CHAPTER2
CUDA编程入门

主　讲：韩君

单　位：早稻田大学

公众号：自动驾驶之心

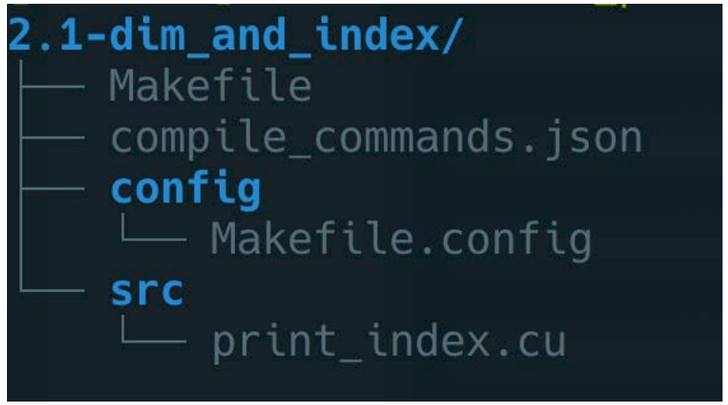
主要内容

1. CUDA中的线程与线程束
2. 使用CUDA进行MATMUL计算
3. 共享内存以及BANK CONFLICT
4. STREAM和EVENT
5. 使用CUDA进行预处理/后处理

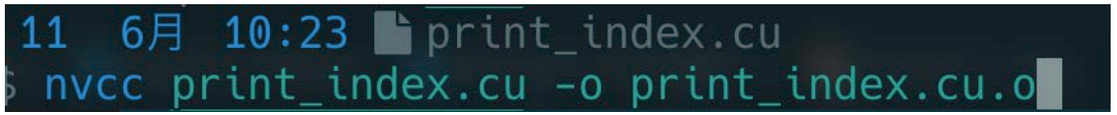
CUDA中的线程与线程束

Goal: 理解在CUDA中一维二维三维的grid, block的写法，以及遍历thread的方法

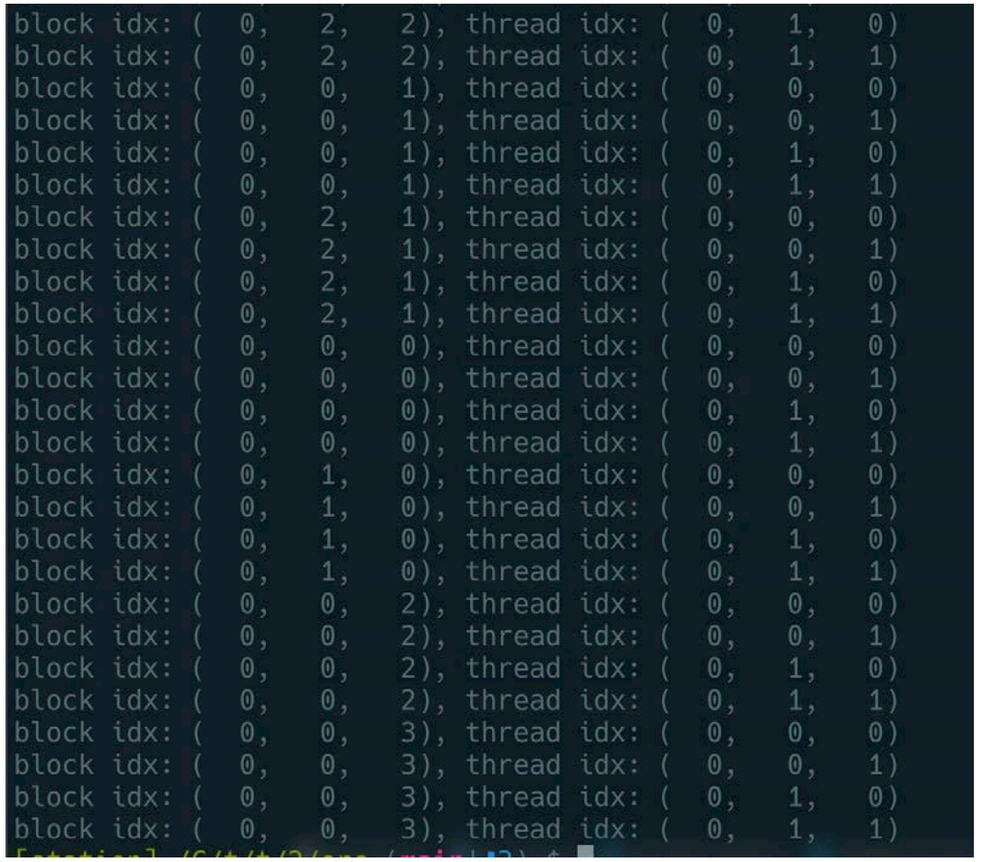
执行一下我们的第一个CUDA程序



2.1-dim_and_index
(先不要管compile_commands.json，这个是neovim下进行c++函数定义声明调用跳转的东西。类似与.vscode)



先不要用make，直接手敲一下nvcc指令。
(nvcc是cuda的编译器，定位和gcc, g++是一样的)



执行结果的一部分
(这个是显示一个核函数的grid和block信息的一个程序)

执行一下我们的第一个CUDA程序

```
int main() {
  /*
    synchronize是同步的意思，有几种synchronize

    cudaDeviceSynchronize: cpu端停止执行，知道gpu端完
    cudaStreamSynchronize: 跟cudaDeviceSynchronize很像
    cudaThreadSynchronize: 现在已经不被推荐使用的方法
    __syncthreads:         线程块内同步
  */
  // print_one_dim();
  print_two_dim();
  return 0;
}
```

```
__global__ void print_idx_kernel(){
  printf("block idx: (%3d, %3d, %3d), thread idx: (%3d, %3d, %3d)\n",
         blockIdx.z, blockIdx.y, blockIdx.x,
         threadIdx.z, threadIdx.y, threadIdx.x);
}
```

跟以往的c/c++程序中不一样的几个点

- 文件的后缀名是.cu
- <<<>>>的出现
- __global__

```
void print_two_dim(){
  int inputWidth = 8;

  int blockDim = 2;
  int gridDim = inputWidth / blockDim;

  dim3 block(blockDim, blockDim);
  dim3 grid(gridDim, gridDim);

  print_idx_kernel<<<grid, block>>>();
  // print_dim_kernel<<<grid, block>>>();
  // print_thread_idx_per_block_kernel<<<grid, block
  // print_thread_idx_kernel<<<grid, block>>>();

  cudaDeviceSynchronize();
}
```

执行一下我们的第一个CUDA程序

```
CONFIG           :=  ../config/Makefile.config
CONFIG_LOCAL     :=  ./config/Makefile.config

include $(CONFIG)
include $(CONFIG_LOCAL)


BUILD_PATH    :=  build
SRC_PATH      :=  src
CUDA_DIR      :=  /usr/local/cuda-$(CUDA_VER)

KERNELS_SRC   :=  $(wildcard $(SRC_PATH)/*.cu)

APP_OBJS      +=  $(patsubst $(SRC_PATH)%, $(BUILD_PATH)%, $(KERNELS_SRC:.cu=.cu.o))

APP_MKS       :=  $(APP_OBJS:.o=.mk)

APP_DEPS      +=  $(KERNELS_SRC)



CUCC          :=  $(CUDA_DIR)/bin/nvcc
CUDAFLAGS     :=  -O3 --shared -Xcompiler -fPIC \
                  -I $(CUDA_DIR)/include \

INCS          :=  -I $(CUDA_DIR)/include \
                  -I $(SRC_PATH)

LIBS          := -L "$(CUDA_DIR)/lib64" \

ifeq ($(DEBUG),1)
CUDAFLAGS     +=  -g -O0
else
endif

ifeq ($(SHOW_WARNING),1)
CUDAFLAGS     +=  -Wall -Wunused-function -Wunused-variable -Wfatal-errors
endif
```

Makefile
(各种变量的声明，比如说编译器，flags等等)

