中国科学院大学计算机学院专业选修课

GPU架构与编程

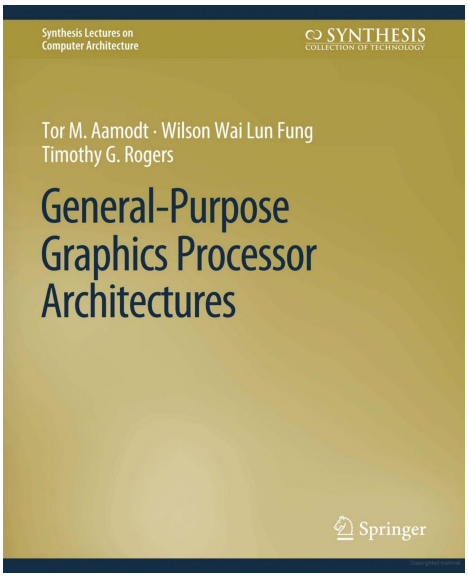
第二课：GPU架构简介

赵地
中科院计算所
2025年秋季学期

讲授内容

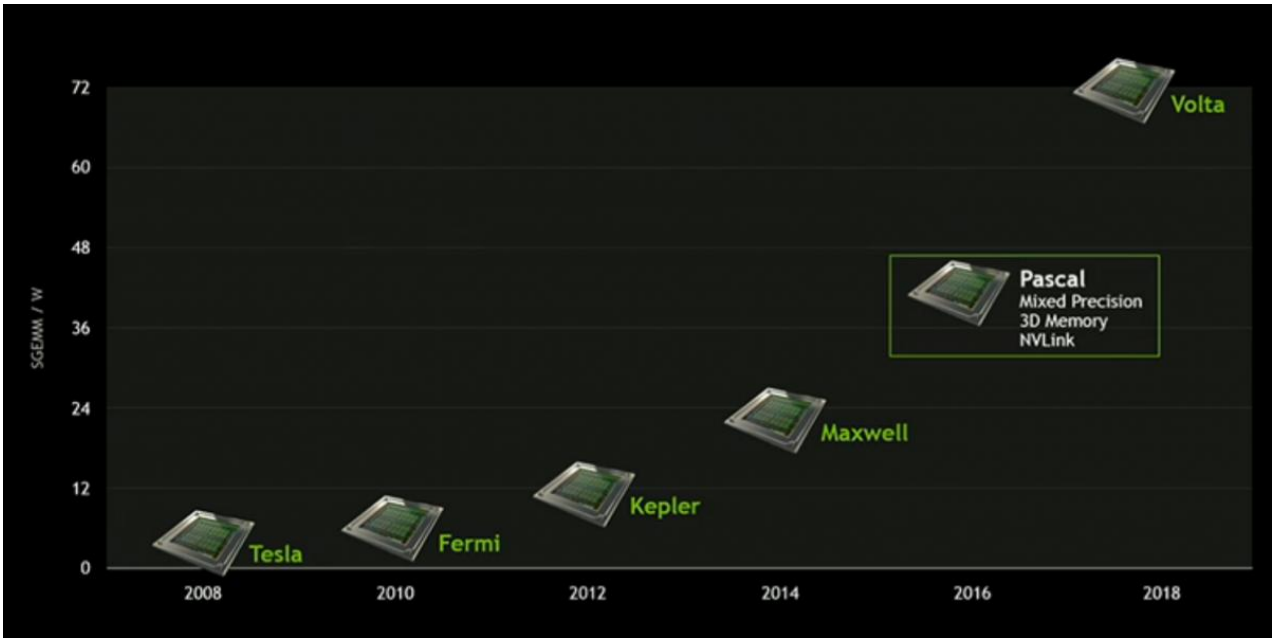
- GPU架构综述
- 编程模型
- SMIT核
- 存储系统
- 最新研究

参考书：GPU架构



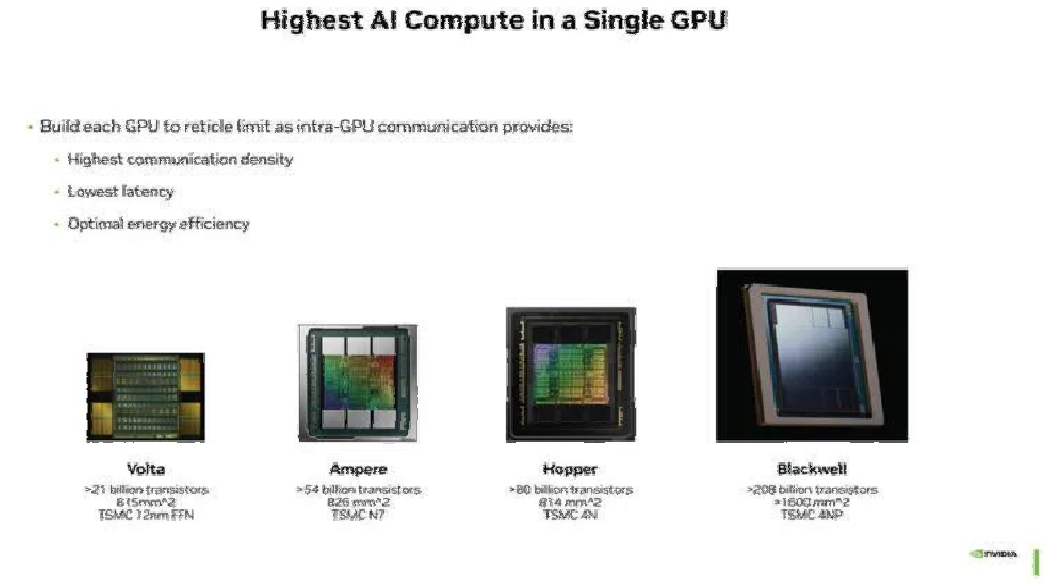
✓https://link.springer.com/book/10.1007/978-3-031-01759-9

GPU的发展史（英伟达GPU为例）



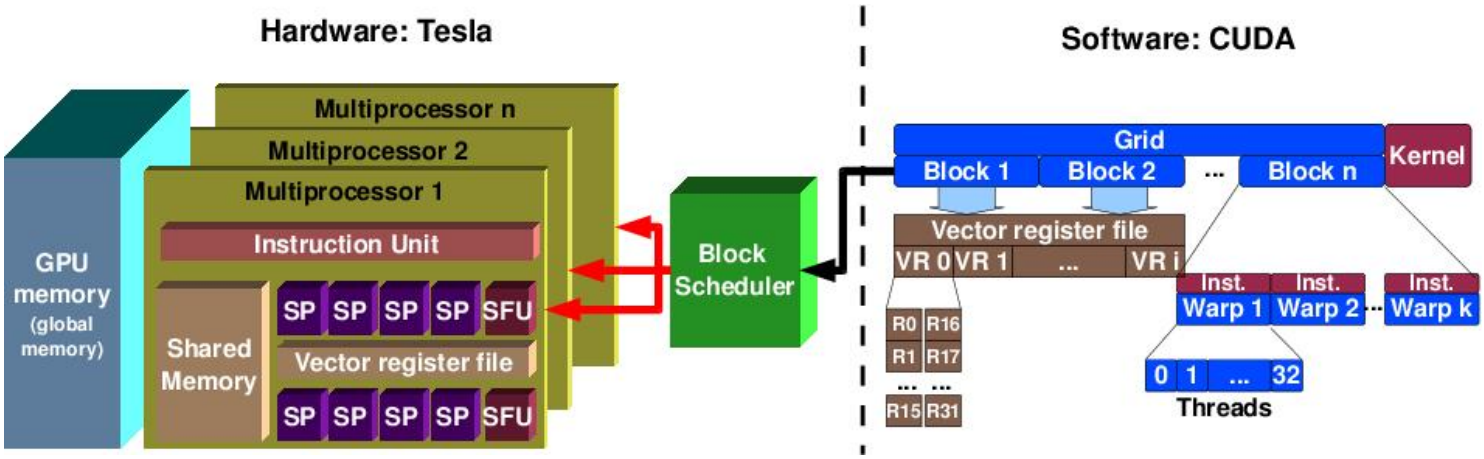
slides from Nvidia

GPU的发展史（英伟达GPU为例）



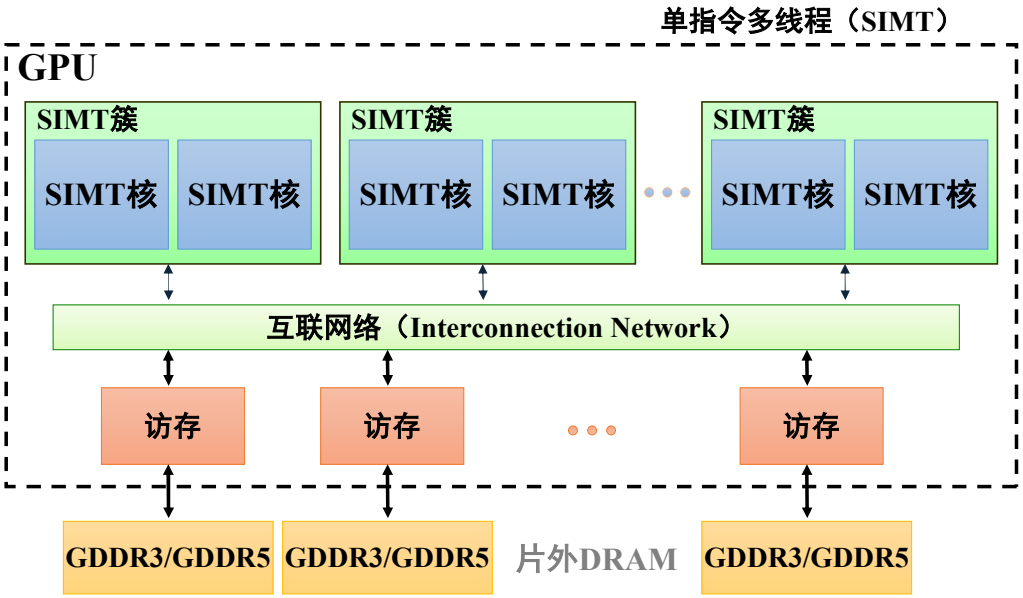
slides from Nvidia

总体构架：从CUDA到GPU



C. Collange, M. Daumas, D. Defour and D. Parello, "Barra: A Parallel Functional Simulator for GPGPU," 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Miami Beach, FL, USA, 2010, pp. 351-360, doi: 10.1109/MASCOTS.2010.43.

总体构架：GPU架构（GPGPU-Sim）



- ✓SIMT核
  - ✓SIMT流水线（SIMT Frontend）
  - ✓缓存
- ✓片上网络（Interconnection Network）
- ✓时钟
- ✓缓存与访存

http://www.gpgpu-sim.org/

Performance between Multicore and Multithreaded Architectures

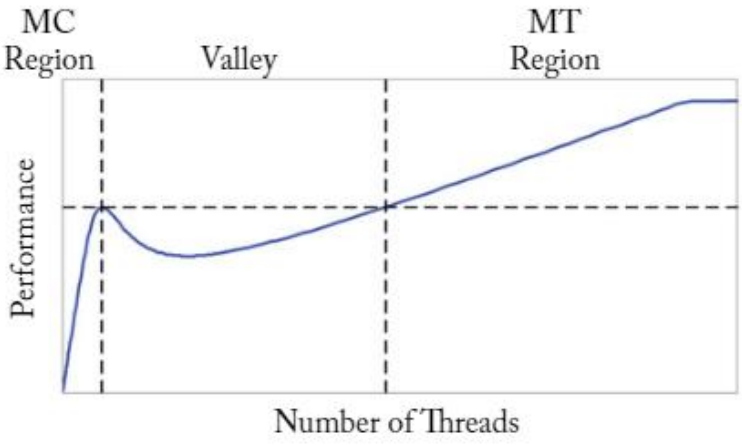


Figure 1.3: An analytical model-based analysis of the performance tradeoff between multicore (MC) CPU architectures and multithreaded (MT) architectures such as GPUs shows a "performance valley" may occur if the number of threads is insufficient to cover off-chip memory access latency (based on Figure 1 from Guz et al. [2009]).

Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. 2018. General-purpose Graphics Processor Architectures. Morgan & Claypool Publishers.

Energy Consumption

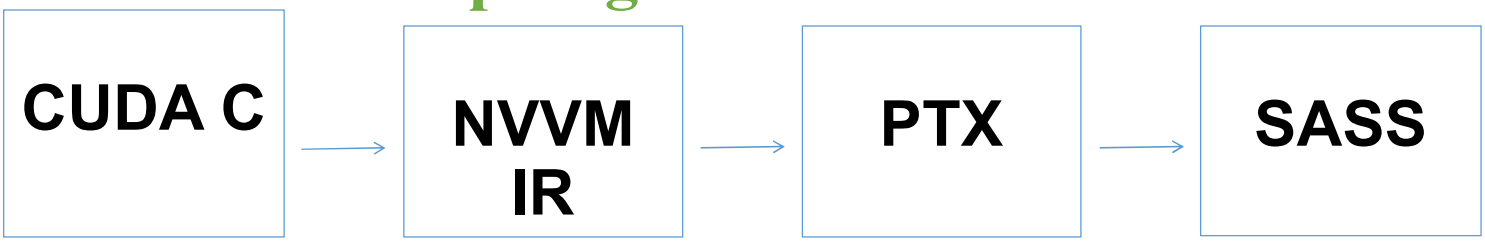
Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit 32KB SRAM	5	50
32 bit DRAM	640	6400

Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. 2018. General-purpose Graphics Processor Architectures. Morgan & Claypool Publishers.

讲授内容

- GPU架构综述
- 编程模型
- SMIT核
- 存储系统
- 最新研究

CUDA Compiling



- **CUDA C**：这是NVIDIA推出的一种并行计算平台和编程模型，允许开发者使用类C语言来编写程序，以利用GPU的大规模并行处理能力进行通用计算。
- **NVVM IR**：这是NVIDIA基于LLVM IR设计的一种中间表示，专门用于表示GPU计算内核，便于NVIDIA的GPU编译器进行优化和代码生成。
- **PTX**：这是一种并行线程执行的虚拟指令集（也称为NVPTX），它作为编译过程中的一个中间步骤，最终由驱动程序转换为特定GPU架构的机器代码（SASS）。
- **SASS**：这是NVIDIA GPU的实际硬件指令集，是PTX代码经过汇编后生成的二进制机器码，直接在GPU流处理器上执行。

NVCC: Nvidia CUDA Compiler

- 核心角色：NVCC是NVIDIA CUDA的编译器驱动程序，其主要作用是处理混合了主机（CPU）代码和设备（GPU）代码的CUDA源文件（如.cu文件）。
- 工作原理：通过分离代码，将设备代码编译成PTX中间代码或二进制机器码（如CUBIN），同时调用系统的主机编译器（如gcc或cl.exe）来编译主机代码。
- 输出选项：NVCC可以根据编译选项生成不同类型的输出，包括可直接运行的可执行文件、PTX汇编代码、对象文件或胖二进制文件（Fat Binary），后者可包含多种架构的代码以提升兼容性。
- 运行依赖：编译生成的CUDA可执行文件需要依赖CUDA核心库（如cuda）和CUDA运行时库（cudart）才能正常运行。

https://en.wikipedia.org/wiki/Nvidia_CUDA_Compiler

NVCC: Nvidia CUDA Compiler

- 关键技术特性：采用分层编译模式，将设备代码编译成与GPU架构无关的PTX中间代码，然后再转换为特定架构的二进制代码，以平衡兼容性与性能。
- 基于LLVM构建：从CUDA 4.1版本开始，NVCC的底层优化器和PTX代码生成器基于LLVM框架构建，以利用其模块化架构实现更好的优化。
- 编译流程特点：NVCC本身不直接完成所有编译工作，其主要角色是分离代码，并调用主机编译器（如gcc）处理主机代码，调用专用工具链（如cicc, ptxas）处理设备代码，最后协调链接。
- 兼容性与灵活性：通过生成PTX代码和打包多架构二进制文件的“fatbinary”机制，NVCC确保了编译出的程序能够在不同代际的NVIDIA GPU上运行，实现了向前兼容。

https://en.wikipedia.org/wiki/Nvidia_CUDA_Compiler

PTX：并行线程执行

✓并行线程执行（Parallel Thread eXecution，PTX）是英伟达公司使用的指令集；

✓PTX是伪汇编指令集：不在硬件上直接执行；

✓ptxas是NVIDIA发布的汇编器，将PTX汇编成在硬件运行的指令（SASS）；

✓每一代硬件都支持不同版本的SASS，PTX在编译时被编译成多个版本的SASS，对应不同版本的硬件。

PTX：并行线程执行

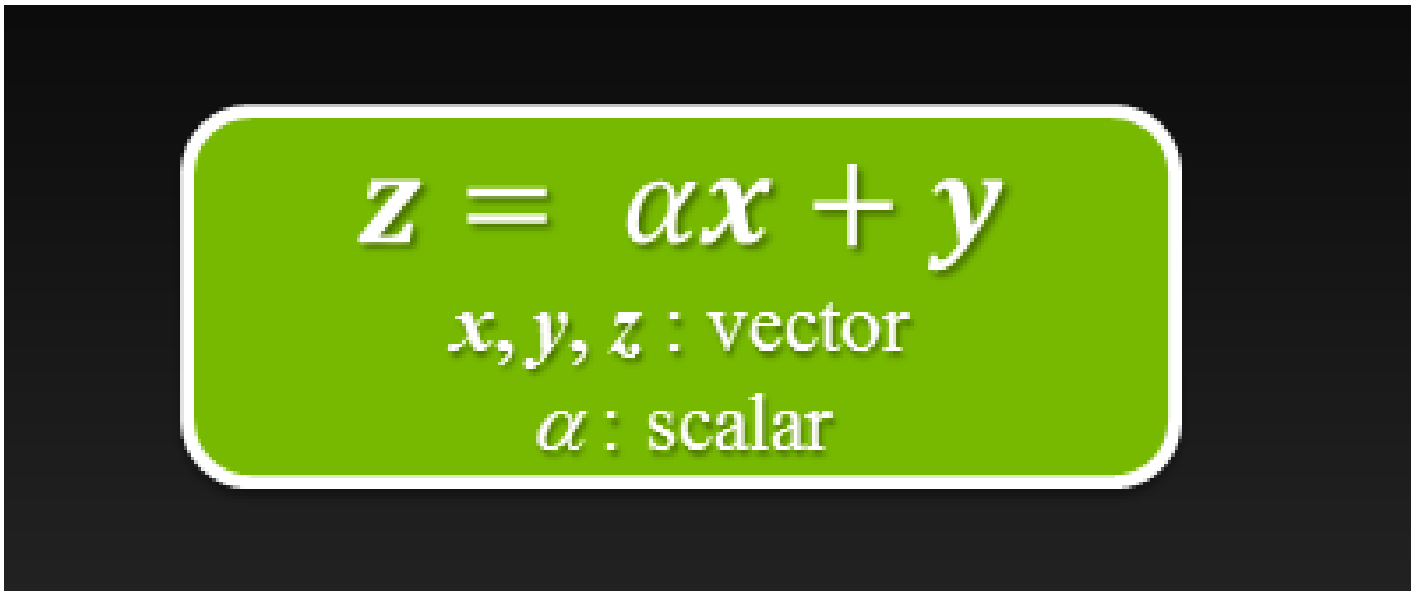
- ✓PTX代码仍然嵌入到二进制文件中，以支持未来的硬件；
- ✓在运行时，运行时系统根据可用硬件选择合适版本的SASS运行；
- ✓如果没有，运行时系统（the runtime system）会调用嵌入式PTX上的即时(just-in-time，JIT) 编译器，将PTX代码编译为与可用硬件相对应的SASS。

SASS

- ✓底层机器码：SASS是NVIDIA GPU直接执行的底层机器码指令集，与具体的流式多处理器（SM）硬件架构紧密对应。
- ✓官方有限的工具与文档：NVIDIA提供了闭源编译器nvcc和反汇编工具cuobjdump，可将二进制代码转换为SASS汇编，但并未提供官方的SASS汇编器将汇编代码翻译回二进制。同时，官方对SASS的文档支持也较为有限。
- ✓非官方汇编器尝试：由于缺乏官方汇编器，研究人员和开发者创建了如TuringAs（支持Volta、Turing、Ampere架构）等第三方开源SASS汇编器，以支持底层代码生成和研究，但它们属于非官方工具。

Ari B. Hayes, Fei Hua, Jin Huang, Yanhao Chen, Eddy Z. Zhang, Decoding CUDA Binary, CGO 2019, 229-241;

Example: SAXPY computation



https://developer.nvidia.com/blog/six-ways-saxpy/

CPU version of SAXPY

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
main() {
    float *x, *y;
    int n;
    // omitted: allocate CPU memory for x and y and
initialize contents
    saxpy_serial(n, 2.0, x, y); // Invoke serial SAXPY
kernel
    // omitted: use y on CPU, free memory pointed to by x
and y
}
```

https://developer.nvidia.com/blog/six-ways-saxpy/

CUBLAS version of SAXPY

```
int N = 1<<20;
cublasInit();
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);
cublasShutdown();
```

https://developer.nvidia.com/blog/six-ways-saxpy/

OpenACC version of SAXPY

```
void saxpy(int n, float a, float *
restrict x, float * restrict y)
{
#pragma acc kernels
  for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}
...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
```

https://developer.nvidia.com/blog/six-ways-saxpy/

Thrust version of SAXPY

