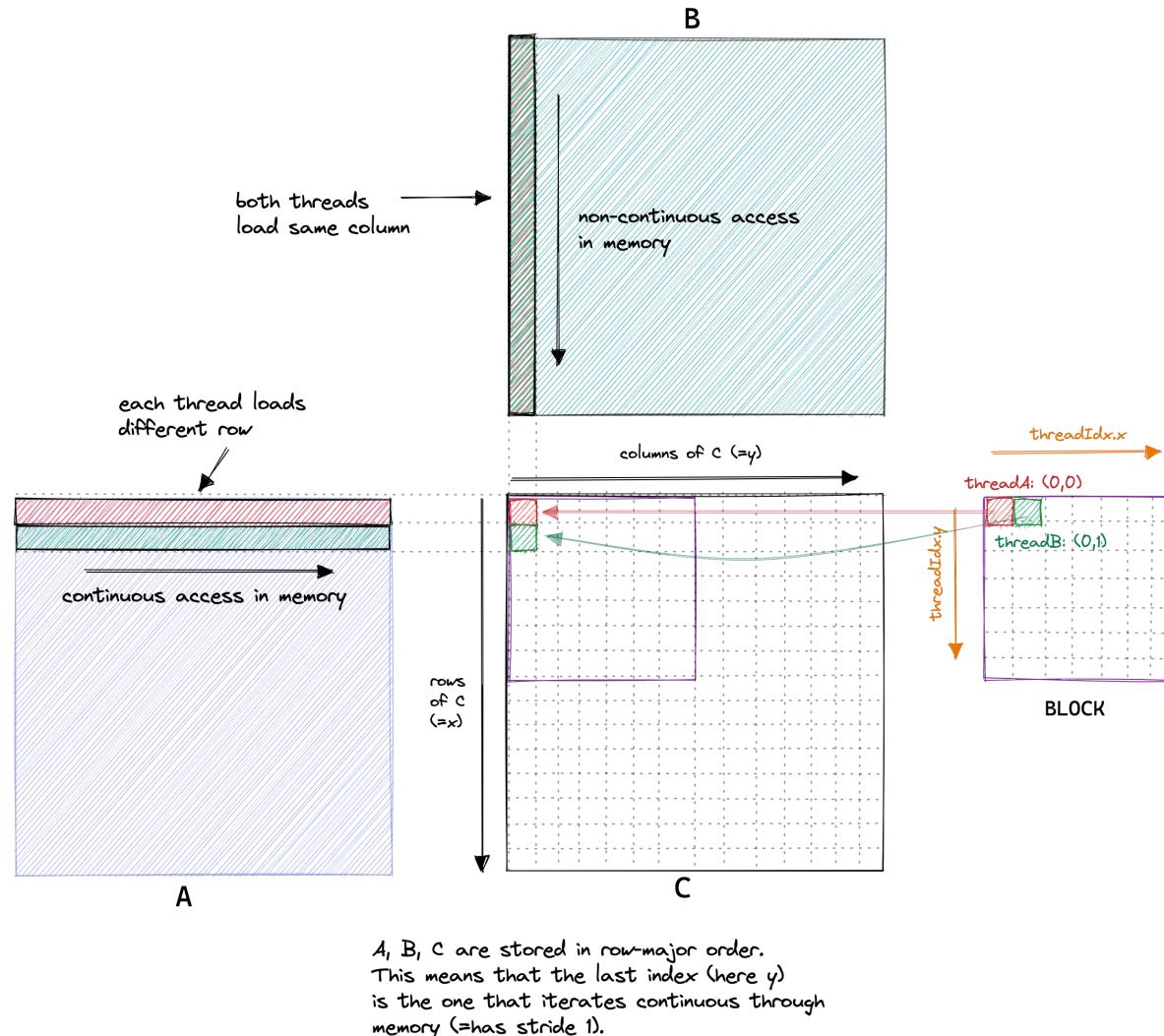
硬件峰值：算力30 TFLOPS，带宽768 GB/s

理论耗时：计算4.5ms，内存访问0.34ms

只要实际数据移动量 $< 2.68$  GB，该问题即为计算受限。

# Kernel-1 (Naive)

- 访存模式分析



naive实现中

- 每个线程需加载  $2 \times 4092 + 1$  个浮点数，共  $4092^2$  个线程，总内存加载量达 548GB。
- 线程间大量重复加载数据（如 B 矩阵的同一列），导致内存访问效率低下。

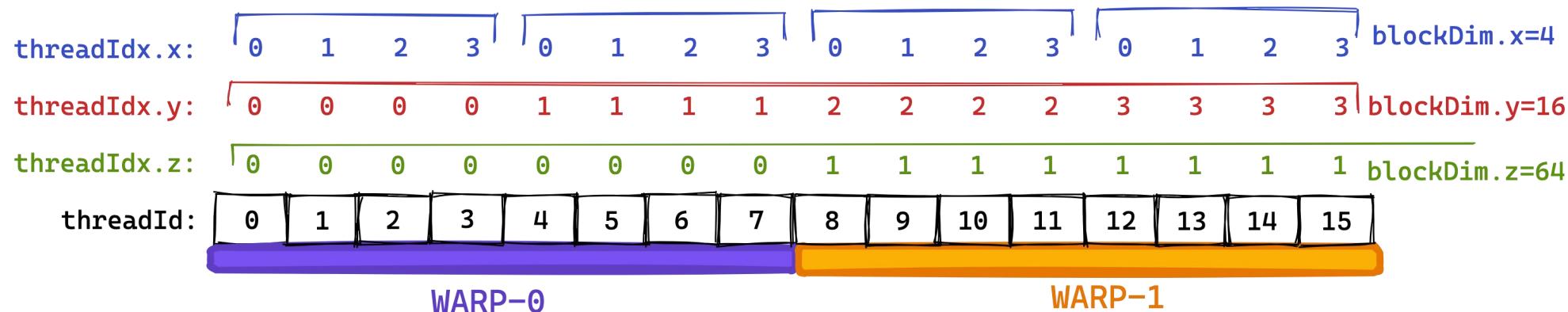
naive在xGPU上运行

- 实测性能：约 300 GFLOPs/s
- 理论峰值性能：30 TFLOPs/s
- 利用率：仅 1%

表现较差，存在较大优化空间。

# Kernel-2 (全局访存合并, GMEM Coalescing)

- MUSA线程组织
- 线程块被分为多个warp，每个warp包含32个连续线程
- warp是基本的执行单元，分配给warp调度器
- 线程ID计算：  $\text{threadId} = \text{threadIdx.x} + \text{blockDim.x} * (\text{threadIdx.y} + \text{blockDim.y} * \text{threadIdx.z})$
- 相邻threadId的线程属于同一 warp，其顺序内存访问可以被合并，这被称为全局内存合并访问



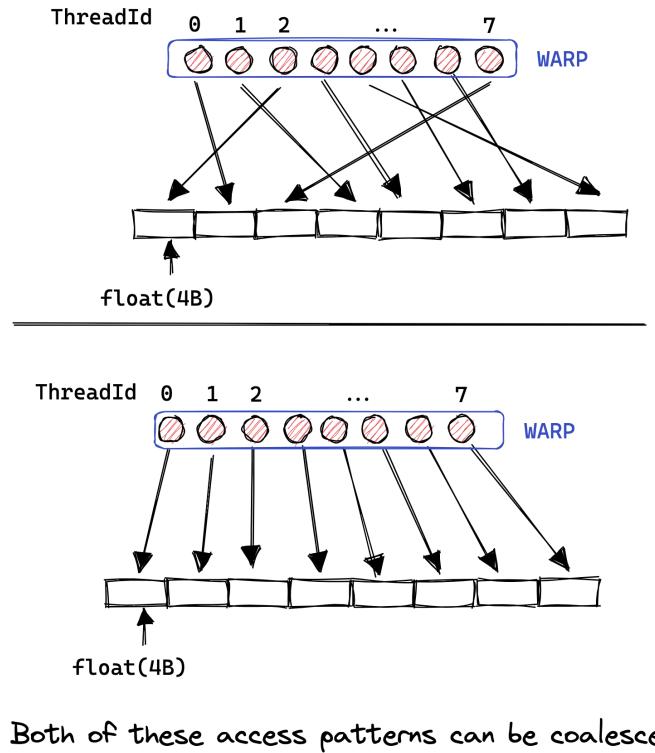
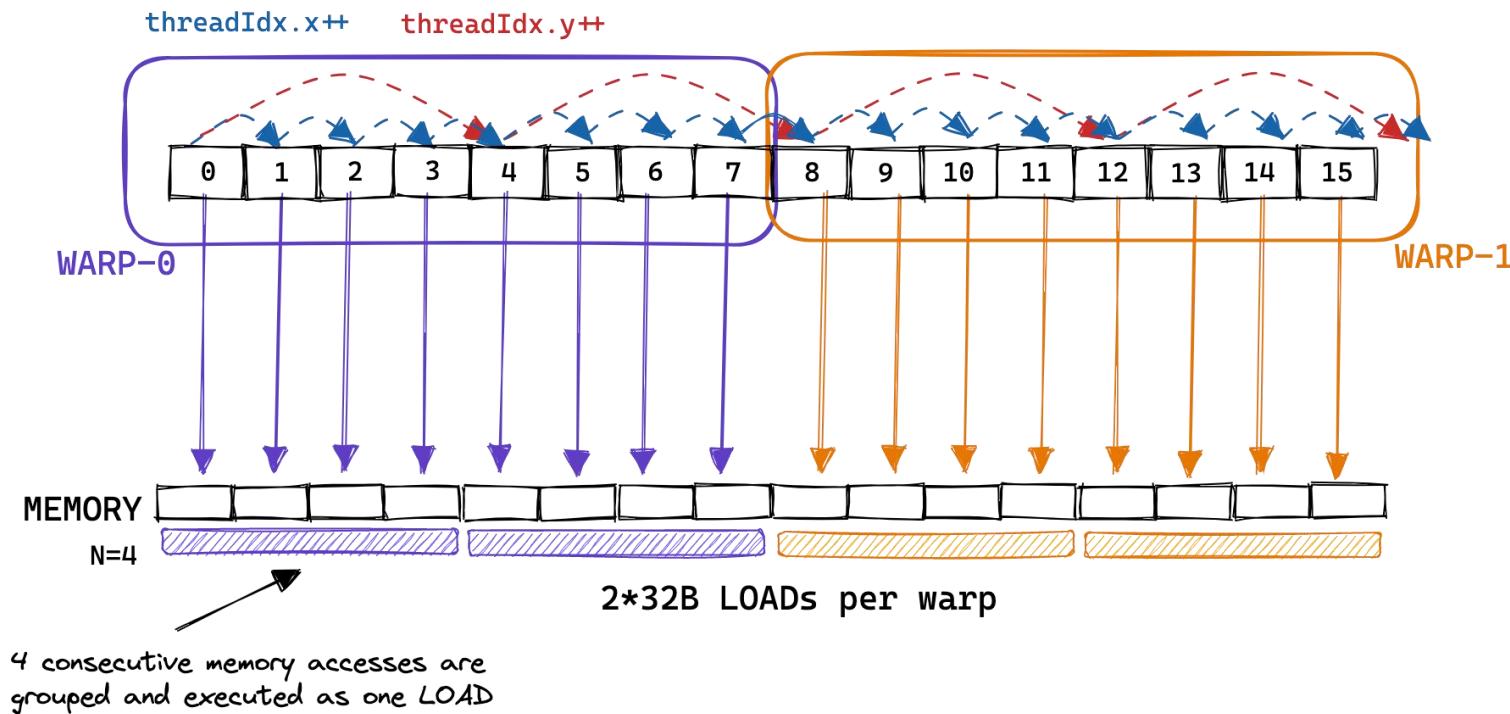
$\text{threadId} = \text{threadIdx.x} + \text{blockDim.x} * \text{threadIdx.y} + \text{blockDim.x} * \text{blockDim.y} * \text{threadIdx.z}$