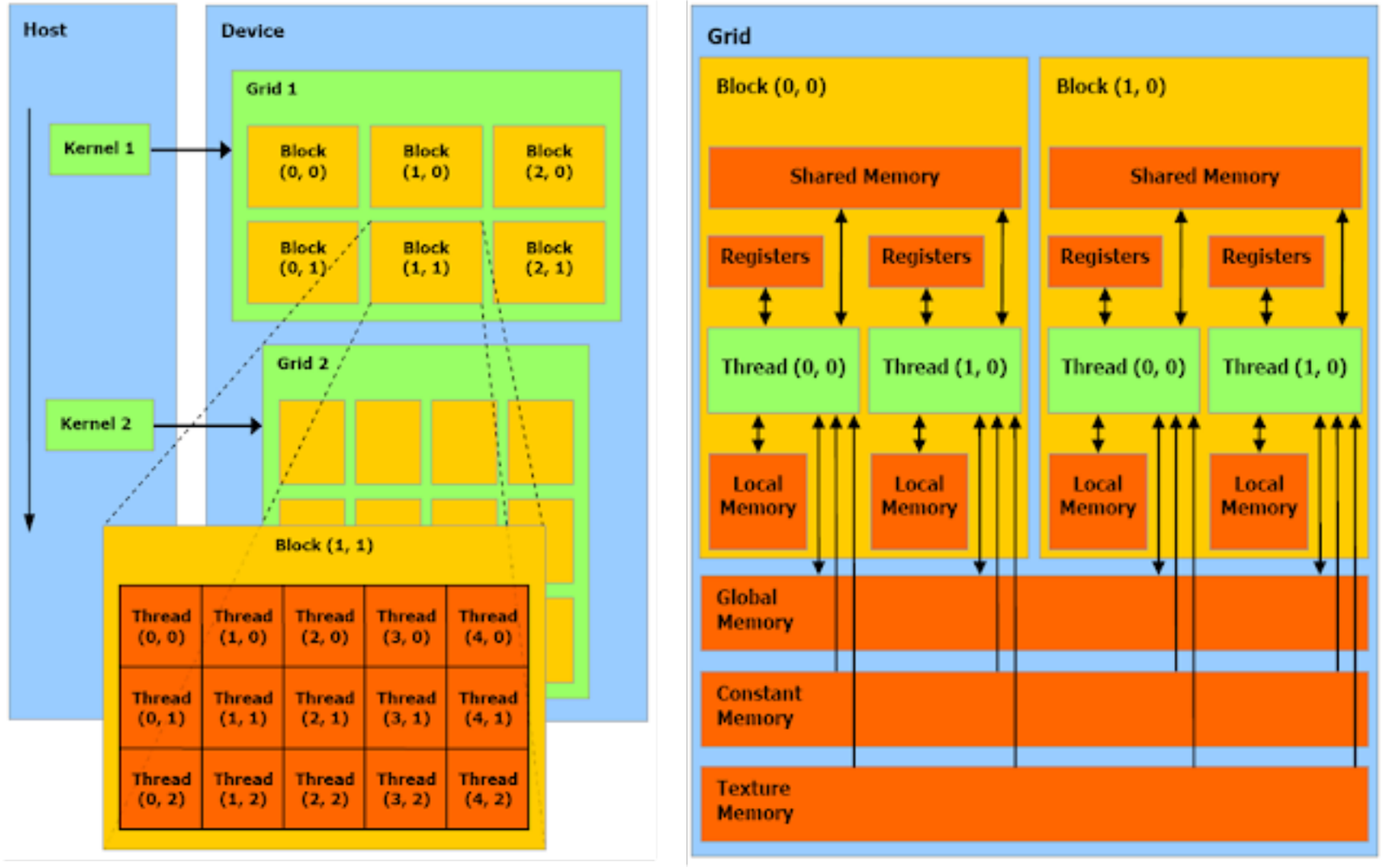
```
all:
  $(MAKE) $(APP)

update: $(APP)
  @echo finished updating $<

$(APP): $(APP_DEPS) $(APP_OBJS)
  @$(CUCC) $(APP_OBJS) -o $@ $(LIBS) $(INCS)
  @echo finished building $@. Have fun!!

show:
  @echo $(BUILD_PATH)
  @echo $(APP_DEPS)
  @echo $(INCS)
  @echo $(APP_OBJS)
  @echo $(APP_MKS)

clean:
  rm -rf $(APP)
  rm -rf build

ifneq ($(MAKECMDGOALS), clean)
-include $(APP_MKS)
endif
```

Makefile
(目标和对应的依赖关系，make all, make update, make show...)

```
# Compile CUDA
$(BUILD_PATH)/%.cu.o: $(SRC_PATH)/%.cu
  @echo Compile CUDA $@
  @mkdir -p $(BUILD_PATH)
  @$(CUCC) -o $@ -c $< $(CUDAFLAGS) $(INCS)
```

Makefile
(使用nvcc对.cu程序进行编译)