```
using thrust::placeholders;

int N = 1<<20;
thrust::host_vector x(N), y(N);
...
thrust::device_vector d_x = x;
// alloc and copy host to device
thrust::device_vector d_y = y;
// Perform SAXPY on 1M elements
thrust::transform(d_x.begin(), d_x.end(),
d_y.begin(), d_y.begin(), 2.0f * _1 + _2);
y = d_y; // copy results to the host vector
```

https://developer.nvidia.com/blog/six-ways-saxpy/

CUDA version of SAXPY

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
      int i = blockIdx.x*blockDim.x + threadIdx.x;
      if(i<n)
           y[i] = a*x[i] + y[i];
}
int main() {
      float *h_x, *h_y;
      int n;
      // omitted: allocate CPU memory for h_x and h_y and initialize contents
      float *d_x, *d_y;
      int nblocks = (n + 255) / 256;
      cudaMalloc( &d_x, n * sizeof(float) );
      cudaMalloc( &d_y, n * sizeof(float) );
      cudaMemcpy( d_x, h_x, n * sizeof(float), cudaMemcpyHostToDevice );
      cudaMemcpy( d_y, h_y, n * sizeof(float), cudaMemcpyHostToDevice );
      saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
      cudaMemcpy( h_x, d_x, n * sizeof(float), cudaMemcpyDeviceToHost );
      // omitted: use h_y on CPU, free memory pointed to by h_x, h_y, d_x, and d_y
}
```

https://developer.nvidia.com/blog/six-ways-saxpy/

PTX version of SAXPY

```
.visible .entry _Z5saxpyifPfS_(
.param .u32 _Z5saxpyifPfS__param_0,
.param .f32 _Z5saxpyifPfS__param_1,
.param .u64 _Z5saxpyifPfS__param_2,
.param .u64 _Z5saxpyifPfS__param_3
)
{
.reg .pred %p<2>;
.reg .f32 %f<5>;
.reg .b32 %r<6>;
.reg .b64 %rd<8>;

ld.param.u32 %r2,
[_Z5saxpyifPfS__param_0];
ld.param.f32 %f1,
[_Z5saxpyifPfS__param_1];
ld.param.u64 %rd1,
[_Z5saxpyifPfS__param_2];
ld.param.u64 %rd2,
[_Z5saxpyifPfS__param_3];
mov.u32 %r3, %ctaid.x;
mov.u32 %r4, %ntid.x;
mov.u32 %r5, %tid.x;
mad.lo.s32 %r1, %r4, %r3, %r5;
setp.ge.s32 %p1, %r1, %r2;
@%p1 bra BB0_2;

cvta.to.global.u64 %rd3, %rd2;
cvta.to.global.u64 %rd4, %rd1;
mul.wide.s32 %rd5, %r1, 4;
add.s64 %rd6, %rd4, %rd5;
ld.global.f32 %f2, [%rd6];
add.s64 %rd7, %rd3, %rd5;
ld.global.f32 %f3, [%rd7];
fma.rn.f32 %f4, %f2, %f1, %f3;
st.global.f32 [%rd7], %f4;

BB0_2:
ret;
}
```

Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. 2018. General-purpose Graphics Processor Architectures. Morgan & Claypool Publishers.

SASS version of SAXPY (NVIDIA Fermi Architecture)

```
Address     Dissassembly                                    Encoded Instruction
========    ============================================    ======================
/*0000*/         MOV R1, c[0x1][0x100];                    /* 0x2800440400005de4 */
/*0008*/         S2R R0, SR_CTAID.X;                       /* 0x2c00000094001c04 */
/*0010*/         S2R R2, SR_TID.X;                         /* 0x2c00000084009c04 */
/*0018*/         IMAD R0, R0, c[0x0][0x8], R2;             /* 0x2004400020001ca3 */
/*0020*/         ISETP.GE.AND P0, PT, R0, c[0x0][0x20], PT; /* 0x1b0e40008001dc23 */
/*0028*/     @P0 BRA.U 0x78;                               /* 0x40000001200081e7 */
/*0030*/    @!P0 MOV32I R5, 0x4;                           /* 0x18000000100161e2 */
/*0038*/    @!P0 IMAD R2.CC, R0, R5, c[0x0][0x28];         /* 0x200b8000a000a0a3 */
/*0040*/    @!P0 IMAD.HI.X R3, R0, R5, c[0x0][0x2c];       /* 0x208a8000b000e0e3 */
/*0048*/    @!P0 IMAD R4.CC, R0, R5, c[0x0][0x30];         /* 0x200b8000c00120a3 */
/*0050*/    @!P0 LD.E R2, [R2];                            /* 0x840000000020a085 */
/*0058*/    @!P0 IMAD.HI.X R5, R0, R5, c[0x0][0x34];       /* 0x208a8000d00160e3 */
/*0060*/    @!P0 LD.E R0, [R4];                            /* 0x8400000000402085 */
/*0068*/    @!P0 FFMA R0, R2, c[0x0][0x24], R0;            /* 0x3000400090202000 */
/*0070*/    @!P0 ST.E [R4], R0;                            /* 0x9400000000402085 */
/*0078*/         EXIT;                                     /* 0x8000000000001de7 */
```

Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. 2018. General-purpose Graphics Processor Architectures. Morgan & Claypool Publishers.

SASS version of SAXPY (NVIDIA Pascal Architecture)

```
Address     Dissassembly                                  Encoded Instruction
=======     ============================================  ====================
                                                          /* 0x001c7c00e22007f6 */
/*0008*/       MOV R1, c[0x0][0x20];                      /* 0x4c98078000870001 */
/*0010*/       S2R R0, SR_CTAID.X;                        /* 0xf0c8000002570000 */
/*0018*/       S2R R2, SR_TID.X;                          /* 0xf0c8000002170002 */
                                                          /* 0x001fd840fec20ff1 */
/*0028*/       XMAD.MRG R3, R0.reuse, c[0x0] [0x8].H1, RZ; /* 0x4f107f8000270003 */
/*0030*/       XMAD R2, R0.reuse, c[0x0] [0x8], R2;       /* 0x4e0001000270002 */
/*0038*/       XMAD.PSL.CBCC R0, R0.H1, R3.H1, R2;        /* 0x5b30011800370000 */
                                                          /* 0x081fc400ffa007ed */
/*0048*/       ISETP.GE.AND P0, PT, R0, c[0x0][0x140], PT; /* 0x4b6d038005070007 */
/*0050*/  @P0 EXIT;                                       /* 0xe30000000000000f */
/*0058*/       SHL R2, R0.reuse, 0x2;                     /* 0x3848000000270002 */
                                                          /* 0x081fc440fec007f5 */
/*0068*/       SHR R0, R0, 0x1e;                          /* 0x3829000001e70000 */
/*0070*/       IADD R4.CC, R2.reuse, c[0x0][0x148];       /* 0x4c10800005270204 */
/*0078*/       IADD.X R5, R0.reuse, c[0x0][0x14c];        /* 0x4c10080005370005 */
                                                          /* 0x0001c800fe0007f6 */
/*0088*/       IADD R2.CC, R2, c[0x0][0x150];             /* 0x4c10800005470202 */
/*0090*/       IADD.X R3, R0, c[0x0][0x154];              /* 0x4c10080005570003 */
/*0098*/       LDG.E R0, [R4];        }                   /* 0xeed4200000070400 */
                                                          /* 0x0007c408fc400172 */
/*00a8*/       LDG.E R6, [R2];                            /* 0xeed4200000070206 */
/*00b0*/       FFMA R0, R0, c[0x0][0x144], R6;            /* 0x4980030005170000 */
/*00b8*/       STG.E [R2], R0;                            /* 0xeedc200000070200 */
                                                          /* 0x001f8000ffe007ff */
/*00c8*/       EXIT;                                      /* 0xe30000000007000f */
/*00d0*/       BRA 0xd0;                                  /* 0xe2400fffff87000f */
/*00d8*/       NOP;                                       /* 0x50b0000000070f00 */
                                                          /* 0x001f8000fc0007e0 */
/*00e8*/       NOP;                                       /* 0x50b0000000070f00 */
/*00f0*/       NOP;                                       /* 0x50b0000000070f00 */
/*00f8*/       NOP;                                       /* 0x50b0000000070f00 */
```

Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. 2018. General-purpose Graphics Processor Architectures. Morgan & Claypool Publishers.

OpenCL version of SAXPY

```
int i;
// Allocate space for vectors A, B and C
float alpha = 2.0;
float *A =
(float*)malloc(sizeof(float)*VECTOR_SIZE);
float *B =
(float*)malloc(sizeof(float)*VECTOR_SIZE);
float *C =
(float*)malloc(sizeof(float)*VECTOR_SIZE);
for(i = 0; i < VECTOR_SIZE; i++)
{
    A[i] = i;
    B[i] = VECTOR_SIZE - i;
    C[i] = 0;
}
```

```
// Get platform and device information
cl_platform_id * platforms = NULL;
cl_uint     num_platforms;
//Set up the Platform
cl_int clStatus = clGetPlatformIDs(0,
NULL, &num_platforms);
platforms = (cl_platform_id *)

malloc(sizeof(cl_platform_id)*num_platf
orms);
clStatus =
clGetPlatformIDs(num_platforms,
platforms, NULL);
```

https://developer.nvidia.com/blog/six-ways-saxpy/

OpenCL version of SAXPY

```
//Get the devices list and choose the
device you want to run on
cl_device_id     *device_list = NULL;
cl_uint           num_devices;

clStatus =
clGetDeviceIDs( platforms[0],
CL_DEVICE_TYPE_GPU, 0,NULL,
&num_devices);
device_list = (cl_device_id *)
malloc(sizeof(cl_device_id)*num_device
s);
clStatus =
clGetDeviceIDs
( platforms[0],CL_DEVICE_TYPE_GPU,
num_devices, device_list, NULL);
// Create one OpenCL context for each
device in the platform
cl_context context;
context = clCreateContext( NULL,
num_devices, device_list, NULL, NULL,
&clStatus);

// Create a command queue
cl_command_queue command_queue =
clCreateCommandQueue(context,
device_list[0], 0, &clStatus);

// Create memory buffers on the device for
each vector
cl_mem A_clmem = clCreateBuffer(context,
CL_MEM_READ_ONLY,VECTOR_SIZE *
sizeof(float), NULL, &clStatus);
cl_mem B_clmem = clCreateBuffer(context,
CL_MEM_READ_ONLY,VECTOR_SIZE *
sizeof(float), NULL, &clStatus);
cl_mem C_clmem = clCreateBuffer(context,
CL_MEM_WRITE_ONLY,VECTOR_SIZE *
sizeof(float), NULL, &clStatus);
```

https://developer.nvidia.com/blog/six-ways-saxpy/

OpenCL version of SAXPY

```
// Copy the Buffer A and B to the device
clStatus =
clEnqueueWriteBuffer(command_queue, A_clmem,
CL_TRUE, 0, VECTOR_SIZE * sizeof(float), A,
0, NULL, NULL);
clStatus =
clEnqueueWriteBuffer(command_queue, B_clmem,
CL_TRUE, 0, VECTOR_SIZE * sizeof(float), B,
0, NULL, NULL);
// Create a program from the kernel source
cl_program program =
clCreateProgramWithSource(context, 1,(const
char **)&saxpy_kernel, NULL, &clStatus);
// Build the program
clStatus = clBuildProgram(program, 1,
device_list, NULL, NULL, NULL);
// Create the OpenCL kernel
cl_kernel kernel = clCreateKernel(program,
"saxpy_kernel", &clStatus);
// Set the arguments of the kernel
clStatus = clSetKernelArg(kernel, 0,
sizeof(float), (void *)&alpha);
clStatus = clSetKernelArg(kernel, 1,
sizeof(cl_mem), (void *)&A_clmem);
clStatus = clSetKernelArg(kernel, 2,
sizeof(cl_mem), (void *)&B_clmem);
clStatus = clSetKernelArg(kernel, 3,
sizeof(cl_mem), (void *)&C_clmem);
// Execute the OpenCL kernel on the
list
size_t global_size = VECTOR_SIZE; //
Process the entire lists
size_t local_size = 64;           //
Process one item at a time
clStatus =
clEnqueueNDRangeKernel(command_queue,
kernel, 1, NULL, &global_size,
&local_size, 0, NULL, NULL);
```