(示例使用warp大小=8仅为说明，实际均为32线程)

# Kernel-2 (全局访存合并, GMEM Coalescing)

- 全局访存合并
  - 属于同一warp的线程的顺序内存访问可以被合并。
  - 假设warp内有8个线程，每个线程访问不同地址的8B数据，需要8次内存访问。
  - 但经过访问合并后，整个warp 仅使用 2 次内存访问（每次访问32B）

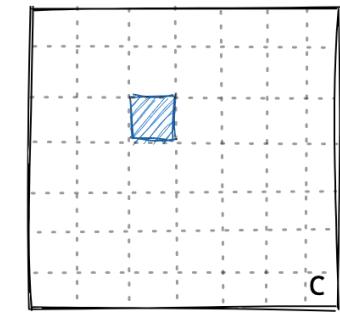
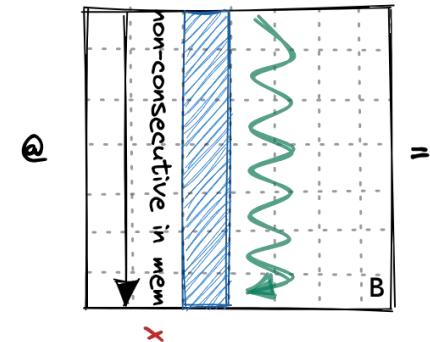
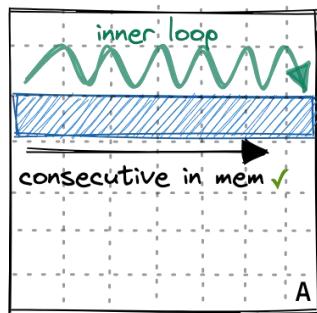
合并访问要求：warp内所有线程访问的内存地址整体构成一个连续区域，但相邻线程的地址不必连续。



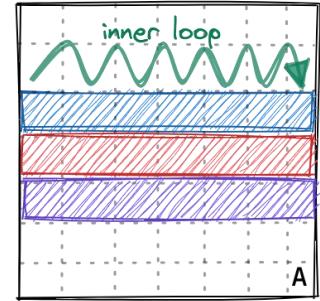
# Kernel-2 (全局访存合并, GMEM Coalescing)

为了达到合并访问的效果，我们可以改变矩阵C元素分配给线程的方式。

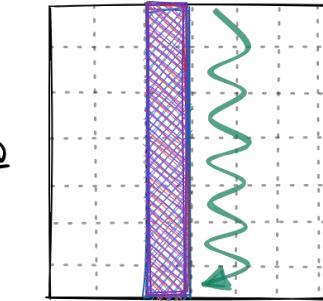
Matrix memory layout:



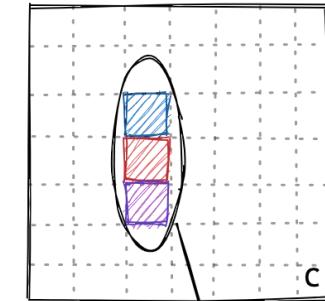
Naive kernel:



threads access non-consecutive values ⇒ cannot coalesce

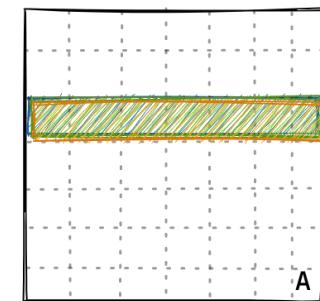


all threads access same values ⇒ within-warp broadcast

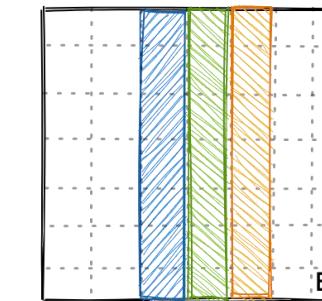


No benefit to putting these threads in same warp

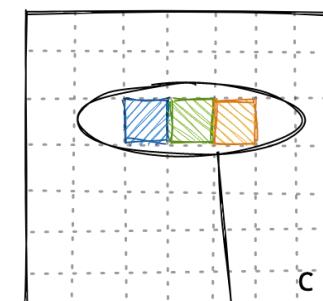
Coalescing kernel:



all threads access same values ⇒ within-warp broadcast



threads access consecutive values ⇒ can coalesce



Make sure these threads end up in same warp to exploit coalescing

# Kernel-2 (全局访存合并， GMEM Coalescing)

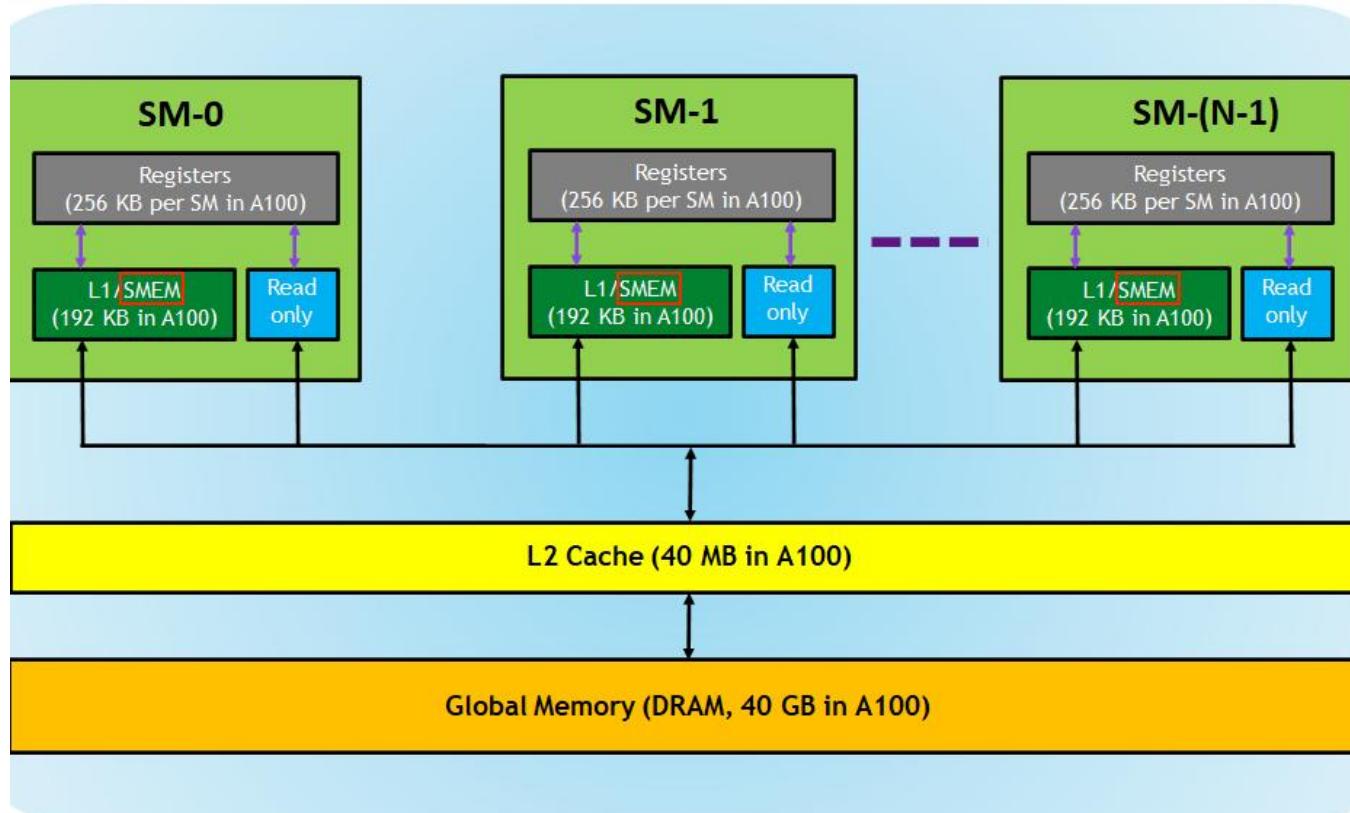
- 改进的kernel

```
1 // improved kernel
2 const int x = blockIdx.x * BLOCKSIZE + (threadIdx.x / BLOCKSIZE);
3 const int y = blockIdx.y * BLOCKSIZE + (threadIdx.x % BLOCKSIZE);
4
5 if (x < M && y < N) {
6     float tmp = 0.0;
7     for (int i = 0; i < K; ++i) {
8         tmp += A[x * K + i] * B[i * N + y];
9     }
10    C[x * N + y] = alpha * tmp + beta * C[x * N + y];
11 }
12
13 // blockDim stays the same
14 dim3 gridDim(CEIL_DIV(M, 32), CEIL_DIV(N, 32));
15 // make blockDim 1-dimensional, but don't change number of threads
16 dim3 blockDim(32 * 32);
17 sgemm_coalescing<<<gridDim, blockDim>>>(M, N, K, alpha, A, B, beta, C);
```

(blockDim使用一维的BLOCKSIZE表示，并且BLOCKSIZE=32，gridDim仍然使用二维)

在新的实现中，全局内存的合并访问将吞吐量从15GB/s提高到110GB/s。性能达到2000 GFLOPs/s，相较于第一个kernel的300 GFLOPs/s有了很大的改善。

# Kernel-3 (利用共享内存, SMEM Caching)

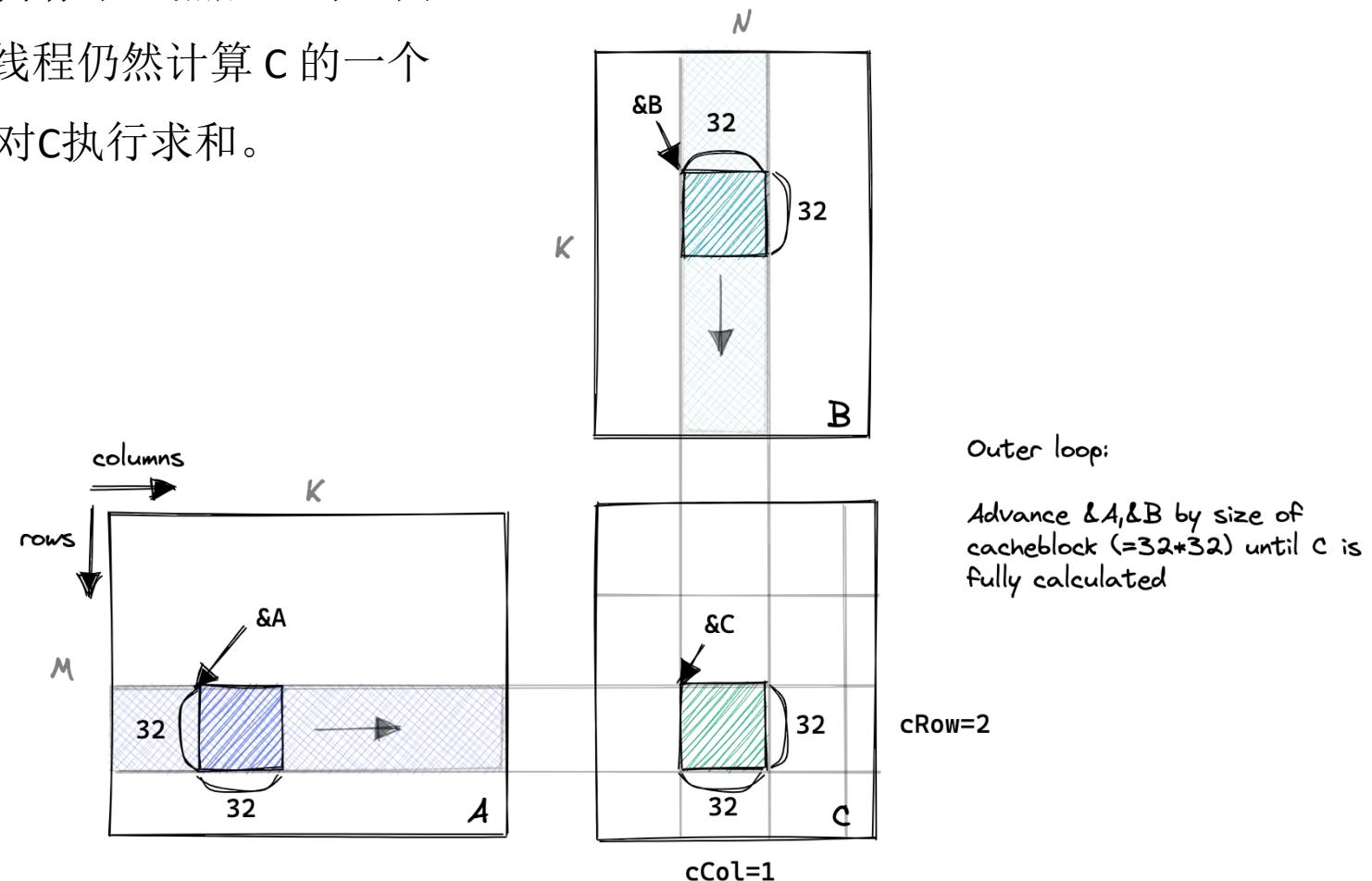


(A100 GPU 内存层次结构展示的共享内存)