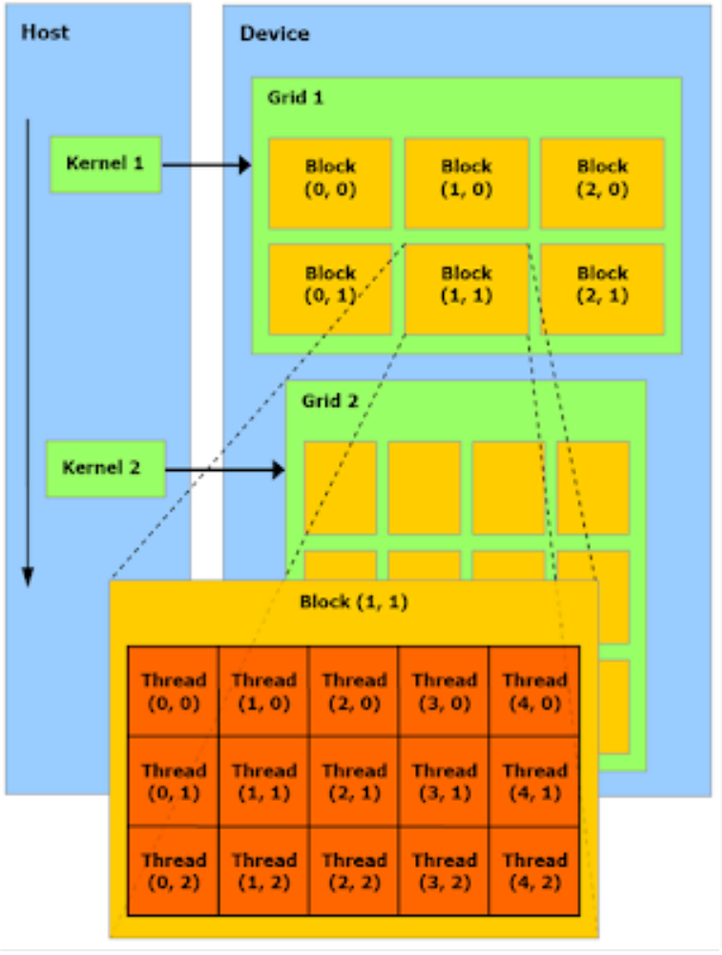
CUDA中的grid和block

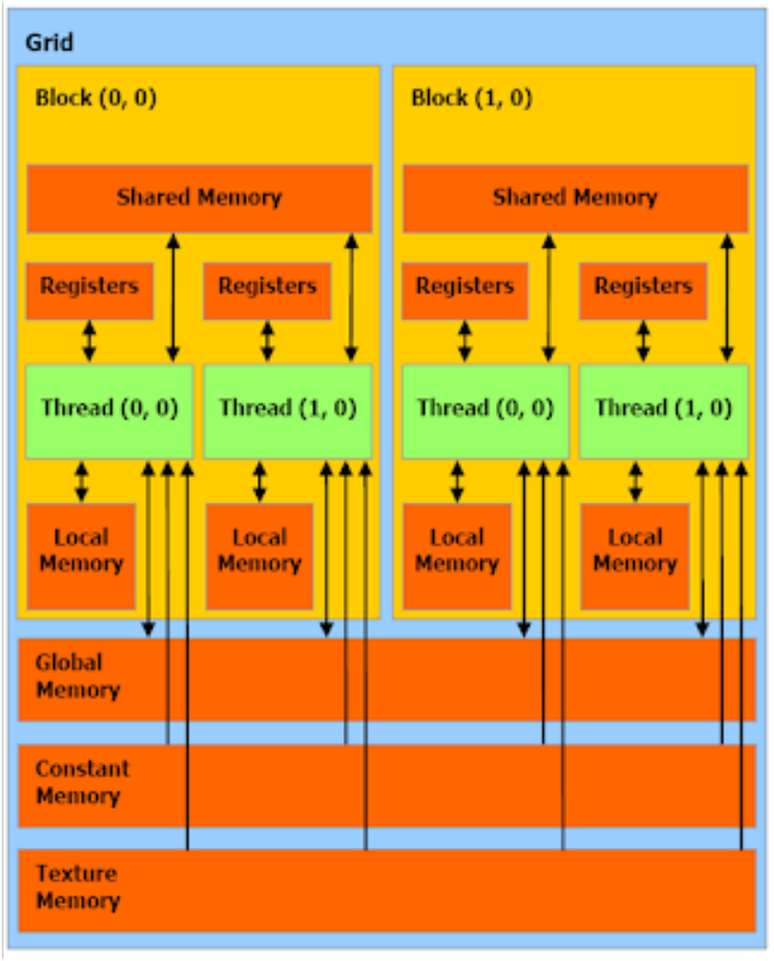


CUDA编程中的grid和block的概念[1]
启动一个kernel的时候需要指定grid和block

[1]Thread and block heuristics in CUDA programming

CUDA中的grid和block





一般来说：

- 一个kernel对应一个grid
- 一个grid可以有多个block，一维~三维
- 一个block可以有多个thread，一维~三维

[1]Thread and block heuristics in CUDA programming

block和thread的遍历(traverse)

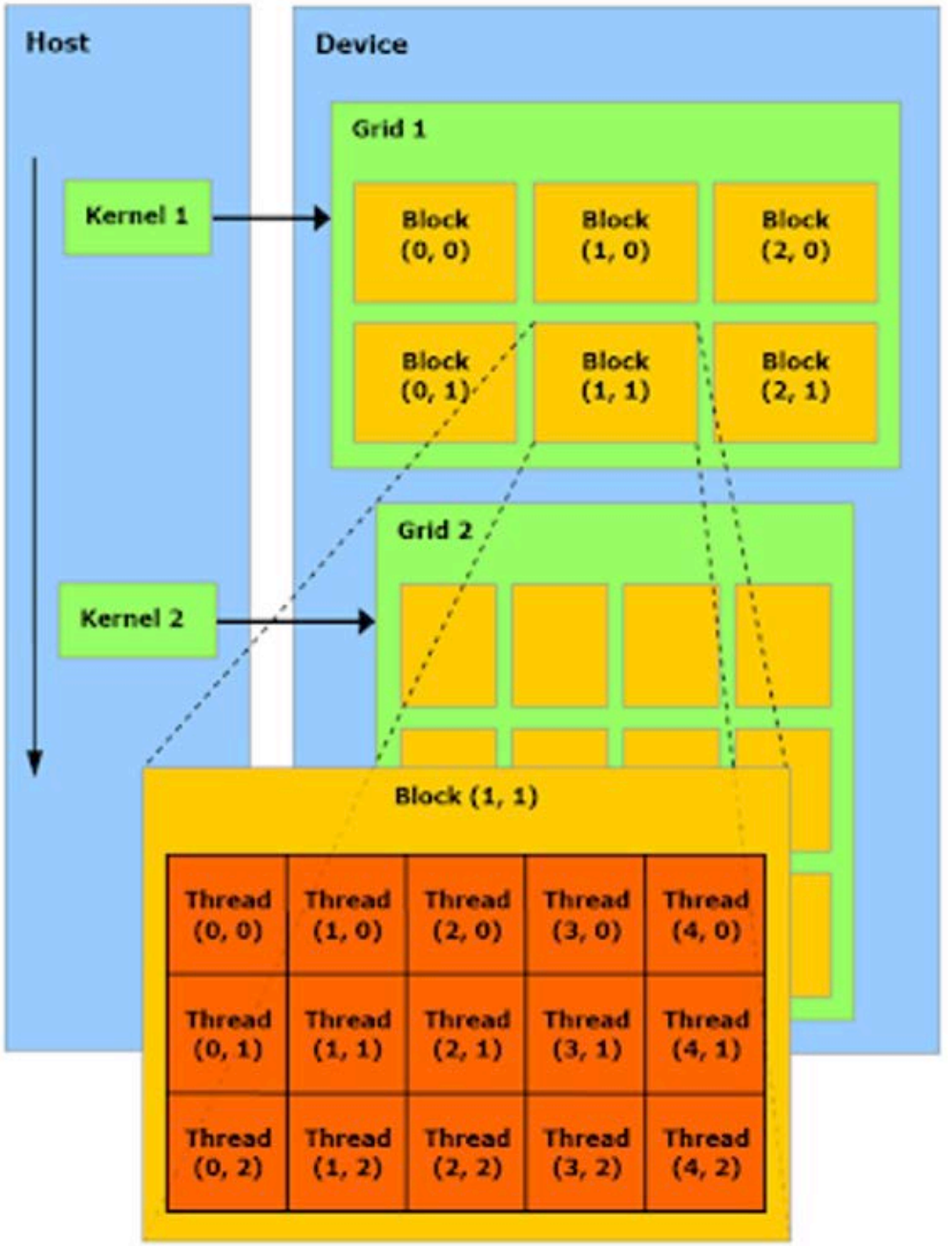
```
void print_one_dim( ){
  int inputSize = 8;

  int blockDim = 4;
  int gridDim = inputSize / blockDim;

  dim3 block(blockDim);
  dim3 grid(gridDim);
```

block和thread的遍历(traverse)

可以按照各个维度进行遍历。(各个维度上的索引)

- blockIdx.z, blockIdx.y, blockIdx.x
- threadIdx.z, threadIdx.y, threadIdx.x

```
__global__ void print_idx_kernel(){
    printf("block idx: (%3d, %3d, %3d), thread idx: (%3d, %3d, %3d)\n",
           blockIdx.z, blockIdx.y, blockIdx.x,
           threadIdx.z, threadIdx.y, threadIdx.x);
}
```

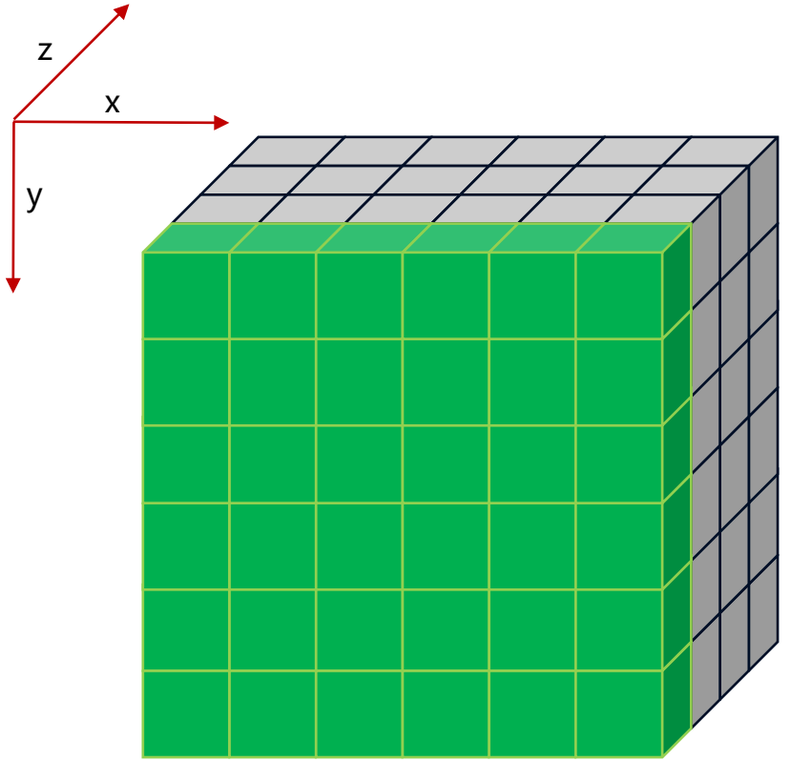


block和thread的遍历(traverse)