https://developer.nvidia.com/blog/six-ways-saxpy/

OpenCL version of SAXPY

```
// Read the cl memory C_clmem on device
to the host variable C
clStatus =
clEnqueueReadBuffer(command_queue,
C_clmem, CL_TRUE, 0, VECTOR_SIZE *
sizeof(float), C, 0, NULL, NULL);

// Clean up and wait for all the comands
to complete.
clStatus = clFlush(command_queue);
clStatus = clFinish(command_queue);

// Display the result to the screen
for(i = 0; i < VECTOR_SIZE; i++)
    printf("%f * %f + %f = %f\n", alpha,
A[i], B[i], C[i]);
```

```
// Finally release all OpenCL allocated
objects and host buffers.
clStatus = clReleaseKernel(kernel);
clStatus = clReleaseProgram(program);
clStatus = clReleaseMemObject(A_clmem);
clStatus = clReleaseMemObject(B_clmem);
clStatus = clReleaseMemObject(C_clmem);
clStatus =
clReleaseCommandQueue(command_queue);
clStatus = clReleaseContext(context);
free(A);
free(B);
free(C);
free(platforms);
free(device_list);
return 0;
```

https://developer.nvidia.com/blog/six-ways-saxpy/

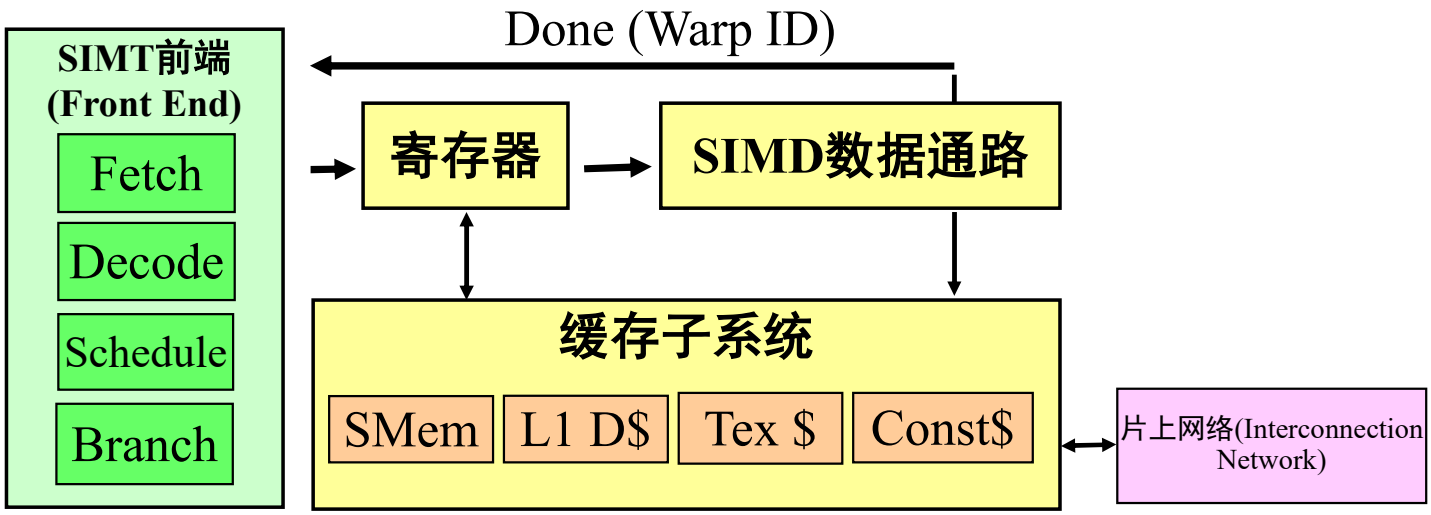
讲授内容

- GPU架构综述
- 编程模型
- SMIT核
- 存储系统
- 最新研究

SMIT核

1. 总体架构
2. 前端
3. 操作数收集器
4. ALU、FPU、SPU、Tensor Core
5. LSU
6. CTA调度器

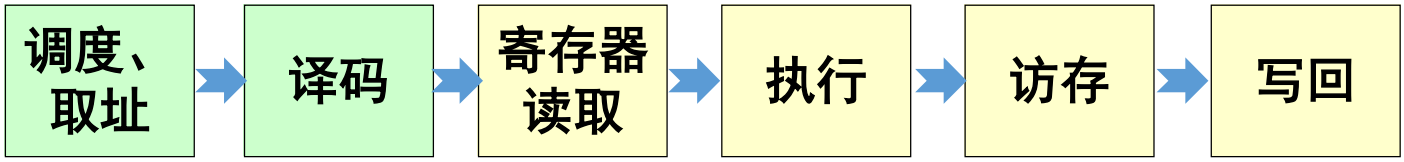
SIMT核



通过细粒度的多线程（fine-grained multithreading）设计，通过交错执行warp，SIMT掩藏warp执行的延迟。

http://www.gpgpu-sim.org/

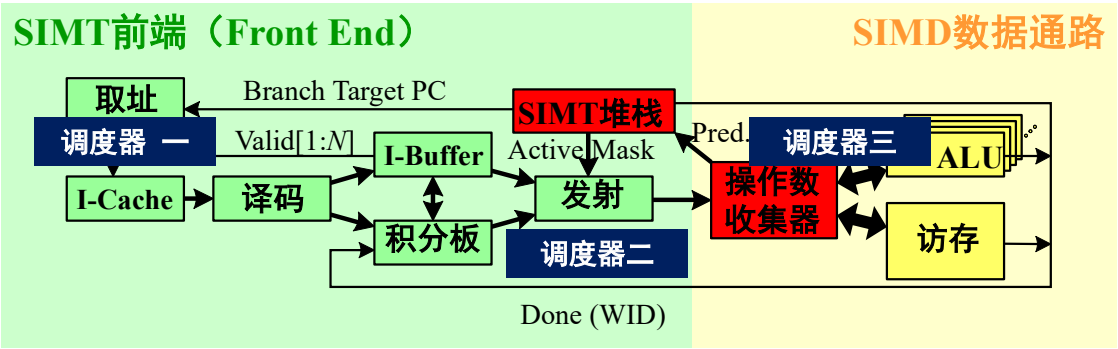
SIMT核



整体思路▸ 在“五级流水顺序核”的基础上，增加多线程并行的功能：

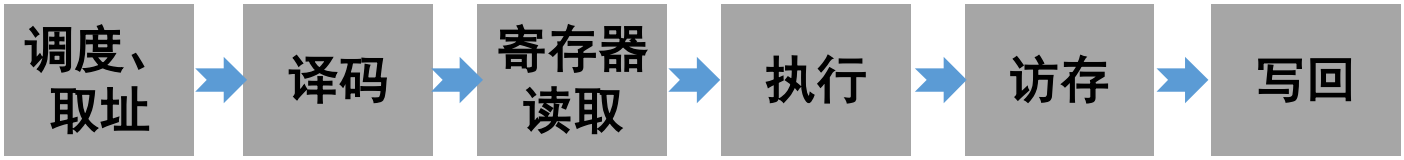
✓SIMT流水线
✓SIMT堆栈（SIMT Stack）

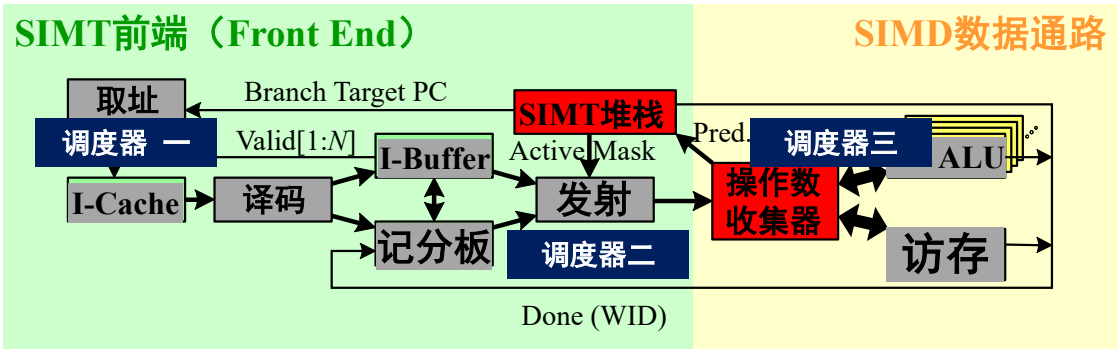
SIMT核架构



✓SIMT流水线：warp调度器（一、二、三）、记分板、操作数收集器（Operand Collector）、SIMD功能单元（ALU、访存、缓存、等）
✓SIMT堆栈（SIMT Stack）

SIMT核架构





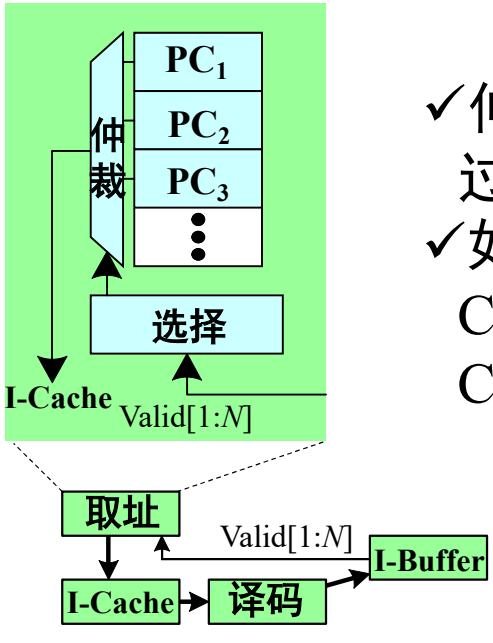
SMIT核

1. 总体架构
2. 前端
3. 操作数收集器
4. ALU、FPU、SPU、Tensor Core
5. LSU
6. CTA调度器

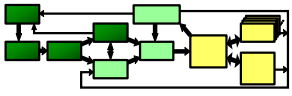
讲授内容：前端

I. 取址、译码
II. 发射
III. SIMT堆栈
IV. 记分板（scoreboard）

SIMT核架构：取址

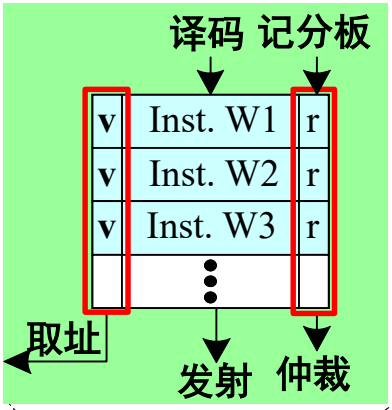


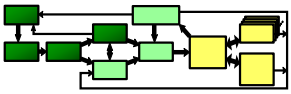
✓仲裁（arbitrate, ARB）：warp通过仲裁的方式访问I-Cache；
✓如果warp访问I-Cache，发生Cache miss，warp将再次访问I-Cache；

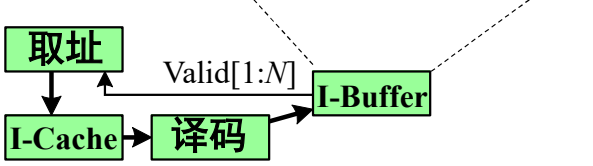


SIMT核架构：译码、I-Buffer

- ✓指令译码以后，存储在I-Buffer之中；
- ✓I-Buffer：
  - ✓由多个表项（entry）构成；
  - ✓每个表项由三部分构成：vacant、译码后指令、记分板项；
  - ✓每个warp占据一个或以上的表项（entry）。