- 共享内存（SMEM）是位于GPU上的相比全局内存要小得多的一块内存区域。每个SM有一块共享内存区域。
- 同一个块内的线程可以通过共享内存进行通信。在xGPU上，每个块最多可以访问 48KB 的共享内存。
- 全局内存带宽 750GiB/s，共享内存带宽 12080GiB/s，共享内存是全局内存的约16倍。
- L1缓存和SMEM共享同一块片上内存。通过配置使用更小的L1缓存来换取使用更大的SMEM，其数据一致性需要显式同步。

# Kernel-3 (利用共享内存, SMEM Caching)

我们将A和B的小块CHUNK数据加载到共享内存中。然后，对这两块CHUNK数据执行尽可能多的工作，每个线程仍然计算C的一个元素。沿着A的列和B的行移动CHUNK块，对C执行求和。



# Kernel-3 (利用共享内存, SMEM Caching)

```

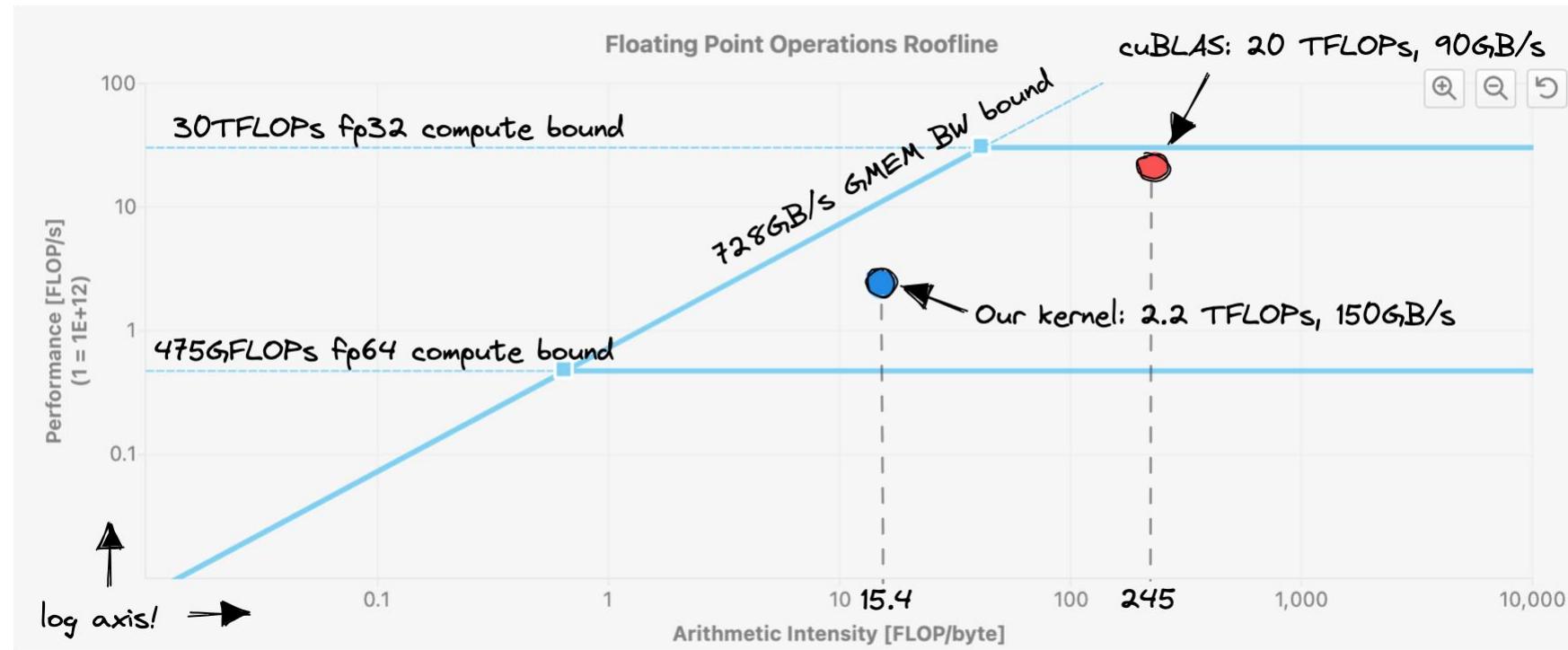
1 // advance pointers to the starting positions
2 A += cRow * BLOCKSIZE * K;                                // row=cRow, col=0
3 B += cCol * BLOCKSIZE;                                     // row=0, col=cCol
4 C += cRow * BLOCKSIZE * N + cCol * BLOCKSIZE; // row=cRow, col=cCol
5 float tmp = 0.0;
6 // the outer loop advances A along the columns and B along
7 // the rows until we have fully calculated the result in C.
8 for (int bkIdx = 0; bkIdx < K; bkIdx += BLOCKSIZE) {
9     // Have each thread load one of the elements in A & B from
10    // global memory into shared memory.
11    // Make the threadCol (=threadIdx.x) the consecutive index
12    // to allow global memory access coalescing
13    As[threadRow * BLOCKSIZE + threadCol] = A[threadRow * K + threadCol];
14    Bs[threadRow * BLOCKSIZE + threadCol] = B[threadRow * N + threadCol];
15    // block threads in this block until cache is fully populated
16    __syncthreads();
17    // advance pointers onto next chunk
18    A += BLOCKSIZE;
19    B += BLOCKSIZE * N;
20    // execute the dotproduct on the currently cached block
21    for (int dotIdx = 0; dotIdx < BLOCKSIZE; ++dotIdx) {
22        tmp += As[threadRow * BLOCKSIZE + dotIdx] *
23            Bs[dotIdx * BLOCKSIZE + threadCol];
24    }
25    // need to sync again at the end, to avoid faster threads
26    // fetching the next block into the cache before slower threads are done
27    __syncthreads();
28 }
29 C[threadRow * N + threadCol] = alpha * tmp + beta * C[threadRow * N + threadCol];

```

- 代码中，每个block仍然负责计算C矩阵的32x32的块，每个线程负责其中一个元素的计算。
- 在外层的bkIdx循环中，先从A和B中加载32x32的小块CHUNK数据到共享内存As和Bs中。
- 内层的dotIdx循环中，一个线程计算32x32中的一个元素。

# Kernel-3 (利用共享内存, SMEM Caching)

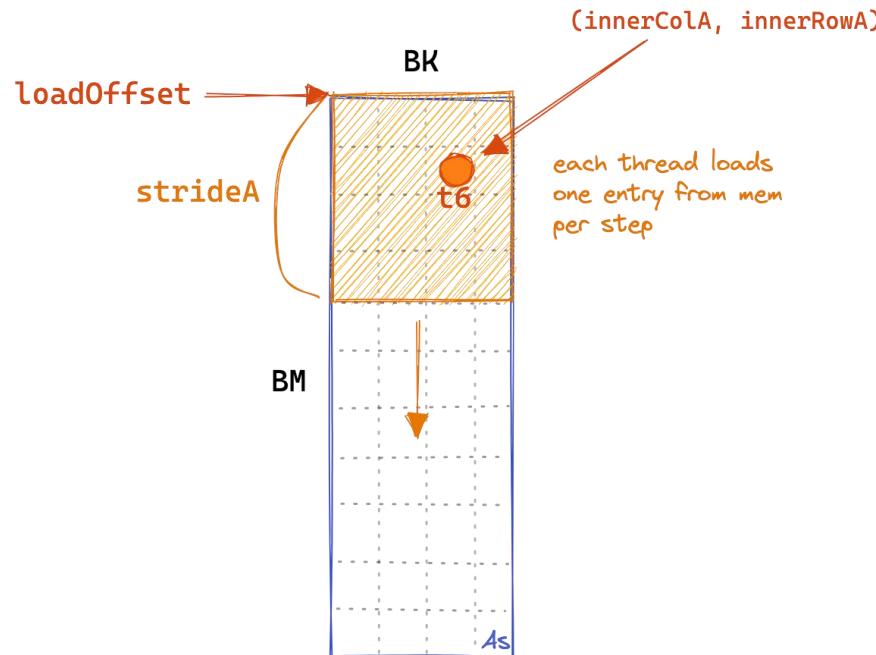
Kernel-3的性能达到了~2200 GFLOPs/s，比kernel-2的版本提高了10%（kernel-2为2000GFLOPs/s）。距离 GPU 能够提供的约 30 TFLOPs 还很远。从下面的屋顶线图中可以看出这一点。