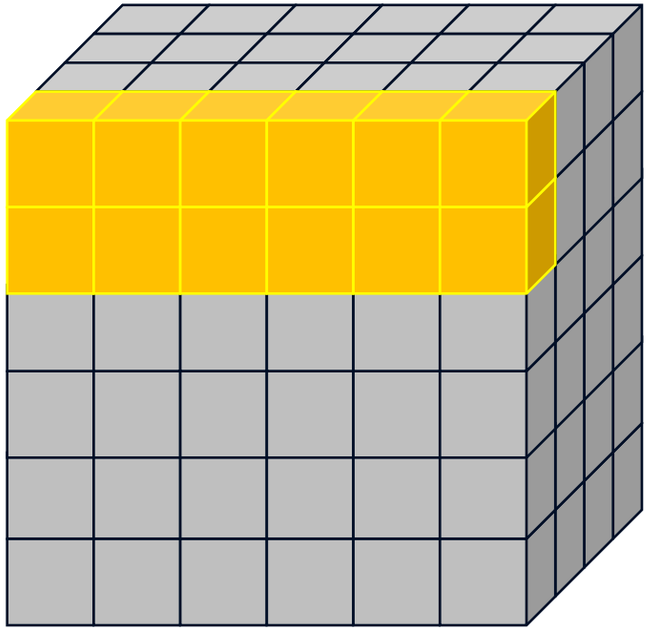
可以按照各个维度进行遍历。(一维索引)

- blockIdx.z, blockIdx.y, blockIdx.x
- threadIdx.z, threadIdx.y, threadIdx.x
- gridDim.z, gridDim.y, gridDim.x
- blockDim.z, blockDim.y, blockDim.x

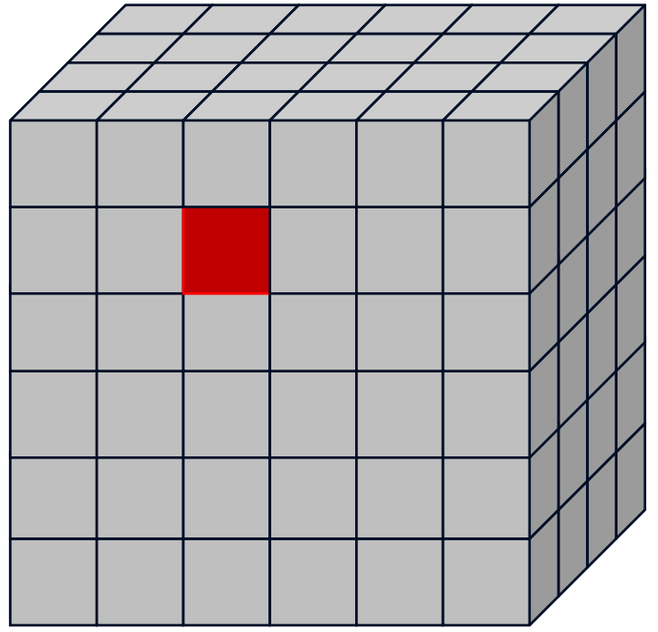
```
__global__ void print_thread_idx_per_block_kernel(){
  int index = threadIdx.z * blockDim.x * blockDim.y + \
              threadIdx.y * blockDim.x + \
              threadIdx.x;

  printf("block idx: (%3d, %3d, %3d), thread idx: %3d\n",
         blockIdx.z, blockIdx.y, blockIdx.x,
         index);
}
```



先遍历z方向
(每一个z方向都有blockDim.x * blockDim.y的thread)

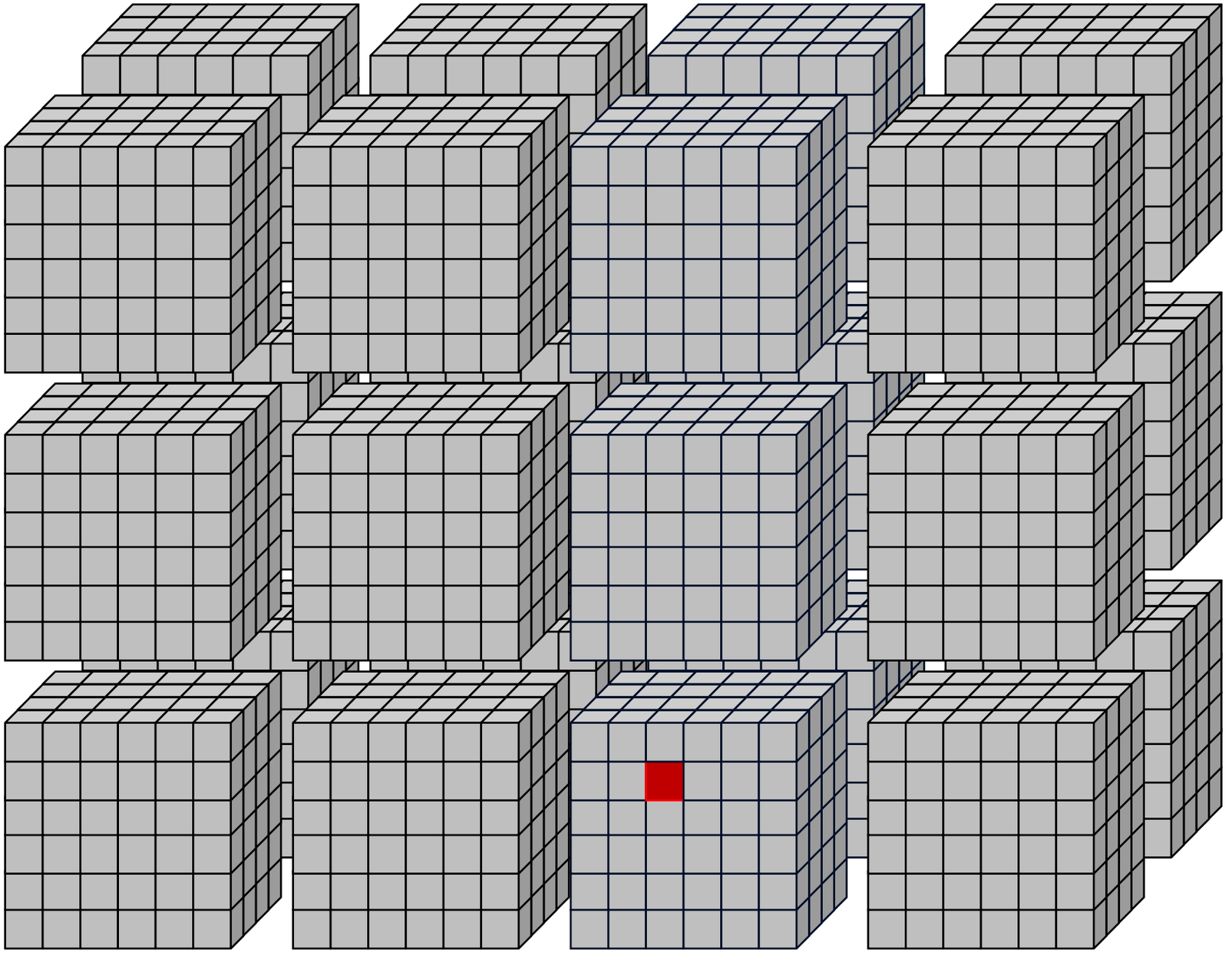


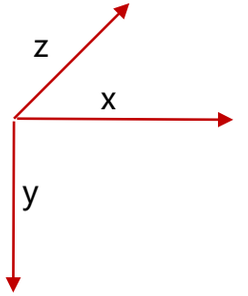
先遍历y方向
(每一个y方向都有blockDim.x的thread)



先遍历x方向
(threadIdx就是这个thread在当前block下的索引)

block和thread的遍历(traverse)