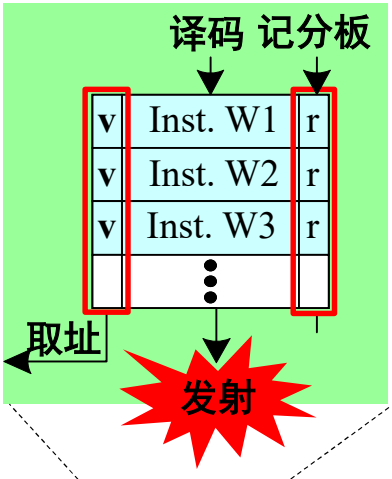
http://www.gpgpu-sim.org/

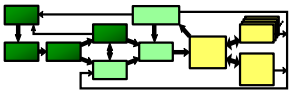
讲授内容：前端

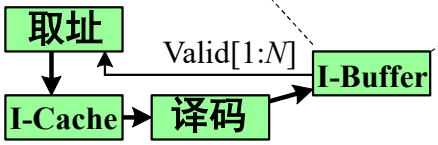
I. 取址、译码
II. 发射
III. SIMT堆栈
IV. 记分板（scoreboard）

SIMT核架构：发射

- ✓选择一个warp，并从这个warp对应的I-Buffer表项中，取出一条指令，并发射执行；
- ✓选择算法：轮询（Round-Robin Priority）
- ✓双发射



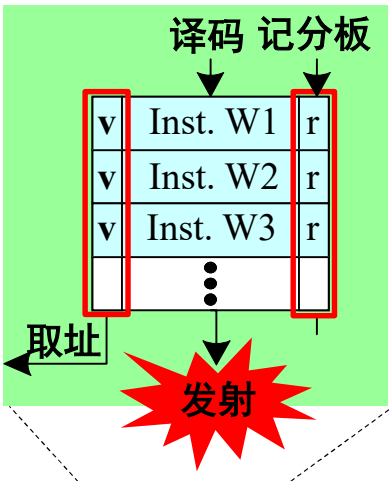


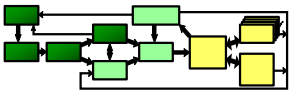


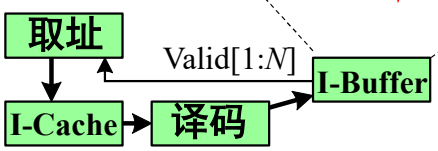
http://www.gpgpu-sim.org/

SIMT核架构：发射

- ✓发射后操作：
  - ✓在功能部件中执行（functional execution），并返回执行结果；
  - ✓合并内存操作；
  - ✓在记分板（scoreboard）中保留输出寄存器（output register ）；
  - ✓更新SIMT堆栈（SIMT stack）。





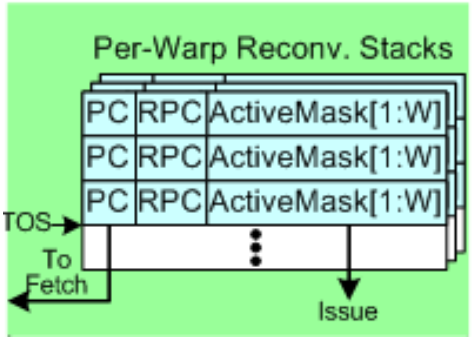


http://www.gpgpu-sim.org/

讲授内容：前端

I. 取址、译码
II. 发射
III. SIMT堆栈
IV. 记分板（scoreboard）

SIMT堆栈

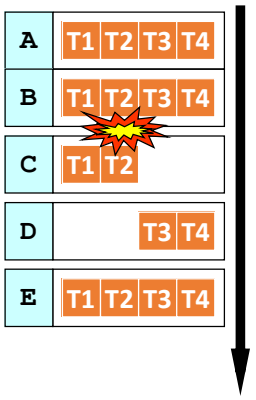


✓SIMT堆栈：Per-Warp Reconvergence Stack，用于解决同一个warp内分支发散（Branch Divergence）问题，通过mask 在分支执行过程中，将不需要执行的线程通过mask 去掉；
✓每一个warp拥有一个独立的SIMT堆栈。

http://www.gpgpu-sim.org/

SIMT堆栈

```
foo[] = {4,8,12,16};

  A: v = foo[tid.x];
  B: if (v < 10)
  C:     v = 0;
     else
  D:     v = 10;
  E: w = bar[tid.x]+v;
```



SIMT堆栈

PC	RPC	Active Mask
E	-	1111
D	E	0011
C	E	1100

- ✓ SIMT堆栈：用于解决同一个warp内分支发散（Branch Divergence）问题，通过mask在分支执行过程中，将不需要执行的线程通过mask 去掉；
- ✓ 每一个warp拥有一个独立的SIMT堆栈。

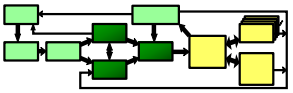
http://www.gpgpu-sim.org/

讲授内容：前端

I. 取址、译码
II. 发射
III. SIMT堆栈
IV. 记分板（scoreboard）

记分板（scoreboard）

- ✓记分板是一集中控制部件，其功能是控制数据寄存器与处理部件之间的数据传送；
- ✓在记分牌中保存有与各个处理部件相联系的寄存器中的数据装载情况；
- ✓当一个处理部件所要求的数据都已就绪（装载完毕），记分牌允许处理部件开始执行；当执行完成后，处理部件通知记分牌释放相关资源；
- ✓解决操作数的冲突问题，防止RAW和WAW冒险：对于存在RAW和WAW冒险的指令，在记分板做相应的标记。



http://www.gpgpu-sim.org/

写回

- ✓GPU流水线包含多个级：取址（fetch）、译码（decode）、发射（issue）、操作数收集（operand collection）、写回（write-back）。
- ✓CUDA核的四个主要组成部分共享同一个写回级（writeback）。

SMIT核

1. 总体架构
2. 前端
3. 操作数收集器
4. ALU、FPU、SPU、Tensor Core
5. LSU
6. CTA调度器

操作数收集器（Operand Collector）

✓设置一个缓冲区，去乱序读取指令所需要的寄存器操作数，缓冲区保证了最终指令执行的顺序；

✓解决操作数的冲突问题，防止寄存器的 bank conflict：对于存在RAW和WAW冒险的指令，在积分板做相应的标记。

✓组成 OP.COL. 的是一组缓冲器、和一个调度器。

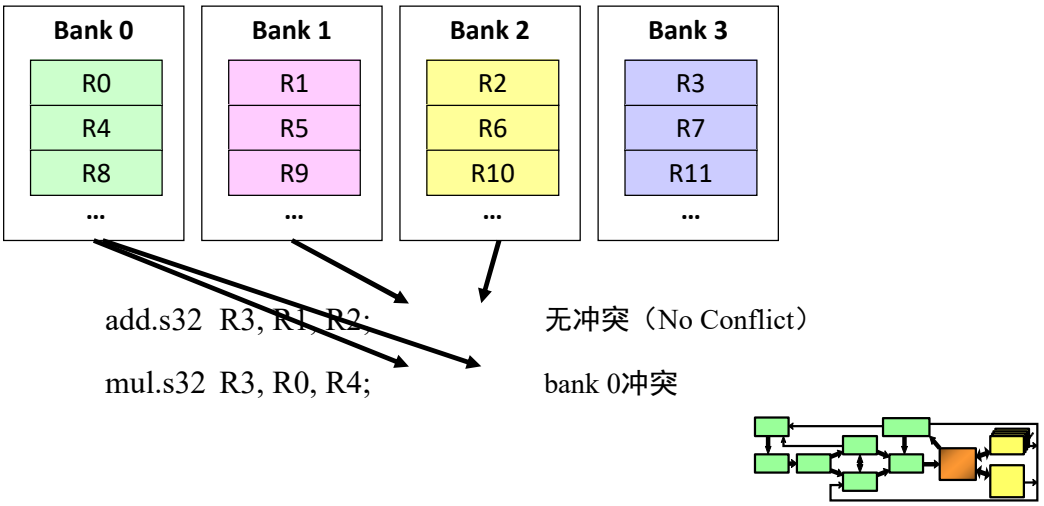
✓每当一条指令被译码后，OP.COL.便为该指令分配空间，用于取数。

✓OP.COL. 单元并没有通过寄存器换名技术来消除寄存器名字依赖，而是通过另一种方式：确保每个周期内，对一个 Bank 的访问，不得超过一次。

✓包含四个Collector Units，每个Unit包含三个操作数条目，和一个标志符，用于指示当前该Unit属于哪个Warp的哪条指令。

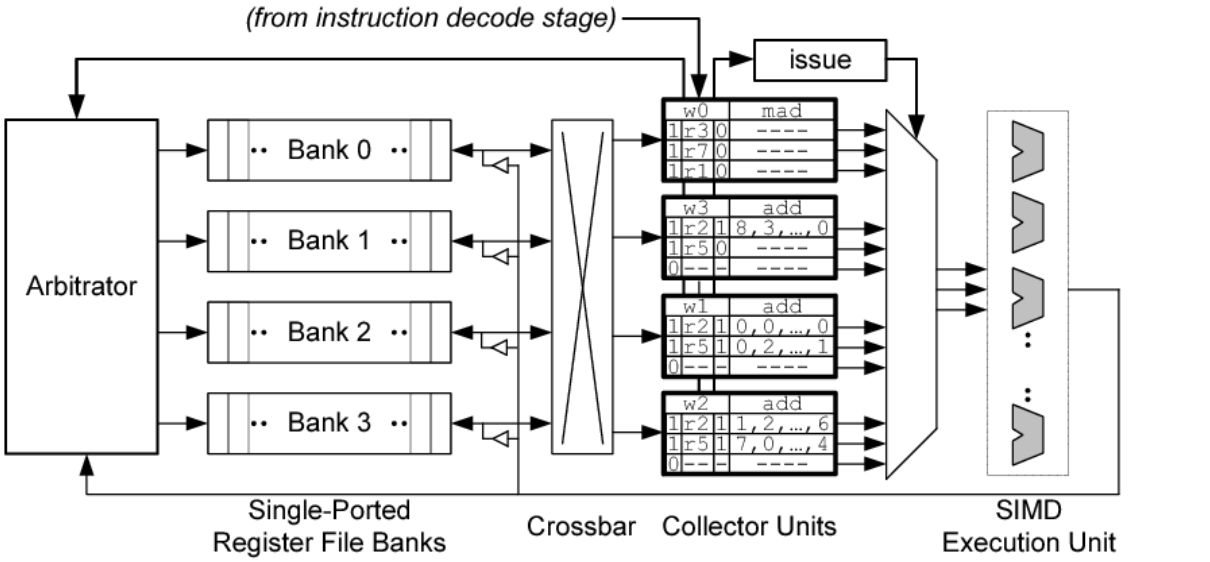
操作数收集器（Operand Collector）

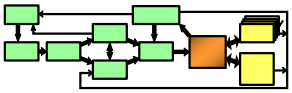
✓作用：从不同的线程中读取（fetch）操作数，从而达到最大利用效率。



http://www.gpgpu-sim.org/

操作数收集器（Operand Collector）





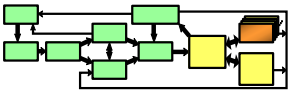
http://www.gpgpu-sim.org/

SMIT核

1. 总体架构
2. 前端
3. 操作数收集器
4. ALU、FPU、SPU、Tensor Core
5. LSU
6. CTA调度器

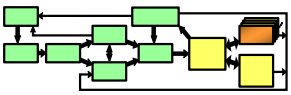
ALU Pipelines

✓ALU是CUDA核的四个主要组成部分之一；
✓ALU包含功能单元：SP、SFU
  ✓SP：SP 执行除trancedental之外的所有ALU指令，trancedental指令包括：EX2、LN2、RCP、RSQ、SIN、COS、等；
  ✓SFU：SFU 执行trancedental指令。
✓根据待执行指令的要求，ALU可以灵活配置带宽（bandwidth）和延迟（latency）。