# Kernel-4 (每个线程计算8x8区域的元素， 2D-Blocktiling)

Kernel-4的基本思想是每个线程计算 C 中一个  $8 \times 8$  区域的元素。内核的第一阶段是让所有线程共同填充共享内存。

从全局内存加载数据到共享内存的过程:



```

1 // populate the SMEM caches
2 for (uint loadOffset = 0; loadOffset < BM; loadOffset += strideA) {
3     As[(innerRowA + loadOffset) * BK + innerColA] =
4         A[(innerRowA + loadOffset) * K + innerColA];
5 }
6 for (uint loadOffset = 0; loadOffset < BK; loadOffset += strideB) {
7     Bs[(innerRowB + loadOffset) * BN + innerColB] =
8         B[(innerRowB + loadOffset) * N + innerColB];
9 }
10 __syncthreads();

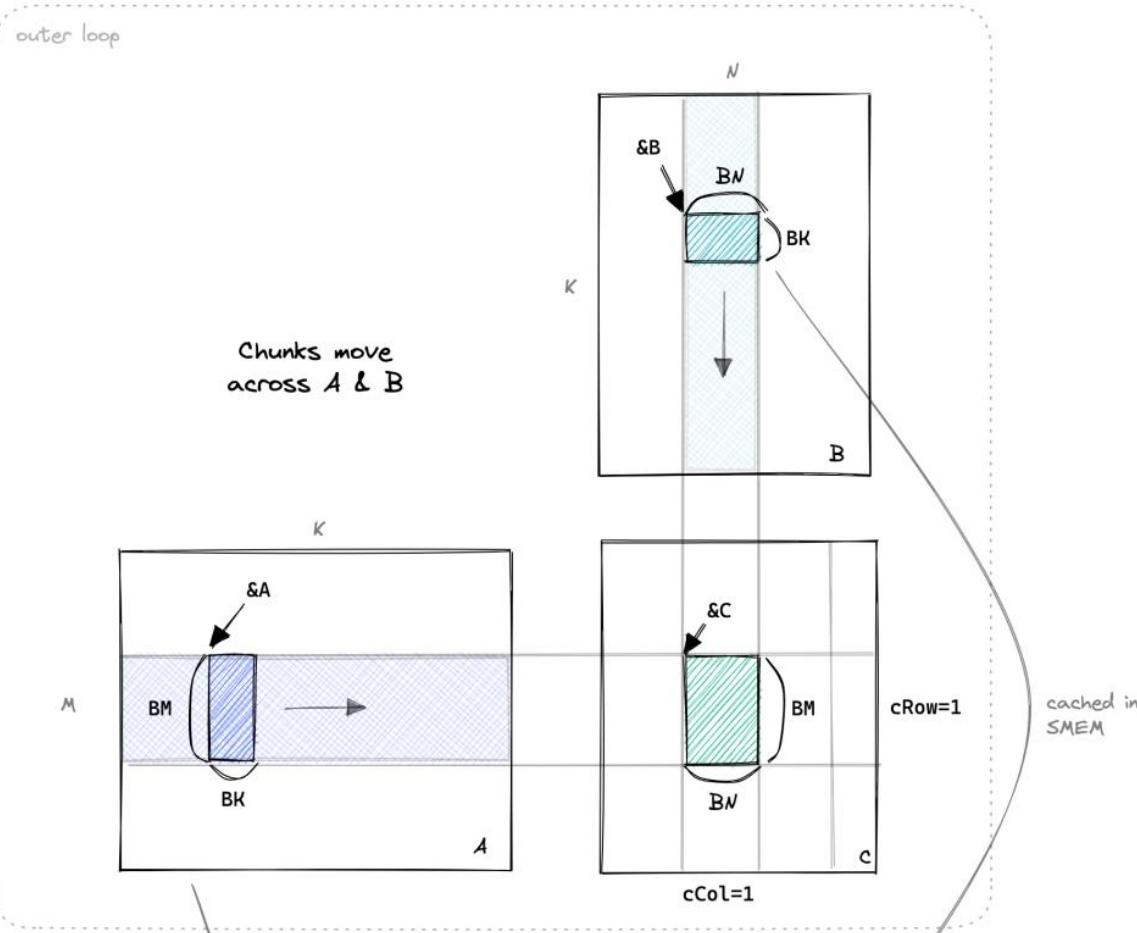
```

# Kernel-4 (每个线程计算8x8区域的元素， 2D-Blocktiling)

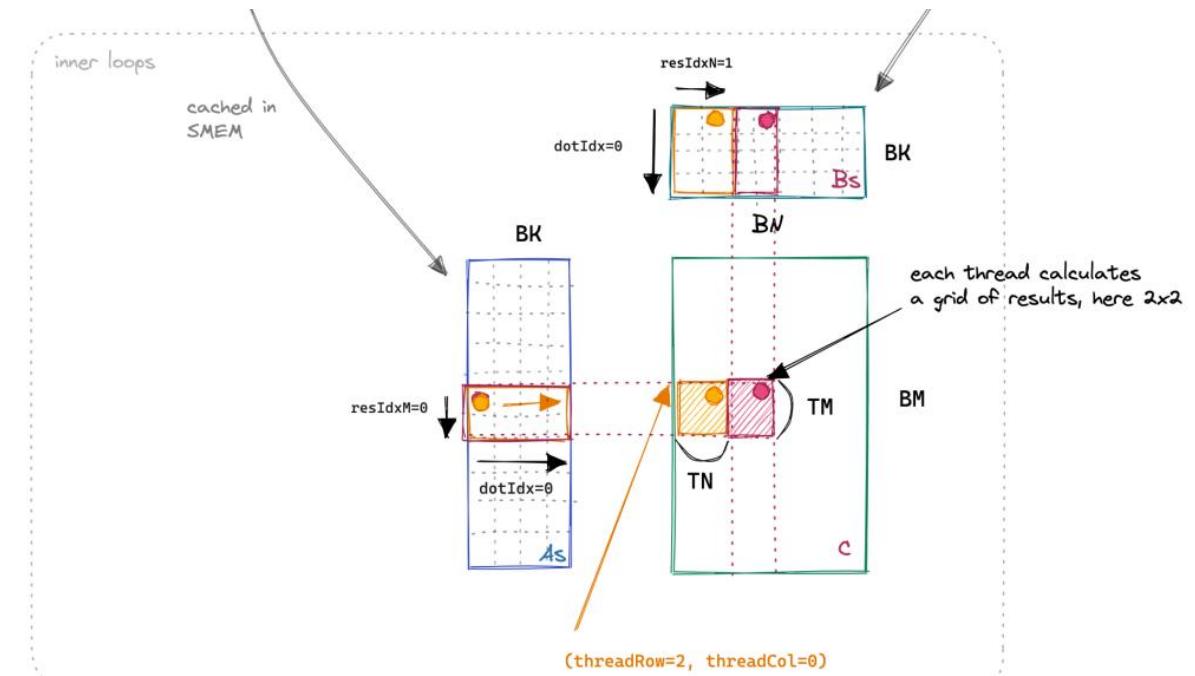
每个线程将相关的共享内存元素相乘，并将结果累积到本地寄存器中。

第一个外循环沿着BK方向，第二个和第三个循环沿着TM和TN进行点积运算：

外循环



内循环

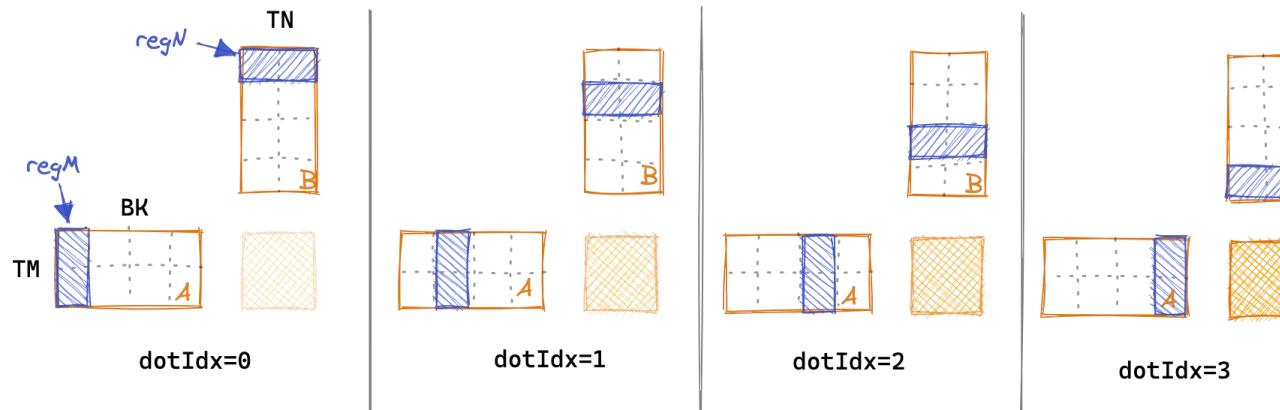


# Kernel-4 (每个线程计算8x8区域的元素， 2D-Blocktiling)

可以将两个内循环所需的值加载到寄存器中来减少共享内存的访问次数。

`dotIdx`随时间变化的循环图，显示了在每个步骤中哪些共享内存的元素被加载到线程本地的寄存器中：

Unrolled `dotIdx` loop:



at each timestep, load the 4 relevant As&Bs entries into `regM` and `regN` registers, and accumulate outer product into `threadResults`.

Benefit: We only issue 16 SMEM loads in total

```

1 // allocate thread-local cache for results in registerfile
2 float threadResults[TM * TN] = {0.0};
3 // register caches for As and Bs
4 float regM[TM] = {0.0};
5 float regN[TN] = {0.0};
6
7 // outer-most loop over block tiles
8 for (uint bkIdx = 0; bkIdx < K; bkIdx += BK) {
9     // populate the SMEM caches
10    // ...
11
12    A += BK;      // move BK columns to right
13    B += BK * N; // move BK rows down
14    for (uint dotIdx = 0; dotIdx < BK; ++dotIdx) {
15        // load relevant As & Bs entries into registers
16        for (uint i = 0; i < TM; ++i) {
17            regM[i] = As[(threadRow * TM + i) * BK + dotIdx];
18        }
19        for (uint i = 0; i < TN; ++i) {
20            regN[i] = Bs[dotIdx * BN + threadCol * TN + i];
21        }
22        // perform outer product on register cache,
23        // accumulate into threadResults
24        for (uint resIdxM = 0; resIdxM < TM; ++resIdxM) {
25            for (uint resIdxN = 0; resIdxN < TN; ++resIdxN) {
26                threadResults[resIdxM * TN + resIdxN] +=
27                    regM[resIdxM] * regN[resIdxN];
28            }
29        }
30    }
31    __syncthreads();
32 }
```

# Kernel-4 (每个线程计算8x8区域的元素， 2D-Blocktiling)

最终性能: 16TFLOPs/s。

假设BM=BN=128, BK=8, 每个线程计算 $TM * TN = 8 * 8 = 64$  个元素, 则有:

全局内存GMEM:  $K/8 * 2 * 1024/256 = K/64$ 。

$K/8$ 为外循环迭代次数。

2 表示需要分别加载A和B,

1024/256 表示 sizeSMEM/numThreads。

共享内存SMEM:  $K/8 * TM * TN * dotIdx * 2 = K/4$ 。

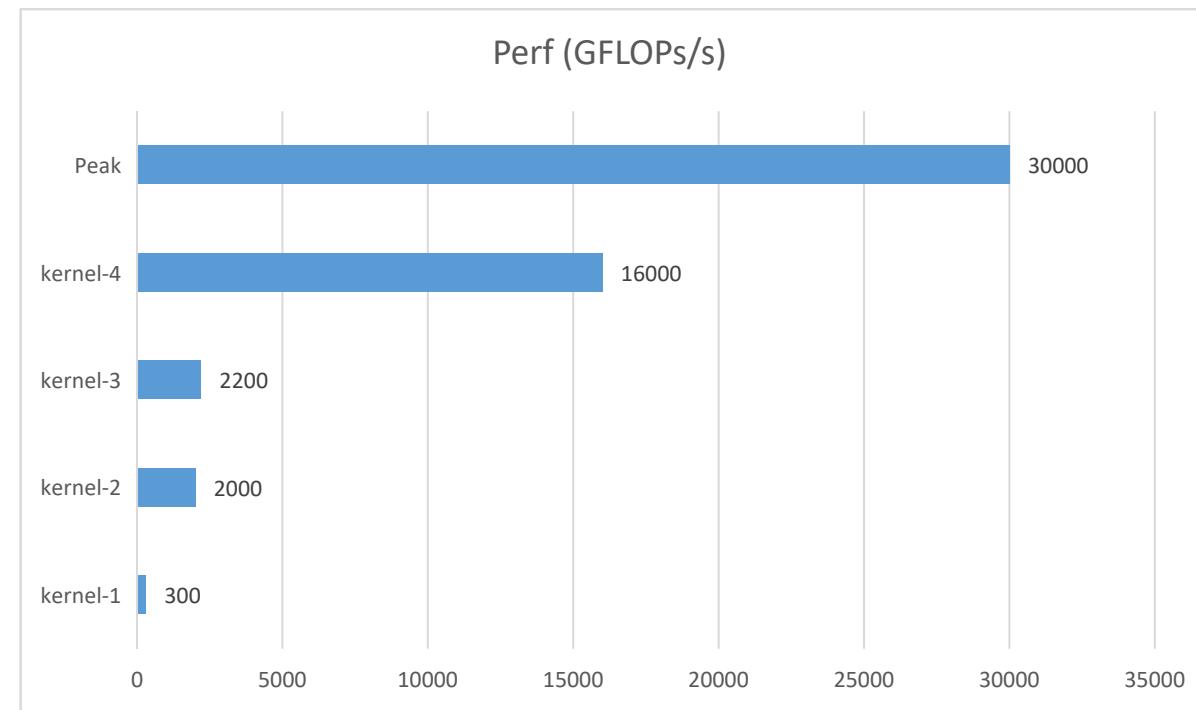
每个线程计算64个结果, 每个结果的内存访问次数:  $K/64$  GMEM、 $K/4$  SMEM。