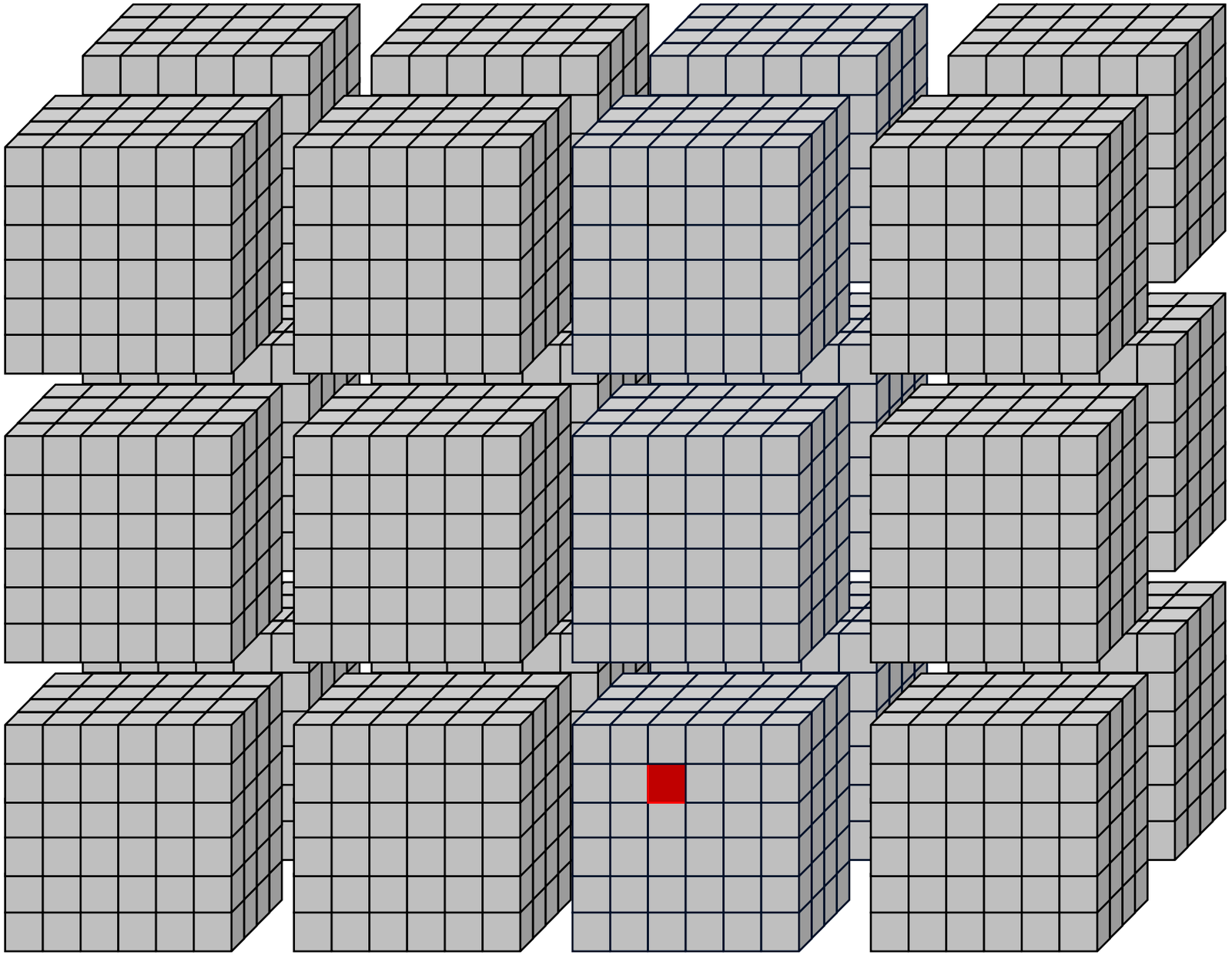


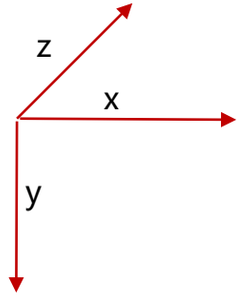



```
__global__ void print_thread_idx_kernel(){
  int bSize  = blockDim.z * blockDim.y * blockDim.x;

  int bIndex = blockIdx.z * gridDim.x * gridDim.y + \
               blockIdx.y * gridDim.x + \
               blockIdx.x;

  int tIndex = threadIdx.z * blockDim.x * blockDim.y + \
               threadIdx.y * blockDim.x + \
               threadIdx.x;

  int index  = bIndex * bSize + tIndex;

  printf("block idx: %3d, thread idx in block: %3d, thread idx: %3d\n",
         bIndex, tIndex, index);
}
```

同样的，我们也可以在grid维度下寻找对应的thread。可以感觉到，在grid中按照这样的方式寻找thread的一维地址有点麻烦

block和thread的遍历(traverse, cord)





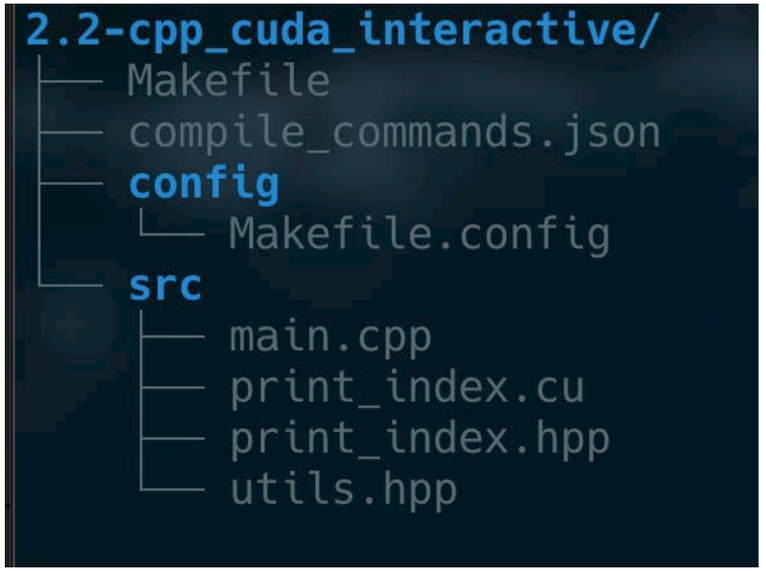

```cpp
__global__ void print_cord_kernel(){
  int index = threadIdx.z * blockDim.x * blockDim.y + \
              threadIdx.y * blockDim.x + \
              threadIdx.x;

  int x  = blockIdx.x * blockDim.x + threadIdx.x;
  int y  = blockIdx.y * blockDim.y + threadIdx.y;

  printf("block idx: (%3d, %3d, %3d), thread idx: %3d, cord: (%3d, %3d)\n",
         blockIdx.z, blockIdx.y, blockIdx.x,
         index, x, y);

}
```