FPU

✓FPU: Floating-Point Unit。



NVIDIA H100 Specifications

Form Factor	H100 SXM	H100 PCIe	H100 NVL[1]
FP64	34 teraFLOPS	26 teraFLOPS	68 teraFLOPs
FP64 Tensor Core	67 teraFLOPS	51 teraFLOPS	134 teraFLOPs
FP32	67 teraFLOPS	51 teraFLOPS	134 teraFLOPs
TF32 Tensor Core	989 teraFLOPS[2]	756 teraFLOPS[2]	1,979 teraFLOPs[2]
BFLOAT16 Tensor Core	1,979 teraFLOPS[2]	1,513 teraFLOPS[2]	3,958 teraFLOPs[2]
FP16 Tensor Core	1,979 teraFLOPS[2]	1,513 teraFLOPS[2]	3,958 teraFLOPs[2]
FP8 Tensor Core	3,958 teraFLOPS[2]	3,026 teraFLOPS[2]	7,916 teraFLOPs[2]
INT8 Tensor Core	3,958 TOPS[2]	3,026 TOPS[2]	7,916 TOPS[2]

https://www.nvidia.com/en-us/data-center/h100/

Tensor Core: Warp Matrix Functions

- ✓ 核心目的：利用GPU上的专用硬件单元Tensor Core，显著加速D = A * B + C形式的矩阵运算。
- ✓ 硬件要求：此功能需要计算能力为7.0或更高的NVIDIA GPU（如Volta、Turing架构及以上）。
- ✓ 精度支持：支持混合精度计算，例如输入矩阵A和B可使用FP16、BF16或INT8，而累加器C和D可使用FP32或FP64，兼顾性能与数值精度。
- ✓ 执行单元：操作以Warp（32个线程）为基本单位进行，需要Warp内所有线程协同工作。
- ✓ 编程接口：主要通过nvcuda::wmma namespace中的API（如load_matrix_sync, mma_sync, store_matrix_sync)来操作不透明的fragment对象。
- ✓ 关键限制：在条件代码中使用这些操作时，必须确保条件在整个Warp内的评估结果一致，否则会导致执行挂起。
- ✓ 性能要点：为确保最佳性能，对全局内存的访问必须满足256位对齐等特定要求。

Warp Matrix Functions: namespace

- ✓ 核心namespace：所有相关的函数和类型都定义在 nvcuda::wmma namespace中。
- ✓ 预览功能状态：子字节操作被视为预览功能，其数据结构和API未来可能会发生变化，且可能与未来版本不兼容。
- ✓ 实验性功能位置：这些额外的子字节操作功能被定义在 nvcuda::wmma::experimental namespace内。

Warp Matrix Functions: fragment类

```
template<typename Use, int m, int n, int k, typename
T, typename Layout=void> class fragment;

void load_matrix_sync(fragment<...> &a, const T*
mptr, unsigned ldm);
void load_matrix_sync(fragment<...> &a, const T*
mptr, unsigned ldm, layout_t layout);
void store_matrix_sync(T* mptr, const fragment<...>
&a, unsigned ldm, layout_t layout);
void fill_fragment(fragment<...> &a, const T& v);
void mma_sync(fragment<...> &d, const fragment<...>
&a, const fragment<...> &b, const fragment<...> &c,
bool satf=false);
```

Warp Matrix Functions: fragment类

- ✓ 定义与分布：**fragment**是一个重载类，包含矩阵的一个tile，该tile分布在一个warp的所有线程中。
- ✓ 模板参数：fragment类通过模板参数指定其用途（如matrix_a, matrix_b, accumulator）、矩阵块形状（m, n, k）、数据类型（T）以及对于输入矩阵的布局（Layout）。
- ✓ 内存布局要求：必须为matrix_a和matrix_b类型的fragment指定行主序（row_major）或列主序（col_major）布局，而累加器（accumulator）通常保留默认布局void。
- ✓ 协同操作：fragment通过load_matrix_sync从内存加载数据，通过store_matrix_sync将结果存回内存，并通过mma_sync函数与其他fragment协同执行矩阵乘加运算（D = A * B + C）。
- ✓ 内部存储不透明：矩阵元素到fragment内部存储的映射是未指定的、不透明的，并且可能随未来架构而变化，因此开发者不应直接操作其内部存储。

Warp Matrix Functions: Use参数

- ✓ 核心约束机制：该模板通过第一个名为Use的模板参数来明确指定代码fragment在矩阵运算中所扮演的特定角色，从而限制了模板参数的有效组合。
- ✓ 参数Use的取值：Use参数的可接受值被预定义为一系列有意义的选项（例如，可能包括UpperTriangle、LowerTriangle、Diagonal等），这些值对应于矩阵运算中的不同部分或特定操作模式。
- ✓ 设计意图与优势：这种设计旨在通过编译期的类型安全约束，确保只有逻辑上有效的参数组合才能被实例化，提升了代码的健壮性和可读性，避免了运行时错误。
- ✓ 使用注意事项：在使用此模板时，必须从预定义的Use值列表中进行选择，任意或不受支持的参数组合将导致编译失败，这是其保证操作语义正确的关键。

Warp Matrix Functions: use 参数定义

- ✓ **matrix_a：当fragment作为第一个被乘数矩阵 A 时使用，其tile维度为 m x k。**
- ✓ **matrix_b：当fragment作为第二个被乘数矩阵 B 时使用，其tile维度为 k x n。**
- ✓ **accumulator：当fragment用作源或目标累加器矩阵（即 C 或 D）时使用，其tile维度为 m x n。**

Warp Matrix Functions: 分块尺寸 *m*、*n*、*k*

- ✓ 定义：*m*、*n*、*k*这三个尺寸共同定义了参与乘累加运算的整个warp级别矩阵分块的形状。
- ✓ 矩阵A：对于matrix_a，其分块的维度为 m x k。
- ✓ 矩阵B：对于matrix_b，其分块的维度为 k x n。
- ✓ 累加器：累加器（accumulator）矩阵分块的维度为 $m \times n$。
- ✓ 精度支持：这些分块操作支持多种精度，例如__half、float、int 等，具体组合取决于计算能力。

Warp Matrix Functions: Description

- ✓ 数据类型 (T)：乘数（multiplicand）支持 double, float, __half, __nv_bfloat16, char, unsigned char；累加器（accumulator）支持 double, float, int, __half，但二者可用的类型组合有限。
- ✓ 布局参数 (Layout)：必须为输入矩阵 matrix_a 和 matrix_b 指定 row_major（行主序）或 col_major（列主序），这决定了矩阵行或列元素在内存中的连续方式。
- ✓ 累加器布局：累加器矩阵的布局参数应保留默认值 void，仅在其被加载或存储时才需指定行或列布局。

Warp Matrix Functions: load_matrix_sync函数

- ✓ 同步加载：load_matrix_sync函数会等待Warp（线程束）中的所有32个线程都到达调用点后，才从内存中加载矩阵数据fragment。
- ✓ 内存对齐要求：指向矩阵首元素的内存指针mptr必须是256位（32字节）对齐的。
- ✓ 步长参数ldm：参数ldm定义了矩阵在内存中连续行（对于行主序）或列（对于列主序）之间的跨度（步长），对于 __half 数据类型必须是8的倍数，对于float类型必须是4的倍数。
- ✓ 布局规则：如果加载的是累加器fragment（accumulator fragment），必须明确指定其内存布局为 mem_row_major 或 mem_col_major；而对于 matrix_a 和 matrix_b 类型的fragment，布局由fragment自身的模板参数决定。
- ✓ 线程一致性：Warp内的所有线程调用load_matrix_sync函数时，所使用的 mptr、ldm、layout 以及所有模板参数的值必须完全相同，否则结果未定义。
- ✓ load_matrix_sync函数是NVIDIA CUDA编程中利用Tensor Core进行高性能矩阵运算（Warp Matrix Multiply-Accumulate, WMMA）的关键步骤之一。

Warp Matrix Functions: store_matrix_sync函数

- ✓ 核心功能：store_matrix_sync函数用于在执行矩阵乘加运算后，将Warp 级别计算得到的矩阵片段同步地存储到全局内存中。
- ✓ 同步要求：store_matrix_sync函数会阻塞当前线程，直到Warp中的所有线程都执行到 store_matrix_sync 调用点，确保数据一致性后再进行存储操作。
- ✓ 内存对齐与指针：目标内存指针 mptr 必须指向矩阵的首元素，并且是256位对齐的，这有助于实现高效的内存访问。
- ✓ 步长与数据类型： 参数ldm定义了矩阵在内存中的步长，其值必须根据数据类型满足特定倍数要求：对于 __half 类型，ldm需是 8 的倍数；对于 float 类型，则需是 4 的倍数，以确保符合 16 字节的内存边界对齐。
- ✓ 布局：必须明确指定输出矩阵的内存布局是行优先还是列优先，通过 mem_row_major 或 mem_col_major 参数来定义。
- ✓ 线程一致性：Warp内的所有线程在调用此函数时，必须提供完全一致的参数值，包括 mptr、ldm、layout 以及矩阵片段的模板参数，这是函数正确执行的必要条件。

Warp Matrix Functions: fill_fragment函数

- ✓ 功能定义：fill_fragment 是 CUDA 的 Warp Matrix 函数之一，用于使用常量值v填充一个矩阵fragment。
- ✓ 核心要求：fill_fragment操作需要同一个 warp 内的所有线程协同调用，并且为参数v传递一个相同的值（common value）。
- ✓ 底层原因：由于矩阵元素到每个线程所持有的fragment部分的映射关系是未指定且不透明的，因此必须通过 warp 级的同步调用来确保填充的正确性。
- ✓ 主要应用：fill_fragment函数通常用于在执行矩阵乘累加（MMA）操作之前，初始化累加器片段（例如，将其设置为零）。

Warp Matrix Functions: mma_sync函数

- ✓ **核心功能：mma_sync是一个同步操作，它会等待同一个warp中的所有线程都执行到该函数后，才执行矩阵乘加运算 D = A * B + C，并支持原地计算 C = A * B + C。**
- ✓ **硬件加速：mma_sync函数利用GPU的Tensor Core来高效执行矩阵运算，支持多种精度和矩阵尺寸组合。**
- ✓ **参数一致性要求：warp内所有线程为矩阵片段（A, B, C, D）指定的模板参数（*m*, *n*, *k*）必须完全相同，satf参数的值也必须一致。**
- ✓ **执行条件：warp中的所有线程都必须调用mma_sync函数，否则会导致未定义的结果。**

Warp Matrix Functions: Description

- ✓ 关于Tensor Core中satf（saturate to finite value）模式在异常数值处理时的关键规则：
  - ✓ +Infinity的处理：若元素结果为正无穷大，则对应的累加器将饱和为正的最大规范数（+MAX_NORM）
  - ✓ -Infinity的处理：若元素结果为负无穷大，则对应的累加器将饱和为负的最大规范数（-MAX_NORM）
  - ✓ NaN的处理：若元素结果为NaN，则对应的累加器内容将被设置为正零（+0）
- ✓ 当satf模式启用时，所有非有限的异常计算结果（±Infinity和NaN）都会被强制约束到一个有限的规范数值范围内。