# Kernel对比

我们从一个naive版本的CUDA矩阵乘法实现开始，逐步迭代优化该版本，不断提升性能。

汇总对比如下：

	Note	Perf (GFLOPs/s)	Improved
kernel-1	Naive	300	—
kernel-2	GMEM Coalescing	2000	567%
kernel-3	SMEM Caching	2200	633%
kernel-4	2D-Blocktiling	16000	5233%
Peak	Theoretical	30000	9900%



AI编译器

# MLIR 概述

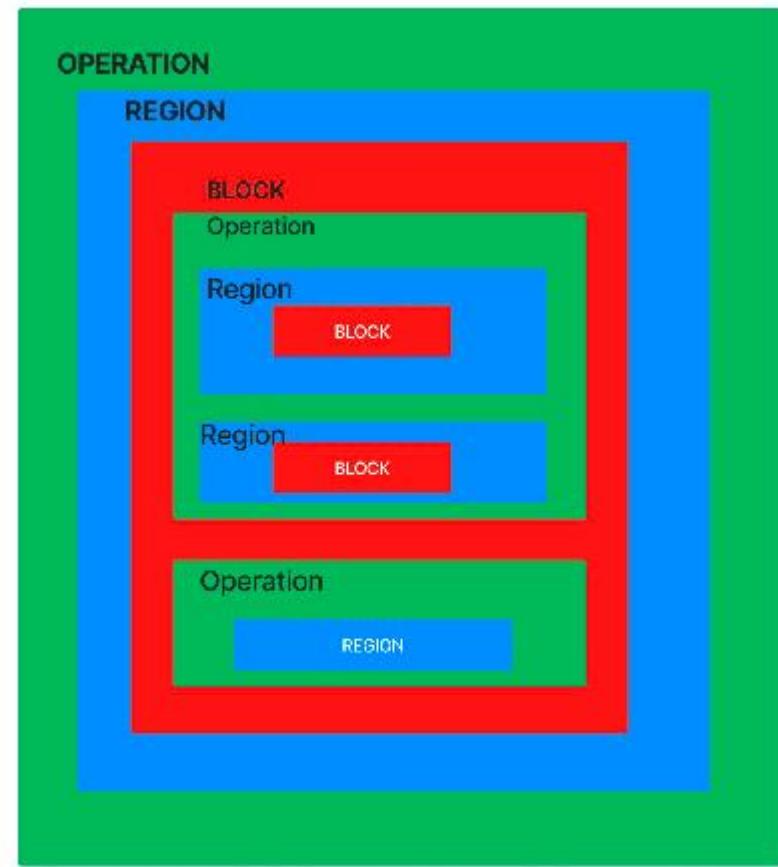
- MLIR is **Multi-Level Intermediate Representation**
- MLIR 表示 **MLIR Language (a hybrid IR)**
- MLIR 表示 **MLIR Framework (用于自定义 IR 和 Pass)**

# MLIR 概述

## MLIR Operation Spec

Number of Value returned	Dialect prefix	Op Id	Argument	Index in the producer's results	List of attributes: constant named arguments
<pre>%res: 2 = "mydialect.morph"(%input#5) {some.attribute=true, other_attribute=1.5}</pre>					
<pre>:(!mydialect&lt;"custom_type"&gt;)-&gt;(!mydialect&lt;"other_type"&gt;, !mydialect&lt;"other_type"&gt;)</pre>					
<pre>loc(callsite("foo" at "mysource.cc": 10: 8))</pre>					
Name of the results	Dialect prefix for the type	Opaque string/Dialect-specific type Location			

## MLIR Region/Block/Op 结构



# MLIR 概述

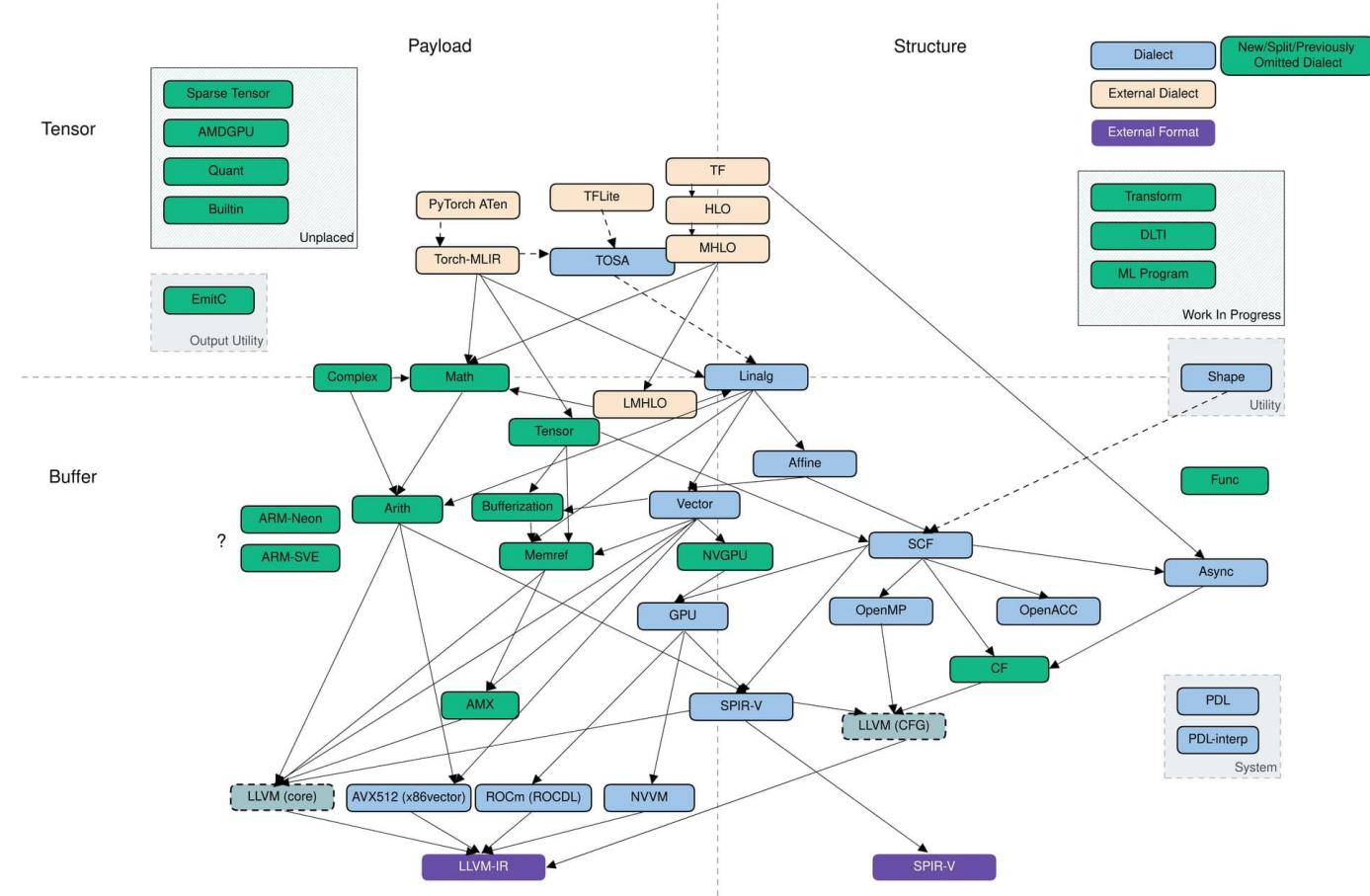
MLIR 实现了一套 IR 和 Pass 的基础设施

- Operation/Block/Region
- Type/Attribute
- Value: Operands/Results
- Analysis
- Pass Manager
- Conversion/Transformation
- Rewrite Pattern Engine

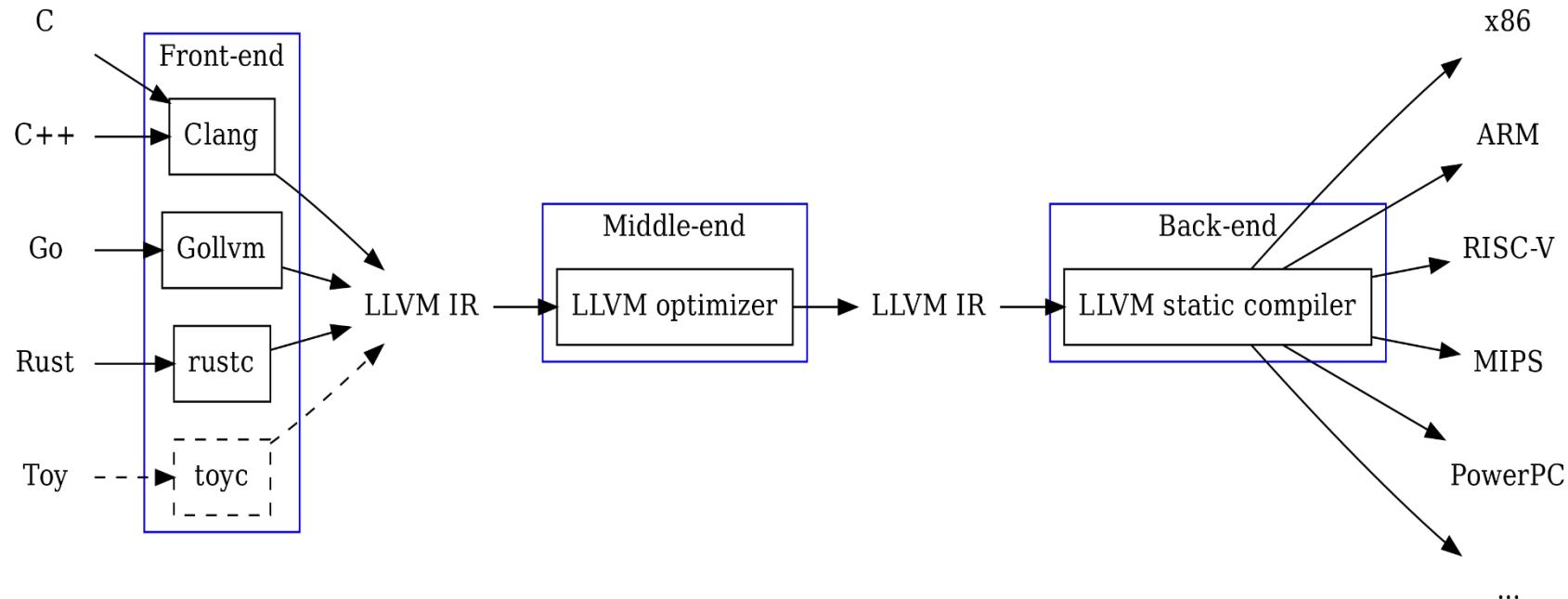
# MLIR 概述

MLIR 内置了许多 Dialect 和 Pass  
包括各种常用的优化算法：

CSE(公共子表达式消除), DCS(死代码消除), LICM(循环不变代码外提), Affine 相关优化, Buffer 相关优化...