一般来说推荐这样寻找坐标。使用CUDA在图像中进行寻找坐标的时候推荐这么使用

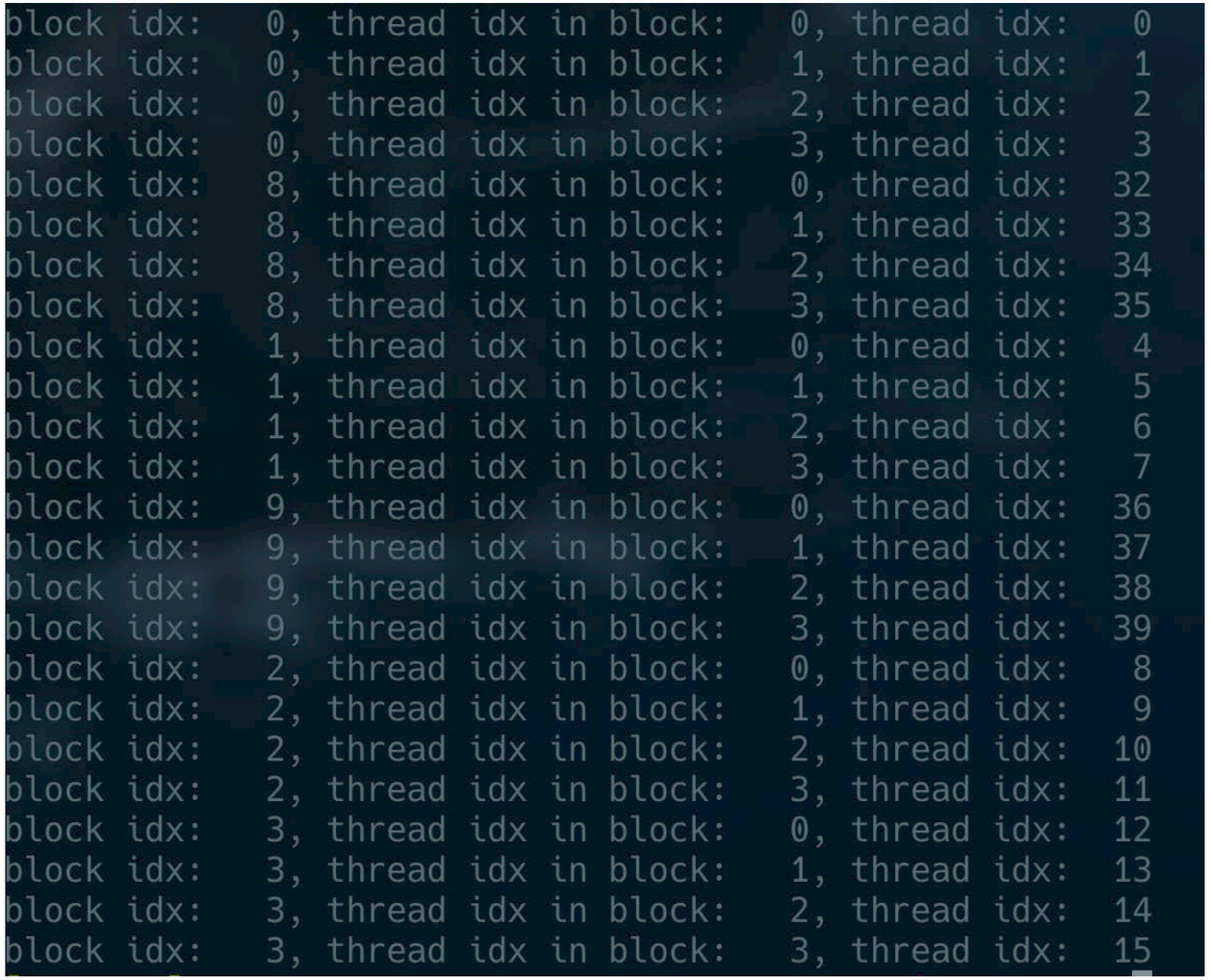
.cu和.cpp相互引用以及Makefile

Goal: 理解.cu和.cpp中相互引用时的注意事项，理解Makefile的写法

执行一下我们的第二个CUDA程序



2.2-cpp_cuda_interactive
(先不要管compile_commands.json，这个是neovim下进行c++函数定义声明调用跳转的东西。类似与.vscode)



执行trt-cuda的一部分结果
(这个是显示一个核函数的grid和block信息的一个程序)

执行一下我们的第二个CUDA程序

```cpp
void print_two_dim(int inputSize, int blockSize){
    int gridSize = inputSize / blockSize;

    dim3 block(blockSize, blockSize);
    dim3 grid(gridSize, gridSize);

    // print_idx_device(block, grid);
    // print_dim_device(block, grid);
    // print_thread_idx_per_block_device(block, grid);
    print_thread_idx_device(block, grid);
}
```

main.cpp

```cpp
void print_thread_idx_device(dim3 grid, dim3 block){
  print_thread_idx_kernel<<<grid, block>>>();
  cudaDeviceSynchronize();
}
```

print_index.cu

(给cpp文件提供了一个接口，用来间接执行cuda核函数)

```cpp
__global__ void print_thread_idx_kernel(){
  int bSize  = blockDim.z * blockDim.y * blockDim.x;

  int bIndex = blockIdx.z * gridDim.x * gridDim.y + \
               blockIdx.y * gridDim.x + \
               blockIdx.x;

  int tIndex = threadIdx.z * blockDim.x * blockDim.y + \
               threadIdx.y * blockDim.x + \
               threadIdx.x;

  int index  = bIndex * bSize + tIndex;

  printf("block idx: %3d, thread idx in block: %3d, thread idx: %3d\n",
         bIndex, tIndex, index);
}
```

print_index.cu

(具体的核函数的实现部分)

执行一下我们的第二个CUDA程序

```cpp
void print_two_dim(int inputSize, int blockSize){
    int gridSize = inputSize / blockSize;

    dim3 block(blockSize, blockSize);
    dim3 grid(gridSize, gridSize);

    // print_idx_device(block, grid);
    // print_dim_device(block, grid);
    // print_thread_idx_per_block_device(block, grid);
    print_thread_idx_device(block, grid);
}
```

main.cpp

```cpp
__global__ void print_thread_idx_kernel(){
  int bSize  = blockDim.z * blockDim.y * blockDim.x;

  int bIndex = blockIdx.z * gridDim.x * gridDim.y + \
               blockIdx.y * gridDim.x + \
               blockIdx.x;

  int tIndex = threadIdx.z * blockDim.x * blockDim.y + \
               threadIdx.y * blockDim.x + \
               threadIdx.x;

  int index  = bIndex * bSize + tIndex;

  printf("block idx: %3d, thread idx in block: %3d, thread idx: %3d\n",
         bIndex, tIndex, index);
}
```

print_index.cu
(具体的核函数的实现部分)

```
src/print_index.cpp:59:42: error: expected primary-expression before '>' token
   59 |    print_thread_idx_kernel<<<grid, block>>>();
      |                                          ^
```

我们不能够从main.cpp中直接调用cuda语法<<<>>>，<<<>>>是nvcc(cuda编译器)才可以识别的语法，g++不可以

执行一下我们的第二个CUDA程序

```
#ifndef __PRINT_INDEX_HPP
#define __PRINT_INDEX_HPP

#include <cuda_runtime.h>
void print_idx_device(dim3 grid, dim3 block);
void print_dim_device(dim3 grid, dim3 block);
void print_thread_idx_per_block_device(dim3 grid, dim3 block);
void print_thread_idx_device(dim3 grid, dim3 block);

#endif //__PRINT_INDEX_HPP
```

print_index.hpp

在这个案例中，.cu中提供给.cpp的接口函数定义在了.hpp中，这样可以在make的时候通过头文件找到函数的声明
当然，我们也可以把.hpp中函数的声明直接写在.cpp中，这样可以把这个.hpp删掉，让目录结构更简单一点。有兴趣的同学可以试一下

执行一下我们的第二个CUDA程序

```
42 void print_idx_device(dim3 grid, dim3 block){
43     print_idx_kernel<<<grid, block>>>();
44     CUDA_CHECK(cudaDeviceSynchronize());
45 }
```

print_index.cu

有心的同学已经注意到，这里有一个cudaDeviceSynchronize()
synchronize的中文是“同步”
这个语句的意思就是host和device的同步。也就是cpu和gpu执行的同步
因为kernel函数的执行默认是异步的。所以需要cudaDeviceSynchroize()来强制性的让kernel函数的结果执行结束之后host再执行下一步。

CUDA_CHECK是一种CUDA程序的排查错误的手段。error handler
有关错误排查后面仔细会讲。

Makefile添加的部分

```
CXX_SRC       +=  $(wildcard $(SRC_PATH)/*.cpp)
KERNELS_SRC   :=  $(wildcard $(SRC_PATH)/*.cu)

APP_OBJS      :=  $(patsubst $(SRC_PATH)%, $(BUILD_PATH)%, $(CXX_SRC:.cpp=.cpp.o))
APP_OBJS      +=  $(patsubst $(SRC_PATH)%, $(BUILD_PATH)%, $(KERNELS_SRC:.cu=.cu.o))

APP_MKS       :=  $(APP_OBJS:.o=.mk)

APP_DEPS      :=  $(CXX_SRC)
APP_DEPS      +=  $(KERNELS_SRC)
APP_DEPS      +=  $(wildcard $(SRC_PATH)/*.h)
```

Makefile

添加了一些对.cpp文件的变量

```
74 # Compile CXX
75 $(BUILD_PATH)/%.cpp.o: $(SRC_PATH)/%.cpp
76     @echo Compile CXX $@
77     @mkdir -p $(BUILD_PATH)
78     @$(CC) -o $@ -c $< $(CXXFLAGS) $(INCS)
```