Warp Matrix Functions: Description

- ✓ 常规访问方式：由于矩阵元素到每个线程fragment内部的映射是未指定且不透明的，在调用store_matrix_sync将数据写回共享内存或全局内存后，必须从内存中访问单个矩阵元素。
- ✓ 直接访问的例外情况：当Warp内的所有线程需要统一地对fragment中的所有元素施加相同的逐元素操作（例如，将累加器矩阵缩放一半）时，可以直接操作fragment。
- ✓ 实现方法：直接访问通过fragment类的num_elements成员（指示该线程片段包含的元素数量）和数组成员x[]（存储元素值）来实现。

Warp Matrix Functions: Description

```
enum fragment<Use, m, n, k, T, Layout>::num_elements;
T fragment<Use, m, n, k, T, Layout>::x[num_elements];
```

- ✓ “缩放累加器矩阵为原到尺寸的一半”代码示例
  - ✓ 核心目标：该代码示例演示了如何将累加器矩阵的尺寸缩小为原来的一半
  - ✓ 关键方法：实现缩放通常涉及图像重采样，可使用如Lanczos3等算法来平衡处理速度与输出质量
  - ✓ 底层操作：此类缩放操作的本质是应用矩阵变换并对变换后的像素进行重采样，以确保图像缩放后的视觉效果

```
wmma::fragment<wmma::accumulator, 16, 16, 16, float> frag;
float alpha = 0.5f; // Same value for all threads in warp
/*...*/
for(int t=0; t<frag.num_elements; t++)
frag.x[t] *= alpha;
```

Warp Matrix Functions: bfloat16（BF16）

- ✓ 定义与结构：bfloat16（BF16）是一种16位浮点数格式，包含1位符号位、8位指数位和7位尾数位，其设计借鉴自FP32，可视为FP32的直接截断。
- ✓ 动态范围：BF16的最大优势在于其8位指数位与FP32相同，因此具有和FP32一样的宽广数值范围，能有效避免大模型训练中因数值过大或过小而导致的梯度溢出或下溢问题，相比FP16更为稳定。
- ✓ 精度：BF16的尾数位仅有7位，其数值精度低于FP16（10位尾数）和FP32，但在神经网络训练中，这种精度损失通常可以被接受。
- ✓ 性能与内存：BF16的存储空间仅为FP32的一半，有助于降低内存占用、提高计算吞吐量，并支持更大的批次大小（Batch Size），从而加速训练和推理过程。

Warp Matrix Functions: bfloat16（BF16）

- ✓ 硬件支持：BF16得到了现代AI加速硬件（如Google TPU、NVIDIA的Ampere及以上架构GPU）的原生支持，能够利用Tensor Core等专用单元实现硬件加速。
- ✓ 主要应用场景：BF16尤其适合大规模深度学习模型的训练和推理，特别是在使用Transformer架构的大语言模型（LLM）中，它已成为平衡数值稳定性与计算效率的主流选择。
- ✓ 编程实践：在CUDA编程中，可通过 <cuda_bf16.h> 头文件使用 __nv_bfloat16 数据类型，并利用Warp级矩阵操作（WMMA）API来执行高效的矩阵乘加运算。

Warp Matrix Functions: Alternate Floating Point tf32

- ✓ 定义与定位：TF32是NVIDIA专为Tensor Cores设计的一种特殊浮点格式，旨在加速深度学习计算。
- ✓ 精度与范围：TF32保持了与FP32相同的8位指数（动态范围），但将尾数精度降至10位（与FP16相同），在精度和性能间取得平衡。
- ✓ 核心用途：TF32格式主要用于Tensor Cores执行矩阵运算（如WMMA操作），以提升计算效率，通常不适用于其他通用浮点运算。
- ✓ 使用方式：使用TF32时需要手动将FP32输入数据通过__float_to_tf32等内置函数进行转换，输出数值为TF32格式。
- ✓ 重要注意：TF32应仅限于Tensor Cores运算，若与其他浮点类型操作混合，结果的精度和范围将无法保证。

Warp Matrix Functions: Alternate Floating Point tf32

- ✓ 核心用途：TF32（Tensor Float32）是一种由NVIDIA推出的数学格式，旨在使张量核心（Tensor Cores）能够高效执行FP32矩阵计算，在保持与FP32相同数值范围的同时，通过降低精度（尾数位从23位缩减至10位）来提升计算速度。
- ✓ 必要转换：要在WMMA（Warp Matrix Multiply and Accumulate）操作中使用TF32，必须先将输入的matrix_a或matrix_b手动从FP32转换为TF32精度，例如使用__float_to_tf32这样的内联函数。
- ✓ fragment与累加器要求：在load_matrix_sync操作中，需要使用指定了precision::tf32精度的fragment，并且该fragment的数据类型（element type）需设置为float。同时，参与计算的所有累加器fragment也必须具有float数据类型。
- ✓ 支持的矩阵尺寸：当使用TF32精度时，唯一支持的矩阵乘法累加运算的尺寸（m-n-k）为16x16x8。
- ✓ 存储映射：尽管在计算中使用了TF32精度，但片段的元素在存储时表示为标准的float类型，即存在从precision::tf32到float的直接映射。

Warp Matrix Functions: 双精度浮点

- ✓ 支持条件：该功能仅在计算能力（compute capability）为8.0或更高的NVIDIA GPU设备上可用。
- ✓ 使用方法：在使用时，必须将 fragment 的数据类型指定为 double，并通过 mma_sync 操作执行计算。
- ✓ 运算特性：mma_sync 运算会使用 .rn（四舍五入到最接近的偶数）舍入修饰符，以确保数值精度。

Warp Matrix Functions: Sub-byte WMMA 操作

- ✓ 功能定位：Sub-byte WMMA操作提供了访问Tensor Core 低精度计算能力的途径，主要用于处理4位或1位等低于字节宽度的数据。
- ✓ 预览特性：WMMA操作功能被视为预览特性（preview feature），其数据结构和 API 可能会发生变化，并且可能与未来版本不兼容。
- ✓ namespace：此功能通过 nvcuda::wmma::experimental namespace提供，以区别于稳定的 WMMA API。
- ✓ 支持的数据类型：实验性精度类型包括 4 位无符号整数（precision::u4）、4 位有符号整数（precision::s4）以及 1 位数据（precision::b1）。

Warp Matrix Functions: Sub-byte WMMA 操作

- ✓ 支持的矩阵尺寸：例如，对于4位精度，支持的矩阵乘法累加尺寸（m-n-k）为8x8x32。
- ✓ 数据打包存储：由于子字节数据尺寸小，多个元素会打包在一个存储单元中，例如8个4位元素打包在一个32位整数中。
- ✓ 专用API：使用bmma_sync函数执行同步的位矩阵乘法累加运算，支持如XOR或AND等位操作。
- ✓ 布局限制：对于子字节操作，矩阵A的布局必须为行主序（row_major），矩阵B必须为列主序（col_major）。
- ✓ 内存对齐要求：在加载矩阵时，源指针必须256位对齐，并且跨度参数ldm需满足特定对齐要求（如对于4位类型是32的倍数）。

Warp Matrix Functions: Sub-byte WMMA 操作

```
namespace experimental {
    namespace precision {
        struct u4; // 4-bit unsigned
        struct s4; // 4-bit signed
        struct b1; // 1-bit
    }
    enum bmmaBitOp {
        bmmaBitOpXOR = 1, // compute_75 minimum
        bmmaBitOpAND = 2  // compute_80 minimum
    };
    enum bmmaAccumulateOp { bmmaAccumulateOpPOPC = 1 };
}
```

Warp Matrix Functions: Sub-byte WMMA 操作

- ✓ 指定数据类型：使用4位精度时，API保持不变，但必须将fragment数据类型指定为experimental::precision::u4（无符号）或experimental::precision::s4（有符号）。
- ✓ 元素打包存储：由于fragment中的元素是打包存储的，其存储元素数量（num_storage_elements）将少于实际元素数量（num_elements）。
- ✓ 子字节元素计数：对于子字节（如4位）fragment，num_elements返回的是element_type<T>类型元素的数量。
- ✓ 单比特精度映射：此规则同样适用于单比特精度，其element_type<T>到storage_element_type<T>的映射遵循类似机制。

Warp Matrix Functions: Sub-byte WMMA 操作

```
experimental::precision::u4 -> unsigned (8 elements in 1 storage element)
experimental::precision::s4 -> int (8 elements in 1 storage element)
experimental::precision::b1 -> unsigned (32 elements in 1 storage element)
T -> T  //all other types
```

- ✓ 内存布局：在进行Sub-byte（低于8位）精度的Warp Matrix Multiply-Accumulate（WMMA）操作时，矩阵A的片段必须始终使用行优先布局（row_major），而矩阵B的片段必须始终使用列优先布局（col_major）。
- ✓ 内存对齐：在调用load_matrix_sync函数加载Sub-byte矩阵时，参数ldm（引导维度）的值必须满足特定的对齐要求。对于元素类型为experimental::precision::u4或experimental::precision::s4（4位精度）的情况，ldm必须是32的倍数；对于元素类型为experimental::precision::b1（1位精度）的情况，ldm必须是128的倍数。这两种情况均是为了保证16字节的内存对齐。

Warp Matrix Functions: bmma_sync操作

- ✓基本功能：bmma_sync是一个warp级别的同步指令，用于执行位矩阵乘累加运算D = (A op B) + C。
- ✓逻辑操作选项：操作op支持两种位逻辑运算：bmmaBitOpXOR（异或）和bmmaBitOpAND（与），后者要求设备计算能力≥ 8.0。
- ✓累加操作：累加步骤固定为bmmaAccumulateOpPOPC，即对逻辑运算结果中置位（为1）的比特进行计数。

Warp Matrix Functions: 限制条件

- ✓ 架构差异性：Tensor Core所需的特殊数据格式（special format）会因GPU主次设备架构（major and minor device architecture）的不同而存在差异。
- ✓ 不透明片段：线程（threads）并不持有完整的矩阵，而是持有整个矩阵的一个不透明的、与特定架构相关的ABI数据结构片段（opaque architecture-specific ABI data structure fragment）。
- ✓ 禁止映射假设：开发者（developer）不被允许对单个参数如何映射到参与矩阵乘累加（matrix multiply-accumulate, MMA）操作的寄存器做出任何假设（not allowed to make assumptions），这增加了编程的复杂性。

Warp Matrix Functions: fragment跨架构传递风险

- ✓ 核心风险：在不同链接兼容架构（如sm_70和sm_75）的编译单元间直接传递fragment是不安全的。
- ✓ 根本原因：fragment的大小和内存布局是特定于架构的，不同架构间的ABI不兼容。
- ✓ 潜在后果：在非原生架构上使用fragment会导致计算结果错误或数据损坏。
- ✓ 安全替代方案：应通过store_matrix_sync将数据存入内存，再以指针方式传递，而非直接传递Fragment对象本身。
- ✓ 简短总结：为确保计算正确性，应避免在不同架构编译的代码间直接传递fragment对象，而应通过全局内存进行数据交换。

Warp Matrix Functions: Element Types and Matrix Sizes

✓ Tensor Cores支持的数据类型与矩阵尺寸：

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
__half	__half	float	16x16x16
__half	__half	float	32x8x16
__half	__half	float	8x32x16
__half	__half	__half	16x16x16
__half	__half	__half	32x8x16
__half	__half	__half	8x32x16
unsigned char	unsigned char	int	16x16x16
unsigned char	unsigned char	int	32x8x16
unsigned char	unsigned char	int	8x32x16
signed char	signed char	int	16x16x16
signed char	signed char	int	32x8x16
signed char	signed char	int	8x32x16