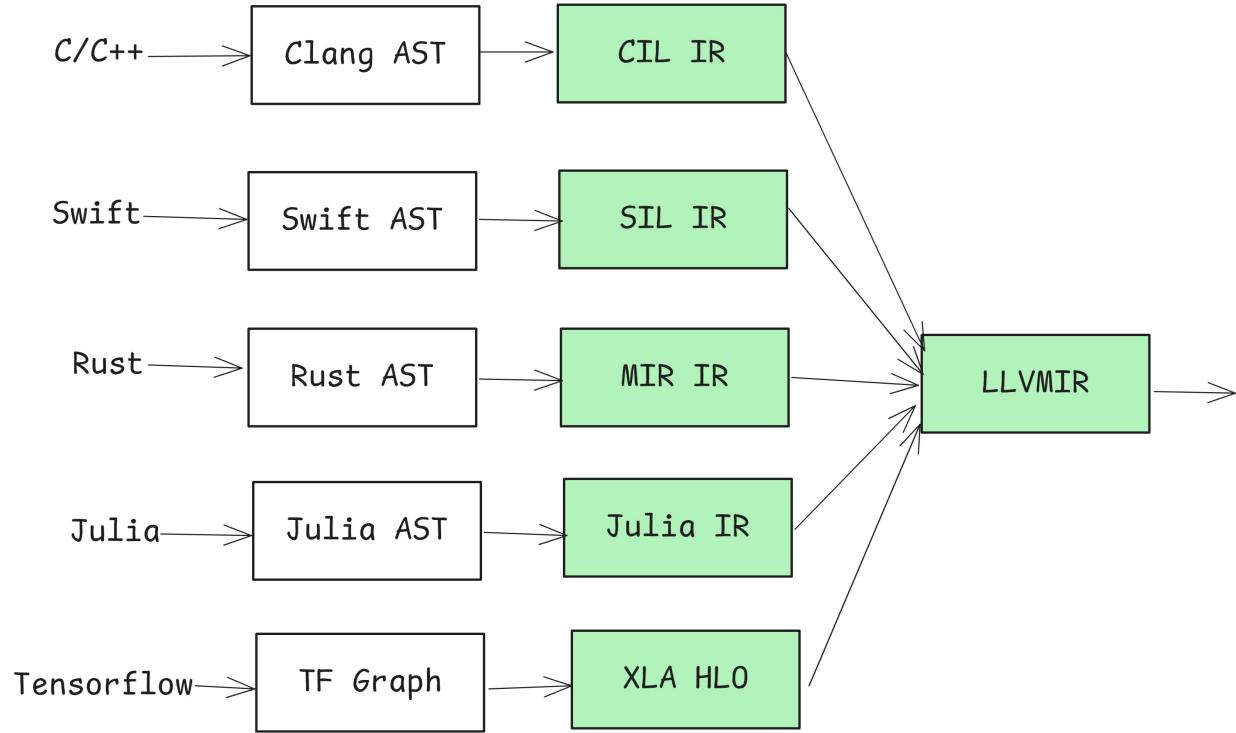


# MLIR 定位



但是在转换成 LLVM IR 之前呢？

# MLIR 定位

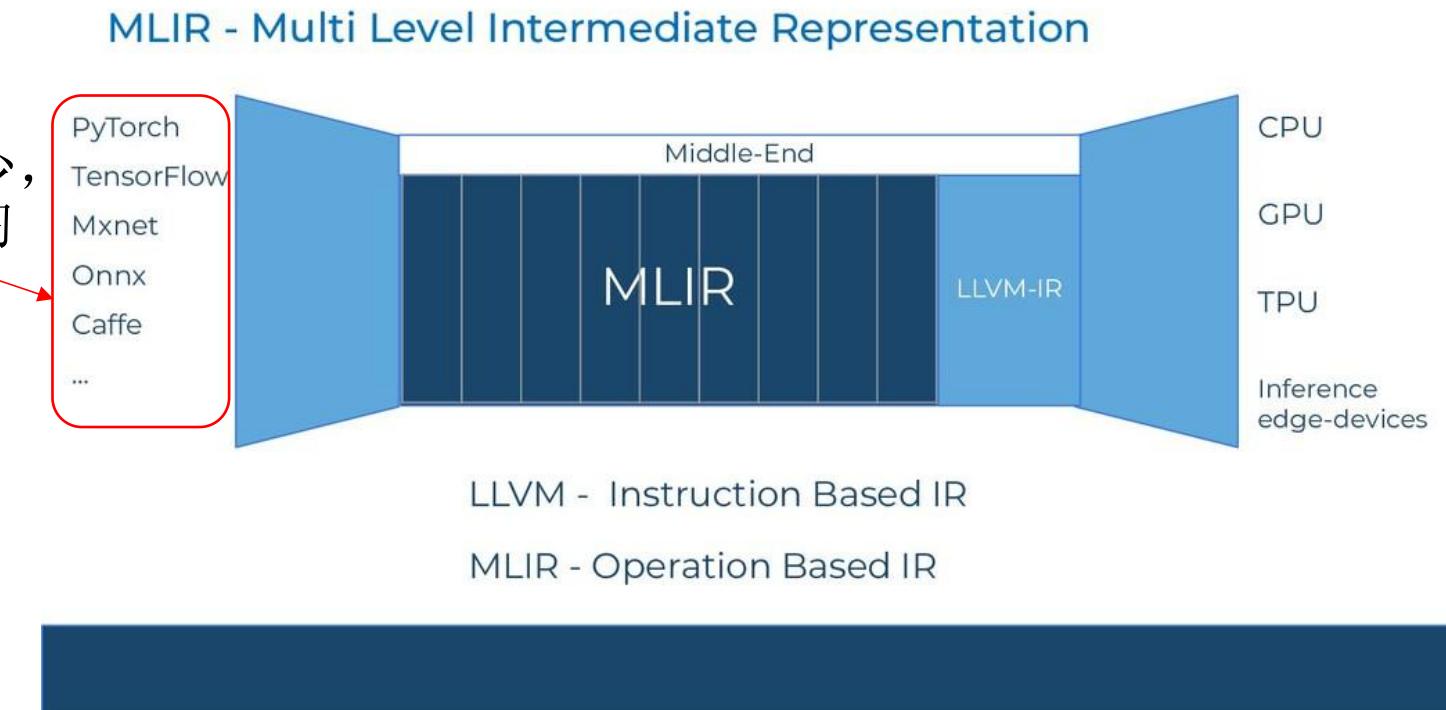


LLVM 并没有涵盖所有中端优化  
一些优化(特别是语言和领域相关)需要在转换成 LLVM IR 之前完成

# MLIR 定位

MLIR 的愿景是把 SSA IR 都纳入到 MLIR 这个统一的 IR 和基础设施中，以此解决编译器软件碎片化问题

但是目前用于通用编程语言较少，  
目前的主要应用场景是深度学习  
领域编译器



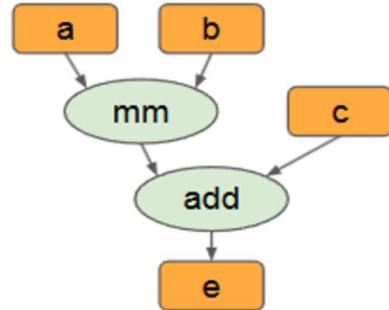
# Triton概述

[Triton](#) is a **language and compiler** for parallel programming. It aims to provide a Python-based programming environment for productively writing custom **DNN compute kernels** capable of running at maximal throughput on modern GPU hardware.

- eDSL: 利用 Python 语法和 Parser
- Compiler: 利用 MLIR 框架实现 IR 和 Pass
- DNN Compute Kernel: 主要用于编写深度神经网络计算密集型算子

# Triton定位

```
a = torch.randn(64, 32)
b = torch.randn(32, 64)
c = torch.randn(64, 64)
d = torch.mm(a, b)
e = c + d
```



```
__global__
void mm(float *a, float *b,
float *c) {
    float *a_tile;
    float *b_tile;
    ...
}
```



Model

Graph

Kernel

Device

Pytorch  
Tensorflow  
JAX

Pytorch FX  
XLA HLO  
TVM Relax

MUSA/CUDA  
Cutlass/Mutlass  
**Triton**

Moore Threads  
Nvidia  
AMD

# Triton定位



# Triton 编程模型

BLOCK\_SIZE 是传递给编译器的参数，表示 Tile 的大小

获取当前的 block id

获取当前 block 负责处理数据的起始地址，例如 64

计算当前 Block 处理 Tile 的偏移，例如 [64, 65, 66, ..., 127]

加载该 Block 负责计算的 Tile

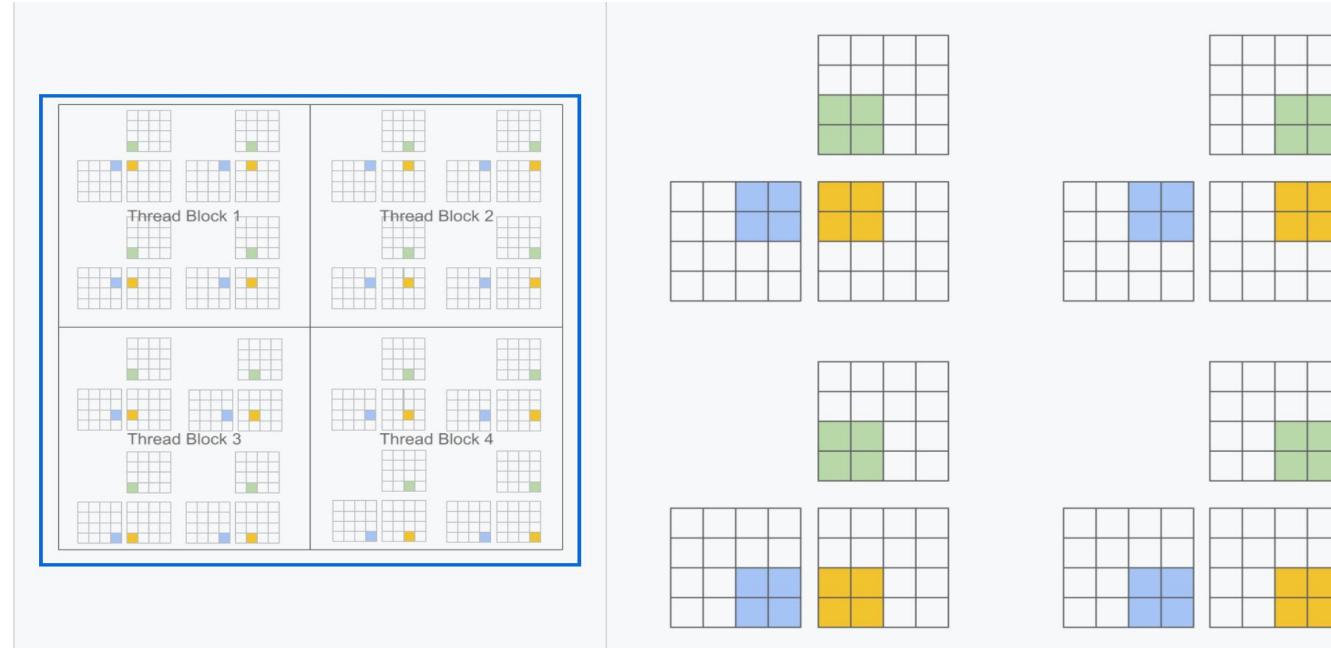
以 Tensor 为单位进行计算

```
1 @triton.jit
2 def add_kernel(x_ptr,           Input
3                 y_ptr,           Output
4                 output_ptr,      Output
5                 n_elements,     Input/Output tensor size
6                 BLOCK_SIZE: tl.constexpr,
7                 ):
8
9     pid = tl.program_id(axis=0)
10    block_start = pid * BLOCK_SIZE
11    offsets = block_start + tl.arange(0, BLOCK_SIZE)
12    mask = offsets < n_elements
13    x = tl.load(x_ptr + offsets, mask=mask)
14    y = tl.load(y_ptr + offsets, mask=mask)
15    output = x + y
16    tl.store(output_ptr + offsets, output, mask=mask)
```