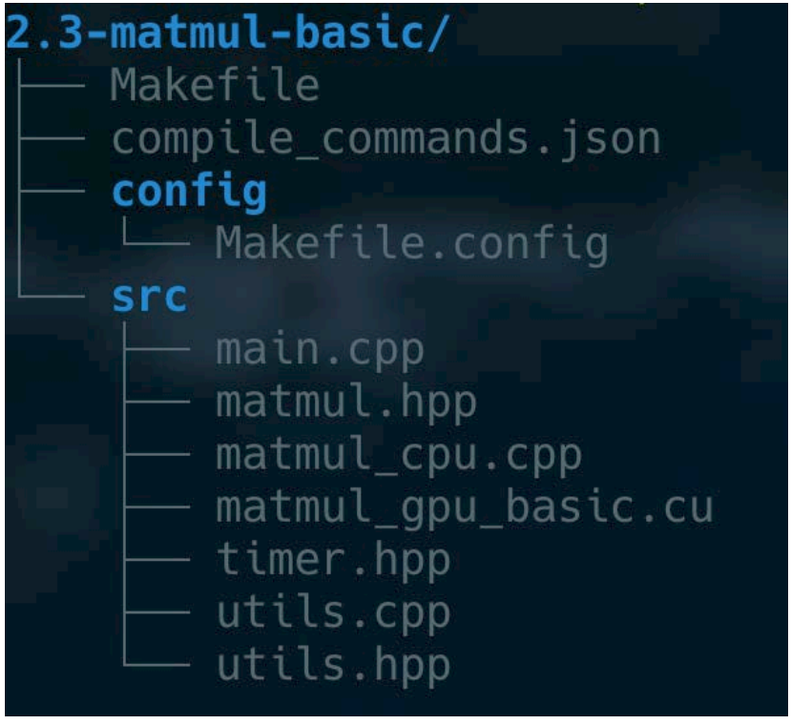
Makefile

添加了一些对.cpp文件的编译

初步计算matmul

Goal: 理解使用cuda进行矩阵乘法的加速方法，tile的用意

执行一下我们的第三个CUDA程序



2.3-matmul-basic
(先不要管compile_commands.json，这个是neovim下进行c++函数定义声明调用跳转的东西。类似与.vscode)

make之后执行trt-cuda的一部分结果
(这个是显示对于矩阵乘法的cpu和gpu的运行时间)

执行一下我们的第三个CUDA程序

```cpp
Timer timer;
int width     = 1<<10; // 1,024
int low       = 0;
int high      = 1;
int size      = width * width;
int blockSize = 16;

float* h_matM = (float*)malloc(size * sizeof(float));
float* h_matN = (float*)malloc(size * sizeof(float));
float* h_matP = (float*)malloc(size * sizeof(float));
float* d_matP = (float*)malloc(size * sizeof(float));

// seed = (unsigned)time(NULL);
seed = 1;
initMatrix(h_matM, size, low, high, seed);
seed += 1;
initMatrix(h_matN, size, low, high, seed);

/* CPU */
timer.start();
MatmulOnHost(h_matM, h_matN, h_matP, width);
timer.stop();
timer.duration<Timer::ms>("matmul in cpu");

/* GPU warmup */
timer.start();
MatmulOnDevice(h_matM, h_matN, d_matP, width, blockSize);
timer.stop();
timer.duration<Timer::ms>("matmul in gpu(warmup)");

/* GPU general implementation */
timer.start();
MatmulOnDevice(h_matM, h_matN, d_matP, width, blockSize);
timer.stop();
timer.duration<Timer::ms>("matmul in gpu(general)");

compareMat(h_matP, d_matP, size);
```

main.cpp
(创建了一个1024x1024大小的矩阵，初始化，计算，以及验证)

```cpp
/*
    CUDA中使用block对矩阵中某一片区域进行集中计算。这个类似于loop中的tile
    感兴趣的同学可以试着改一下blockSize，也就是tileSize，看看速度会发生什么样子的变化
    当blockSize达到一个数量的时候，这个程序会出错。下一个案例中我们会分析
*/
void MatmulOnDevice(float *M_host, float *N_host, float* P_host, int width, int blockSize){
    /* 设置矩阵大小 */
    int size = width * width * sizeof(float);

    /* 分配M，N在GPU上的空间*/
    float *M_device;
    float *N_device;
    cudaMalloc(&M_device, size);
    cudaMalloc(&N_device, size);

    /* 分配M，N拷贝到GPU上*/
    cudaMemcpy(M_device, M_host, size, cudaMemcpyHostToDevice);
    cudaMemcpy(N_device, N_host, size, cudaMemcpyHostToDevice);

    /* 分配P在GPU上的空间*/
    float *P_device;
    cudaMalloc(&P_device, size);

    /* 调用kernel来进行matmul计算，在这个例子中我们用的方案是：将一个矩阵切分成多个blockSize * blockSize的大小 */
    dim3 dimBlock(blockSize, blockSize);
    dim3 dimGrid(width / blockSize, width / blockSize);
    MatmulKernel <<<dimGrid, dimBlock>>> (M_device, N_device, P_device, width);

    /* 将结果从device拷贝回host*/
    cudaMemcpy(P_host, P_device, size, cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();

    /* Free */
    cudaFree(P_device);
    cudaFree(N_device);
    cudaFree(M_device);
}
```

matmul_gpu_basic.cu
(接口的实现。分配grid, block的大小，分配device上的空间)

```cpp
/* matmul的函数实现*/
__global__ void MatmulKernel(float *M_device, float *N_device, float *P_device, int width){
    /*
        我们设定每一个thread负责P中的一个坐标的matmul
        所以一共有width * width个thread并行处理P的计算
    */
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    float P_element = 0;

    /* 对于每一个P的元素，我们只需要循环遍历width次M和N中的元素就可以了*/
    for (int k = 0; k < width; k ++){
        float M_element = M_device[y * width + k];
        float N_element = N_device[k * width + x];
        P_element += M_element * N_element;
    }

    P_device[y * width + x] = P_element;
}
```

matmul_gpu_basic.cu
(核函数的实现)

执行一下我们的第三个CUDA程序

新添加的东西

```
# Compile CXX
$(BUILD_PATH)/%.cpp.o: $(SRC_PATH)/%.cpp
  @echo Compile CXX $@
  @mkdir -p $(BUILD_PATH)
  @$(CC) -o $@ -c $< $(CXXFLAGS) $(INCS)
$(BUILD_PATH)/%.cpp.mk: $(SRC_PATH)/%.cpp
  @echo Compile Dependence CXX $@
  @mkdir -p $(BUILD_PATH)
  @$(CC) -M $< -MF $@ -MT $(@:.cpp.mk=.cpp.o) $(CXXFLAGS) $(INCS)

# Compile CUDA
$(BUILD_PATH)/%.cu.o: $(SRC_PATH)/%.cu
  @echo Compile CUDA $@
  @mkdir -p $(BUILD_PATH)
  @$(CUCC) -o $@ -c $< $(CUDAFLAGS) $(INCS)
$(BUILD_PATH)/%.cu.mk: $(SRC_PATH)/%.cu
  @echo Compile Dependence CUDA $@
  @mkdir -p $(BUILD_PATH)
  @$(CUCC) -M $< -MF $@ -MT $(@:.cu.mk=.cu.o) $(CUDAFLAGS) $(INCS)
```

Makefile

(因为文件变多了，我们希望能够在编译的过程中建立依赖关系，这样当有文件修改了，make的时候只会重新编译修改了的文件，参考[1])

```
void initMatrix(float* data, int size, int low, int high, int seed) {
    srand(seed);
    for (int i = 0; i < size; i ++) {
        data[i] = float(rand()) * float(high - low) / RAND_MAX;
    }
}

void printMat(float* data, int size) {
    for (int i = 0; i < size; i ++) {
        printf("%.8lf", data[i]);
        if (i != size - 1) {
            printf(", ");
        } else {
            printf("\n");
        }
    }
}

void compareMat(float* h_data, float* d_data, int size) {
    double precision = 1.0E-4;
    bool error = false;
    /*
     * 这里注意，浮点数运算时CPU和GPU之间的计算结果是有误差的
     * 一般来说误差保持在1.0E-4之内是可以接受的
    */
    for (int i = 0; i < size; i ++) {
        if (abs(h_data[i] - d_data[i]) > precision) {
            error = true;
            printf("cpu: %.8lf, gpu: %.8lf\n", h_data[i], d_data[i]);
            break;
        }
    }
    if (error)
      printf("Matmul result is different\n");
    else
      printf("Matmul result is same, precision is 1.0E-4\n");
}
```

utils.cpp

(矩阵的初始化，比较，打印等等。如果想调试的话可以把范围改小一点)