Warp Matrix Functions: Element Types and Matrix Sizes

✓ Floating Point support:

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
__nv_bfloat16	__nv_bfloat16	float	16x16x16
__nv_bfloat16	__nv_bfloat16	float	32x8x16
__nv_bfloat16	__nv_bfloat16	float	8x32x16
precision::tf32	precision::tf32	float	16x16x8

Warp Matrix Functions: Element Types and Matrix Sizes

✓ Double Precision Support:

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
double	double	double	8x8x4

Warp Matrix Functions: Example

```
//The following code implements a 16x16x16 matrix
multiplication in a single warp

#include <mma.h>
using namespace nvcuda;

__global__ void wmma_ker(half *a, half *b, float *c) {
   // Declare the fragments
   wmma::fragment<wmma::matrix_a, 16, 16, 16, half,
wmma::col_major> a_frag;
   wmma::fragment<wmma::matrix_b, 16, 16, 16, half,
wmma::row_major> b_frag;
   wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;
```

Warp Matrix Functions: Example

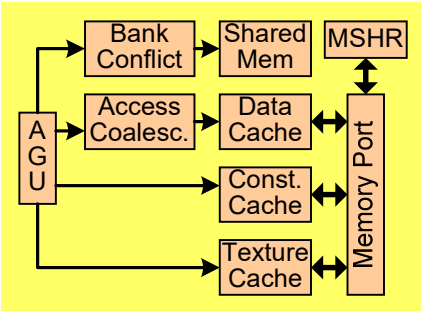
```
//The following code implements a 16x16x16 matrix multiplication
in a single warp

   // Initialize the output to zero
   wmma::fill_fragment(c_frag, 0.0f);

   // Load the inputs
   wmma::load_matrix_sync(a_frag, a, 16);
   wmma::load_matrix_sync(b_frag, b, 16);

   // Perform the matrix multiplication
   wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

   // Store the output
   wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
}
```

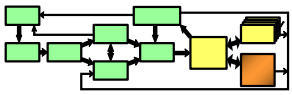
SMIT核

1. 总体架构
2. 前端
3. 操作数收集器
4. ALU、FPU、SPU、Tensor Core
5. LSU
6. CTA调度器

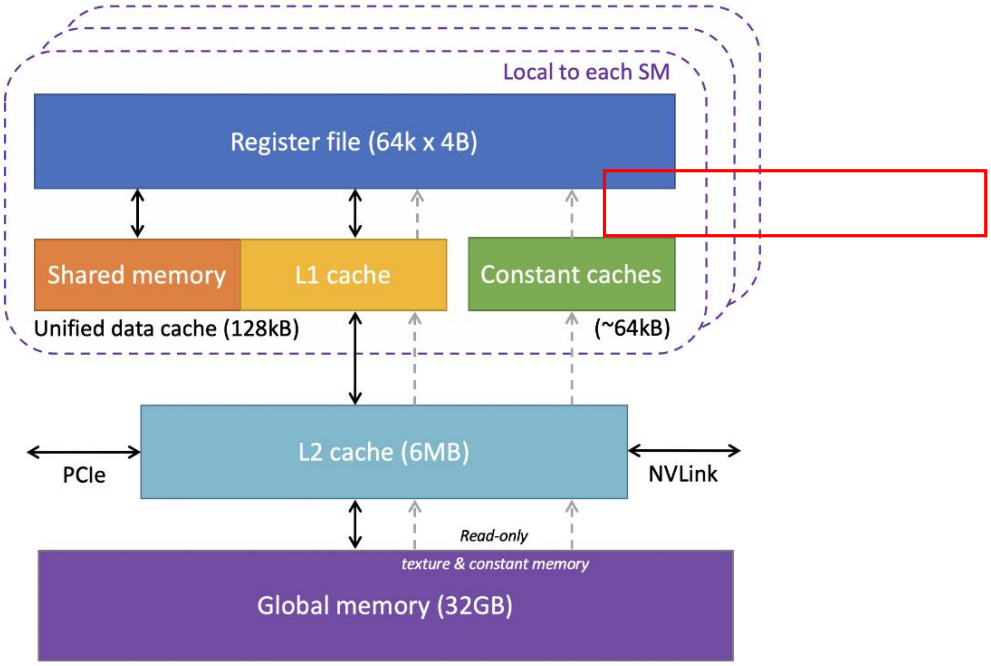
存储单元

内存类型	读写权限	PTX权限
共享内存 L1 Cache	读写	CUDA shared memory accesses only
常量Cache	只读	Constant memory and parameter memory
纹理Cache	只读	Texture accesses only
数据Cache	读写（全局内存：evict-on-write；本地内存：写回）	Global and Local memory accesses





寄存器文件（Register File）

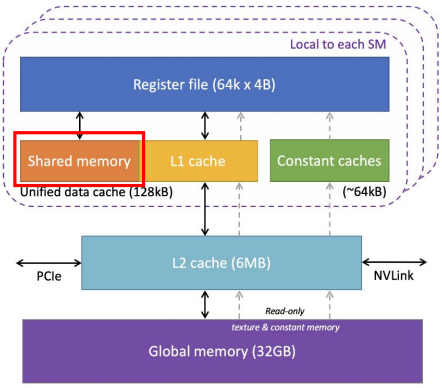


共享内存（Shared Memory）

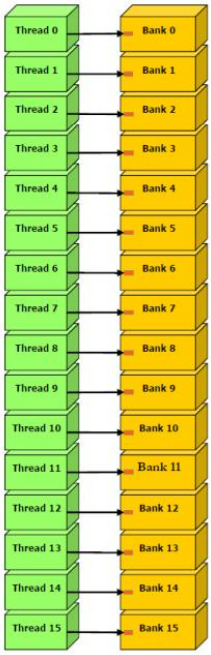
- ✓共享内存（Shared Memory）是显式管理的便签内存（scratchpad memory）；
- ✓同一个block的thread可以通过共享内存（Shared Memory）相互通信；
- ✓每个SIMT核具有独立的共享内存（Shared Memory），可以动态分配给线程块（thread block）；
- ✓因为共享内存（Shared Memory）需要服务的thread数量很大，所以共享内存（Shared Memory）包含的bank数量也很大；

共享内存（Shared Memory）

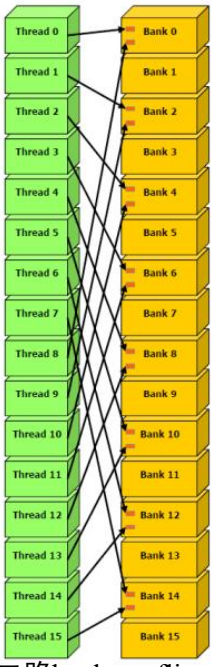
- ✓在同一个时钟周期内，每个thread只能读写一个bank中的一个内存地址；
- ✓在同一个时钟周期内，多个thread读写一个bank中的同一个内存地址，造成bank conflict，多个thread的读写请求将被顺序执行；
- ✓NVIDIA公司GPU，共享内存（Shared Memory）的bank数量可以配置成16或者32；



共享内存（Shared Memory）



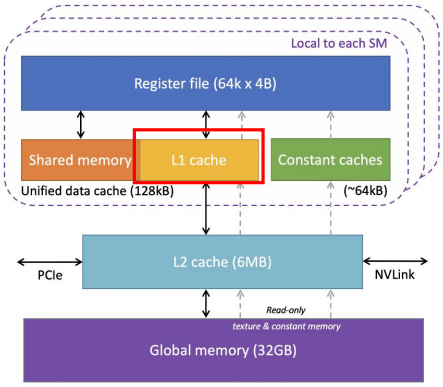
无bank conflict



二路bank conflict

L1 Cache

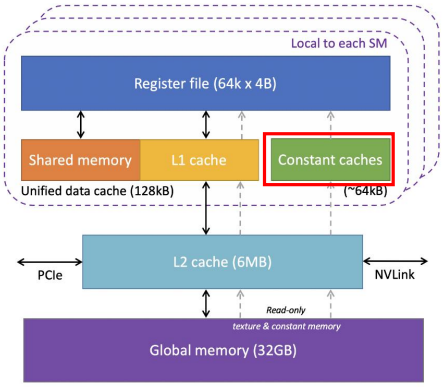
- ✓服务对象：local memory和global memory；
- ✓非一致性，无缓存一致性机制；
- ✓单端口，128位宽；
- ✓对于非合并的内存读写，需要多个周期完成。
- ✓替换策略：



	Local Memory	Global Memory
Write Hit	Write-back	Write-evict
Write Miss	Write no-allocate	Write no-allocate

常量Cache（Constant Cache）

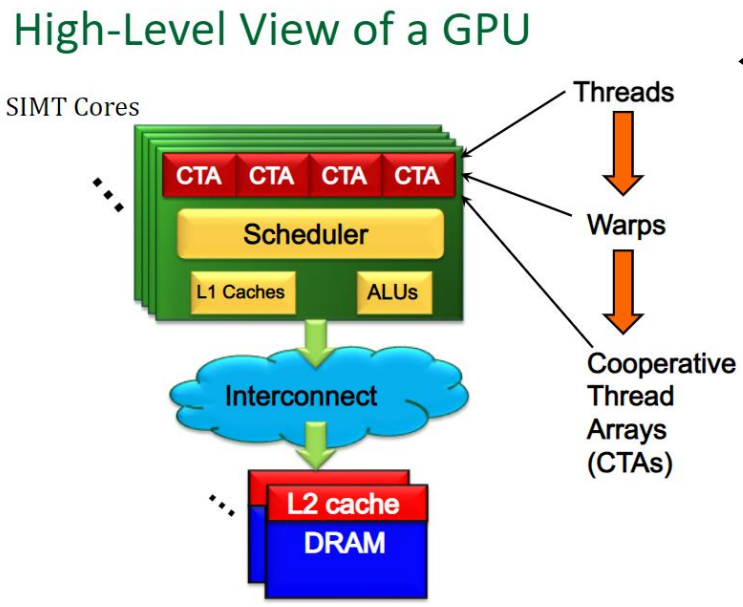
✓只读（Read Only）属性；
✓具有多个读端口，GPGPU-Sim模拟一个常量Cache（Constant Cache）的读端口。



SMIT核

1. 总体架构
2. 前端
3. 操作数收集器
4. ALU、FPU、SPU、Tensor Core
5. LSU
6. CTA调度器

协作线程阵列（CTA）调度



✓协作线程阵列（CTA）是GPU作业分配的基本单元。

OWL: Cooperative Thread Array (CTA) Aware Scheduling Techniques for Improving GPGPU Performance Adwait Jog, Onur Kayiran, Nachiappan CN, Asit Mishra, Mahmut.

协作线程阵列（CTA）调度

✓协作线程阵列（CTA）一个一个的发射到SIMT核执行；
✓在SIMT核的每个时钟周期，协作线程阵列（CTA）以轮询方式（round robin）选择和通过 SIMT 核集群；
✓对于每个选定的SIMT核，如果该SIMT核上有足够的可用资源，将向这个SIMT核发送一个协作线程阵列（CTA）。

✓GPU具有同时执行多个kernel的能力；
✓每一个kernel可以在不同的SIMT核上切换执行；
✓对于每一个SIMT核来说，如果分配有多个kernel等待执行，那么SIMT核以轮询方式（round robin）选择一个kernel来执行。


THANKS