# Triton 编程模型

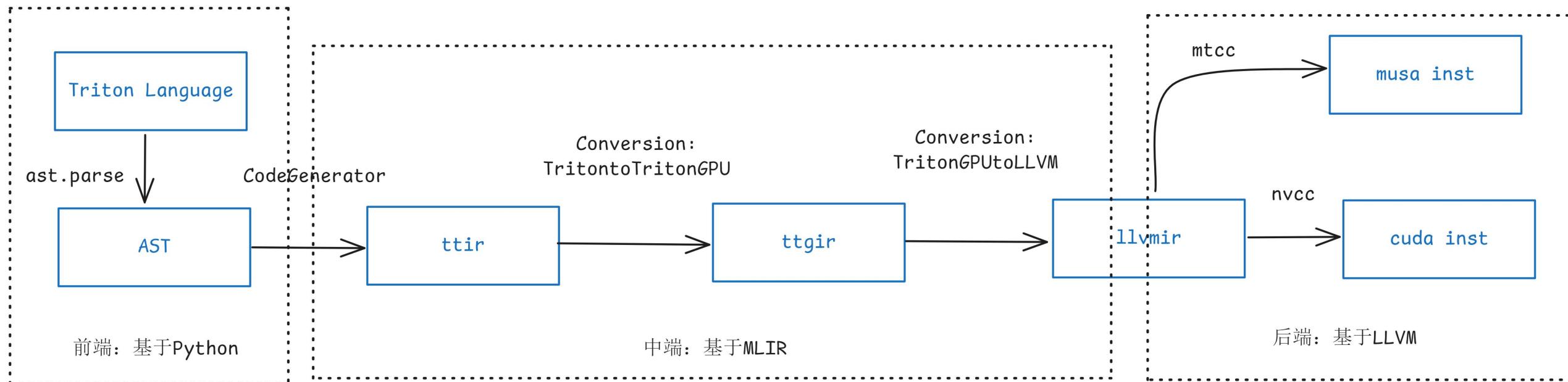
- Triton 在 **Block** 层级编写算子算法，更细粒度的线程级别的工作由 Triton 编译器负责，包括：
  - 线程映射
  - 共享内存
  - Barrier
  - Tensor Core
  - Coalescing 访存合并
- triton.lang 提供扩展 Python 语法，提供各种 Tensor 语义的操作
  - `arrange/cat/cast/full/zeros`
  - `broadcast/join/permute/reshape...`
  - `load/store...`
  - `dot/reduce...`

	CUDA	Triton	Torch Op
Algorithm	User	User	Compiler
Shared memory	User	Compiler	Compiler
Barriers	User	Compiler	Compiler
Distribution to blocks	User	User	Compiler
Grid size	User	User	Compiler
Distribution to Warps/threads	User	Compiler	Compiler
Tensor Core usage	User	Compiler	Compiler
Coalescing	User	Compiler	Compiler
Intermediate data layout	User	Compiler	Compiler
Workgroup size	User	User	Compiler



# Triton编译流程

- 前端：利用 `ast.parse` 把用户使用 triton 编写的 kernel 转换成 AST，使用 `ast.NodeVisitor` 遍历 AST 生成 MLIR
- 中端：利用 MLIR 的基础设施定义多层级 IR 和优化 Pass，对 IR 进行各种优化和转换
- 后端：利用 LLVM 生成各种目标硬件的机器指令



# Triton 编译流程示例

## A matmul kernel

```
● ○ ●

1 @triton.jit
2 def matmul_kernel_tma(a_desc_ptr, b_desc_ptr, c_desc_ptr, M, N, K,
3                         BLOCK_SIZE_M: tl.constexpr,
4                         BLOCK_SIZE_N: tl.constexpr,
5                         BLOCK_SIZE_K: tl.constexpr):
6     pid = tl.program_id(axis=0)
7     num_pid_m = tl.cdiv(M, BLOCK_SIZE_M)
8     pid_m = pid % num_pid_m
9     pid_n = pid // num_pid_m
10    offs_am = pid_m * BLOCK_SIZE_M
11    offs_bn = pid_n * BLOCK_SIZE_N
12    offs_k = 0
13    accumulator = tl.zeros((BLOCK_SIZE_M, BLOCK_SIZE_N), dtype=tl.float32)
14    for k in range(0, tl.cdiv(K, BLOCK_SIZE_K)):
15        a = tl._experimental_descriptor_load(a_desc_ptr, [offs_am, offs_k],
16                                            [BLOCK_SIZE_M, BLOCK_SIZE_K], tl.float16)
17        b = tl._experimental_descriptor_load(b_desc_ptr, [offs_k, offs_bn],
18                                            [BLOCK_SIZE_K, BLOCK_SIZE_N], tl.float16)
19        accumulator = tl.dot(a, b, acc=accumulator)
20        offs_k += BLOCK_SIZE_K
21    accumulator = accumulator.to(tl.float16)
22    tl._experimental_descriptor_store(c_desc_ptr, accumulator, [offs_am, offs_bn])
```

# 编译流程第一步：Triton IR

```
%0 = tt.get_program_id x : i32
%1 = arith.addi %arg3, %c31_i32 : i32
%2 = arith.divsi %1, %c32_i32 : i32
%3 = arith.remsi %0, %2 : i32
%4 = arith.divsi %0, %2 : i32
%5 = arith.muli %3, %c32_i32 : i32
%6 = arith.muli %4, %c32_i32 : i32
%7 = arith.addi %arg5, %c31_i32 : i32
%8 = arith.divsi %7, %c32_i32 : i32
%9:2 = scf.for %arg6 = %c0_i32 to %8 step %c1_i32 iter_args(%arg7 = %cst, %arg8 = %c0_i32) -> (tensor<32x32xf32>, i32) :
i32 {
    %11 = tt.experimental_descriptor_load %arg0[%5, %arg8] : !tt.ptr<i8> -> tensor<32x32xf16>
    %12 = tt.experimental_descriptor_load %arg1[%arg8, %6] : !tt.ptr<i8> -> tensor<32x32xf16>
    %13 = tt.dot %11, %12, %arg7, inputPrecision = tf32 : tensor<32x32xf16> * tensor<32x32xf16> -> tensor<32x32xf32>
    %14 = arith.addi %arg8, %c32_i32 : i32
    scf.yield %13, %14 : tensor<32x32xf32>, i32
}
%10 = arith.truncf %9#0 : tensor<32x32xf32> to tensor<32x32xf16>
```

# TTGIR: 引入layout

```
#mma = #triton_gpu.mthreads_sqmma<{versionMajor = 3, versionMinor = 1, warpsPerCTA = [4, 1], instrShape = [8, 32, 32]}>
#shared = #triton_gpu.shared<{vec = 1, perPhase = 1, maxPhase = 1, order = [1, 0], hasLeadingOffset = false}>

%12 = triton_gpu.local_alloc : () -> !tt.memdesc<32x32xf16, #shared, mutable>
triton_mthreads_gpu.init_arrival %c2_i32, %c4_i32, %c0_i32, %true : i32
triton_mthreads_gpu.async_tme_copy_global_to_local %arg0[%5, %arg8] %c2_i32, %12, %true {sqmma.opIdx = 0 : i32} : <i8>, i32 -> <32x32xf16, #shared, mutable>
%13 = triton_mthreads_gpu.barrier_arrive %c2_i32 : i32 -> i32
triton_mthreads_gpu.wait_barrier %c2_i32, %13 : i32
%14 = triton_gpu.local_alloc : () -> !tt.memdesc<32x32xf16, #shared, mutable>
triton_mthreads_gpu.init_arrival %c1_i32, %c4_i32, %c0_i32, %true : i32
triton_mthreads_gpu.async_tme_copy_global_to_local %arg1[%arg8, %6] %c1_i32, %14, %true {sqmma.opIdx = 1 : i32} : <i8>, i32 -> <32x32xf16, #shared, mutable>
%15 = triton_mthreads_gpu.barrier_arrive %c1_i32 : i32 -> i32
triton_mthreads_gpu.wait_barrier %c1_i32, %15 : i32
%16 = tt.dot %12, %14, %arg7, inputPrecision = tf32 : !tt.memdesc<32x32xf16, #shared, mutable>
* !tt.memdesc<32x32xf16, #shared, mutable> -> tensor<32x32xf32, #mma>
```

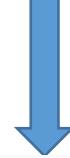
# Layout示例2：Vector ADD

```
#blocked = #triton_gpu.blocked<{sizePerThread = [4], threadsPerWarp = [32], warpsPerCTA = [4], order = [0], CTAsPerCGA = [1], CTASplitNum = [1], CTAOrder = [0]}>
module attributes {"triton_gpu.compute-capability" = 86 : i32, "triton_gpu.num-ctas" = 1 : i32, "triton_gpu.num-warps" = 4 : i32, "triton_gpu.threads-per-warp" = 32 : i32} {
    tt.func public @add_kernel_0d1d2d3de(%arg0: !tt.ptr<f32, 1> {tt.divisibility = 16 : i32}, %arg1: !tt.ptr<f32, 1> {tt.divisibility = 16 : i32}, %arg2: !tt.ptr<f32, 1> {tt.divisibility = 16 : i32}, %arg3: i32 {tt.divisibility = 16 : i32, tt.max_divisibility = 16 : i32}) attributes {noinline = false} {
        %c1024_i32 = arith.constant 1024 : i32
        %0 = tt.get_program_id x : i32
        %1 = arith.muli %0, %c1024_i32 : i32
        %2 = tt.make_range {end = 1024 : i32, start = 0 : i32} : tensor<1024xi32, #blocked>
        %3 = tt.splat %1 : (i32) -> tensor<1024xi32, #blocked>
        %4 = arith.addi %3, %2 : tensor<1024xi32, #blocked>
        %5 = tt.splat %arg3 : (i32) -> tensor<1024xi32, #blocked>
        %6 = arith.cmpi slt, %4, %5 : tensor<1024xi32, #blocked>
        %7 = tt.splat %arg0 : (!tt.ptr<f32, 1>) -> tensor<1024x!tt.ptr<f32, 1>, #blocked>
        %8 = tt.addptr %7, %4 : tensor<1024x!tt.ptr<f32, 1>, #blocked>, tensor<1024xi32, #blocked>
        %9 = tt.load %8, %6 {cache = 1 : i32, evict = 1 : i32, isVolatile = false} : tensor<1024xf32, #blocked>
        %10 = tt.splat %arg1 : (!tt.ptr<f32, 1>) -> tensor<1024x!tt.ptr<f32, 1>, #blocked>
        %11 = tt.addptr %10, %4 : tensor<1024x!tt.ptr<f32, 1>, #blocked>, tensor<1024xi32, #blocked>
        %12 = tt.load %11, %6 {cache = 1 : i32, evict = 1 : i32, isVolatile = false} : tensor<1024xf32, #blocked>
        %13 = arith.addf %9, %12 : tensor<1024xf32, #blocked>
        %14 = tt.splat %arg2 : (!tt.ptr<f32, 1>) -> tensor<1024x!tt.ptr<f32, 1>, #blocked>
        %15 = tt.addptr %14, %4 : tensor<1024x!tt.ptr<f32, 1>, #blocked>, tensor<1024xi32, #blocked>
        tt.store %15, %13, %6 {cache = 1 : i32, evict = 1 : i32} : tensor<1024xf32, #blocked>
        tt.return
    }
}
```

# SQMMA Layout Encoding



```
1 #triton_gpu.mthreads_sqmma<{versionMajor = 3, versionMinor = 1, warpsPerCTA = [4, 1], instrShape = [8, 32, 32]}>
```



T0:0	T1:0	T2:0	T3:0	T4:0	T5:0	T6:0	T7:0	T0:1	T1:1	T2:1	T3:1	T4:1	T5:1	T6:1	T7:1	T0:2	T1:2	T2:2	T3:2	T4:2	T5:2	T6:2	T7:2	T0:3	T1:3	T2:3	T3:3	T4:3	T5:3	T6:3	T7:3
T8:0	T9:0	T10:0	T11:0	T12:0	T13:0	T14:0	T15:0	T8:1	T9:1	T10:1	T11:1	T12:1	T13:1	T14:1	T15:1	T8:2	T9:2	T10:2	T11:2	T12:2	T13:2	T14:2	T15:2	T8:3	T9:3	T10:3	T11:3	T12:3	T13:3	T14:3	T15:3
T16:0	T17:0	T18:0	T19:0	T20:0	T21:0	T22:0	T23:0	T16:1	T17:1	T18:1	T19:1	T20:1	T21:1	T22:1	T23:1	T16:2	T17:2	T18:2	T19:2	T20:2	T21:2	T22:2	T23:2	T16:3	T17:3	T18:3	T19:3	T20:3	T21:3	T22:3	T23:3
T24:0	T25:0	T26:0	T27:0	T28:0	T29:0	T30:0	T31:0	T24:1	T25:1	T26:1	T27:1	T28:1	T29:1	T30:1	T31:1	T24:2	T25:2	T26:2	T27:2	T28:2	T29:2	T30:2	T31:2	T24:3	T25:3	T26:3	T27:3	T28:3	T29:3	T30:3	T31:3
T32:0	T33:0	T34:0	T35:0	T36:0	T37:0	T38:0	T39:0	T32:1	T33:1	T34:1	T35:1	T36:1	T37:1	T38:1	T39:1	T32:2	T33:2	T34:2	T35:2	T36:2	T37:2	T38:2	T39:2	T32:3	T33:3	T34:3	T35:3	T36:3	T37:3	T38:3	T39:3
T40:0	T41:0	T42:0	T43:0	T44:0	T45:0	T46:0	T47:0	T40:1	T41:1	T42:1	T43:1	T44:1	T45:1	T46:1	T47:1	T40:2	T41:2	T42:2	T43:2	T44:2	T45:2	T46:2	T47:2	T40:3	T41:3	T42:3	T43:3	T44:3	T45:3	T46:3	T47:3
T48:0	T49:0	T50:0	T51:0	T52:0	T53:0	T54:0	T55:0	T48:1	T49:1	T50:1	T51:1	T52:1	T53:1	T54:1	T55:1	T48:2	T49:2	T50:2	T51:2	T52:2	T53:2	T54:2	T55:2	T48:3	T49:3	T50:3	T51:3	T52:3	T53:3	T54:3	T55:3
T56:0	T57:0	T58:0	T59:0	T60:0	T61:0	T62:0	T63:0	T56:1	T57:1	T58:1	T59:1	T60:1	T61:1	T62:1	T63:1	T56:2	T57:2	T58:2	T59:2	T60:2	T61:2	T62:2	T63:2	T56:3	T57:3	T58:3	T59:3	T60:3	T61:3	T62:3	T63:3
T64:0	T65:0	T66:0	T67:0	T68:0	T69:0	T70:0	T71:0	T64:1	T65:1	T66:1	T67:1	T68:1	T69:1	T70:1	T71:1	T64:2	T65:2	T66:2	T67:2	T68:2	T69:2	T70:2	T71:2	T64:3	T65:3	T66:3	T67:3	T68:3	T69:3	T70:3	T71:3
T72:0	T73:0	T74:0	T75:0	T76:0	T77:0	T78:0	T79:0	T72:1	T73:1	T74:1	T75:1	T76:1	T77:1	T78:1	T79:1	T72:2	T73:2	T74:2	T75:2	T76:2	T77:2	T78:2	T79:2	T72:3	T73:3	T74:3	T75:3	T76:3	T77:3	T78:3	T79:3
T80:0	T81:0	T82:0	T83:0	T84:0	T85:0	T86:0	T87:0	T80:1	T81:1	T82:1	T83:1	T84:1	T85:1	T86:1	T87:1	T80:2	T81:2	T82:2	T83:2	T84:2	T85:2	T86:2	T87:2	T80:3	T81:3	T82:3	T83:3	T84:3	T85:3	T86:3	T87:3
T88:0	T89:0	T90:0	T91:0	T92:0	T93:0	T94:0	T95:0	T88:1	T89:1	T90:1	T91:1	T92:1	T93:1	T94:1	T95:1	T88:2	T89:2	T90:2	T91:2	T92:2	T93:2	T94:2	T95:2	T88:3	T89:3	T90:3	T91:3	T92:3	T93:3	T94:3	T95:3
T96:0	T97:0	T98:0	T99:0	T100:0	T101:0	T102:0	T103:0	T96:1	T97:1	T98:1	T99:1	T100:1	T101:1	T102:1	T103:1	T96:2	T97:2	T98:2	T99:2	T100:2	T101:2	T102:2	T103:2	T96:3	T97:3	T98:3	T99:3	T100:3	T101:3	T102:3	T103:3
T104:0	T105:0	T106:0	T107:0	T108:0	T109:0	T110:0	T111:0	T104:1	T105:1	T106:1	T107:1	T108:1	T109:1	T110:1	T111:1	T104:2	T105:2	T106:2	T107:2	T108:2	T109:2	T110:2	T111:2	T104:3	T105:3	T106:3	T107:3	T108:3	T109:3	T110:3	T111:3
T112:0	T113:0	T114:0	T115:0	T116:0	T117:0	T118:0	T119:0	T112:1	T113:1	T114:1	T115:1	T116:1	T117:1	T118:1	T119:1	T112:2	T113:2	T114:2	T115:2	T116:2	T117:2	T118:2	T119:2	T112:3	T113:3	T114:3	T115:3	T116:3	T117:3	T118:3	T119:3
T120:0	T121:0	T122:0	T123:0	T124:0	T125:0	T126:0	T127:0	T120:1	T121:1	T122:1	T123:1	T124:1	T125:1	T126:1	T127:1	T120:2	T121:2	T122:2	T123:2	T124:2	T125:2	T126:2	T127:2	T120:3	T121:3	T122:3	T123:3	T124:3	T125:3	T126:3	T127:3
T0:4	T1:4	T2:4	T3:4	T4:4	T5:4	T6:4	T7:4	T0:5	T1:5	T2:5	T3:5	T4:5	T5:5	T6:5	T7:5	T0:6	T1:6	T2:6	T3:6	T4:6	T5:6	T6:6	T7:6	T0:7	T1:7	T2:7	T3:7	T4:7	T5:7	T6:7	T7:7
T8:4	T9:4	T10:4	T11:4	T12:4	T13:4	T14:4	T15:4	T8:5	T9:5	T10:5	T11:5	T12:5	T13:5	T14:5	T15:5	T8:6	T9:6	T10:6	T11:6	T12:6	T13:6	T14:6	T15:6	T8:7	T9:7	T10:7	T11:7	T12:7	T13:7	T14:7	T15:7
T16:4	T17:4	T18:4	T19:4	T20:4	T21:4	T22:4	T23:4	T16:5	T17:5	T18:5	T19:5	T20:5	T21:5	T22:5	T23:5	T16:6	T17:6	T18:6	T19:6	T20:6	T21:6	T22:6	T23:6	T16:7	T17:7	T18:7	T19:7	T20:7	T21:7	T22:7	T23:7
T24:4	T25:4	T26:4	T27:4	T28:4	T29:4	T30:4	T31:4	T24:5	T25:5	T26:5	T27:5	T28:5	T29:5	T30:5	T31:5	T24:6	T25:6	T26:6	T27:6	T28:6	T29:6	T30:6	T31:6	T24:7	T25:7	T26:7	T27:7	T28:7	T29:7	T30:7	T31:7
T32:4	T33:4	T34:4	T35:4	T36:4	T37:4	T38:4	T39:4	T32:5	T33:5	T34:5	T35:5	T36:5	T37:5	T38:5	T39:5	T32:6	T33:6	T34:6	T35:6	T36:6	T37:6	T38:6	T39:6	T32:7	T33:7	T34:7	T35:7	T36:7	T37:7	T38:7	T39:7
T40:4	T41:4	T42:4	T43:4	T44:4	T45:4	T46:4	T47:4	T40:5	T41:5	T42:5	T43:5	T44:5	T45:5	T46:5	T47:5	T40:6	T41:6	T42:6	T43:6	T44:6	T45:6	T46:6	T47:6	T40:7	T41:7	T42:7	T43:7	T44:7	T45:7	T46:7	T47:7
T48:4	T49:4	T50:4	T51:4	T52:4	T53:4	T54:4	T55:4	T48:5	T49:5	T50:5	T51:5	T52:5	T53:5	T54:5	T55:5	T48:6	T49:6	T50:6	T51:6	T52:6	T53:6	T54:6	T55:6	T48:7	T49:7	T50:7	T51:7	T52:7	T53:7	T54:7	T55:7
T56:4	T57:4	T58:4	T59:4	T60:4	T61:4	T62:4	T63:4	T56:5	T57:5	T58:5	T59:5	T60:5	T61:5	T62:5	T63:5	T56:6	T57:6	T58:6	T59:6	T60:6	T61:6	T62:6	T63:6	T56:7	T57:7	T58:7	T59:7	T60:7	T61:7	T62:7	T63:7
T64:4	T65:4	T66:4	T67:4	T68:4	T69:4	T70:4	T71:4	T64:5	T65:5	T66:5	T67:5	T68:5	T69:5	T70:5	T71:5	T64:6	T65:6	T66:6	T67:6	T68:6	T69:6	T70:6	T71:6	T64:7	T65:7	T66:7	T67:7	T68:7	T69:7	T70:7	T71:7
T72:4	T73:4	T74:4	T75:4	T76:4	T77:4	T78:4	T79:4	T72:5	T73:5	T74:5	T75:5	T76:5	T77:5	T78:5	T79:5	T72:6	T73:6	T74:6	T75:6	T76:6	T77:6	T78:6	T79:6	T72:7	T73:7	T74:7	T75:7	T76:7	T77:7	T78:7	T79:7
T80:4	T81:4	T82:4	T83:4	T84:4	T85:4	T86:4	T87:4	T80:5	T81:5	T82:5	T83:5	T84:5	T85:5	T86:5	T87:5	T80:6	T81:6	T82:6	T83:6	T84:6	T85:6	T86:6	T87:6	T80:7	T81:7	T82:7	T83:7	T84:7	T85:7	T86:7	T87:7
T88:4	T89:4	T90:4	T91:4	T92:4	T93:4	T94:4	T95:4	T88:5	T89:5	T90:5	T91:5	T92:5	T93:5	T94:5	T95:5	T88:6	T89:6	T90:6	T91:6	T92:6	T93:6	T94:6	T95:6	T88:7	T89:7	T90:7	T91:7	T92:7	T93:7	T94:7	T95:7
T96:4	T97:4	T98:4	T99:4	T100:4	T101:4	T102:4	T103:4	T96:5	T97:5	T98:5	T99:5	T100:5	T101:5	T102:5	T103:5	T96:6	T97:6	T98:6	T99:6	T100:6	T101:6	T102:6	T103:6	T96:7	T97:7	T98:7	T99:7	T100:7	T101:7	T102:7	T103:7
T104:4	T105:4	T106:4	T107:4	T108:4	T109:4	T110:4	T111:4	T104:5	T105:5	T106:5	T107:5	T108:5	T109:5	T110:5	T111:5	T104:6	T105:6	T106:6	T107:6	T108:6	T109:6	T110:6	T111:6	T104:7	T105:7	T106:7	T107:7	T108:7	T109:7	T110:7	T111:7
T112:4	T113:4	T114:4	T115:4	T116:4	T117:4	T118:4	T119:4	T112:5	T113:5	T114:5	T115:5	T116:5	T117:5	T118:5	T119:5	T112:6	T113:6	T114:6	T115:6	T116:6	T117:6	T118:6	T119:6	T112:7	T113:7	T114:7	T115:7	T116:7	T117:7	T118:7	T119:7
T120:4	T121:4	T122:4	T123:4	T124:4	T125:4	T126:4	T127:4	T120:5	T121:5	T122:5	T123:5	T124:5	T125:5	T126:5	T127:5	T120:6	T121:6	T122:6	T123:6	T124:6	T125:6	T126:6	T127:6	T120:7	T121:7	T122:7	T123:7	T124:7	T125:7	T126:7	T127:7

# 总结

- 1. GPU是多线程并发的硬件实现
- 2. 编程优化主要是面向GPU的存储体系结构： Register File, Shared Memory, L1/L2 Cache
- 3. 当前主流的AI编程语言以eDSL为主，主要目的在于简化编程模型，提高编程效率。
- 4. MLIR是支持AI编程语言的核心技术