[1]https://github.com/shouxieai/tensorRT_Pro/blob/main/Makefile

执行一下我们的第三个CUDA程序

新添加的东西

```cpp
class Timer {
public:
    using s  = std::ratio<1, 1>;
    using ms = std::ratio<1, 1000>;
    using us = std::ratio<1, 1000000>;
    using ns = std::ratio<1, 1000000000>;

public:
    Timer(){};

public:
    void start() {mStart = std::chrono::high_resolution_clock::now();}
    void stop()  {mStop  = std::chrono::high_resolution_clock::now();}

    template <typename span>
    void duration(std::string msg);

private:
    std::chrono::time_point<std::chrono::high_resolution_clock> mStart;
    std::chrono::time_point<std::chrono::high_resolution_clock> mStop;
};

/*
 * 注意：这个实现是不能够非常精准的获取kernel函数的执行时间
 * 要如果想要精准的获取kernel实现需要通过cuda event来进行测量，这个在后面的案例中会讲
 */
template <typename span>
void Timer::duration(std::string msg){
    std::string str;
    char fMsg[100];
    std::sprintf(fMsg, "%-30s", msg.c_str());

    if(std::is_same<span, s>::value) { str = " s"; }
    else if(std::is_same<span, ms>::value) { str = " ms"; }
    else if(std::is_same<span, us>::value) { str = " us"; }
    else if(std::is_same<span, ns>::value) { str = " ns"; }

    std::chrono::duration<double, span> time = mStop - mStart;
    std::cout << fMsg << " uses " << time.count() << str << std::endl;
}
```

timer.hpp

(一个计时器的实现，需要注意的是这个实现不能够十分精准的测量gpu的消耗时间。之后的案例会解释为什么)

host端与device端的数据传输

(*)这些以cuda*开头的api一般被称作*cuda runtime api*,以CU*开头的api被称作*cuda driver api*。*cuda runtime api对底层的操作做好的封装便于使用，包括*

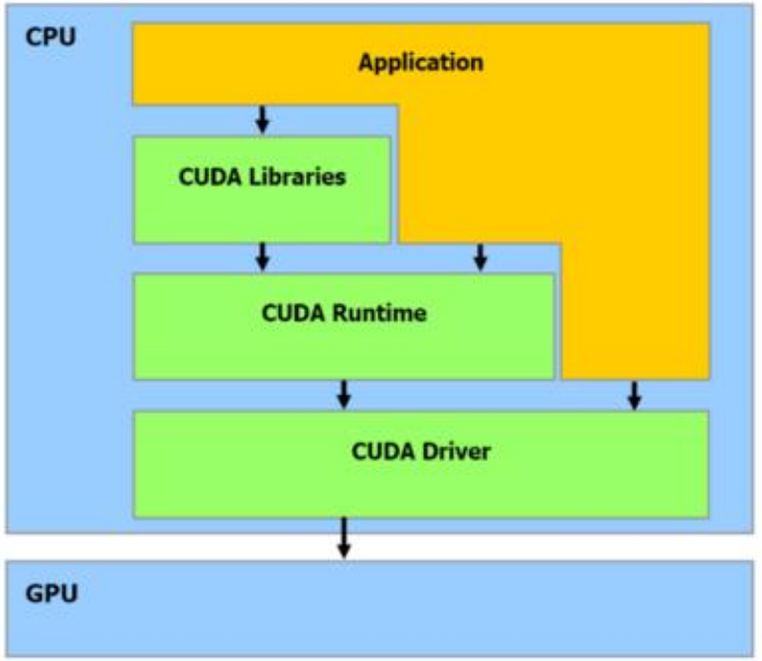
- *implicit initialization (隐式初始化)*
- *context management (上下文管理)*
- *module management (模块管理)*

CPU(host)端

- 分配host与device端的内存空间
- 将数据传送到GPU
- 配置核函数的参数
  - grid dim
  - block dim
  - shared memory size
  - stream
- 启动核函数
- 将数据从GPU传入回来

GPU(device)端

- 根据配置的参数启动核函数
- 多个thread并行计算



cuda各类api以及libraries的层级关系[1]

- cudaMalloc (在device端分配空间)，是一种cuda runtime api(*)
- cudaMallocHost (在host端的pinned memory上分配空间)

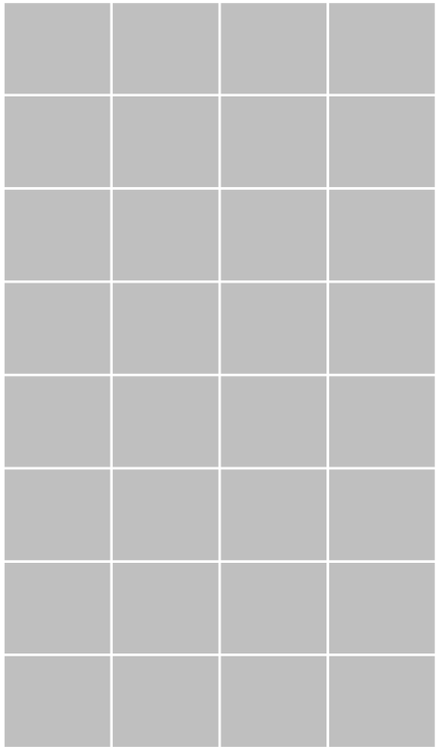
- cudaMemcpy (以同步的方式，将数据在host->device, device->device, device->host进行传输)
- cudaMemcpyAsync (以异步的方式，进行数据传输)

[1]CUDA software stack (NVIDIA, 2007)

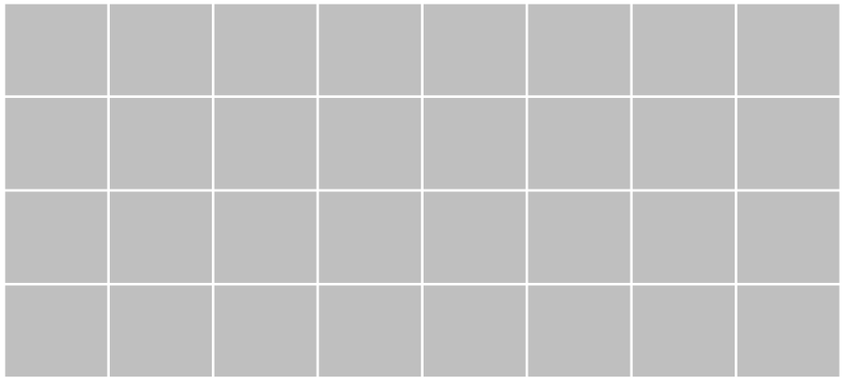
CUDA Core的矩阵乘法计算

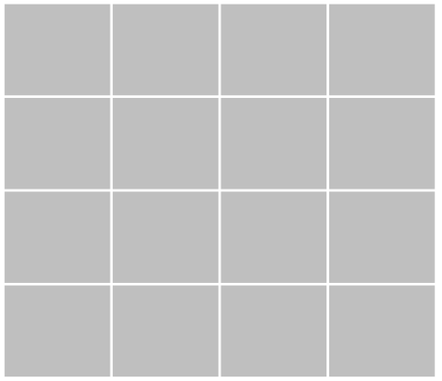
使用CUDA Core
计算 C = A * B

B(8 x 4)



A(4 x 8)



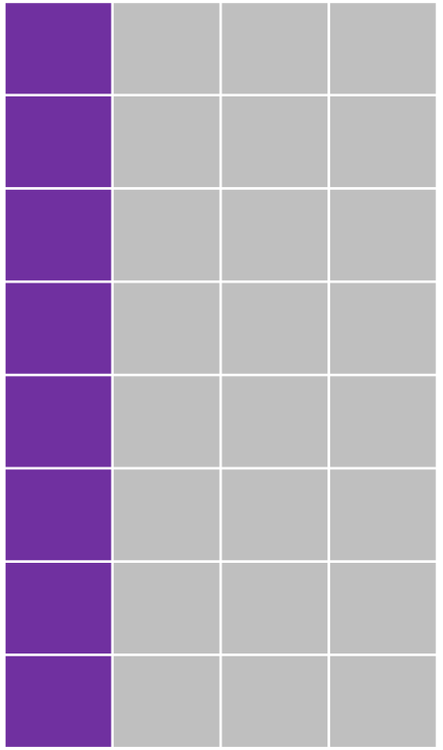


C(4 x 4)

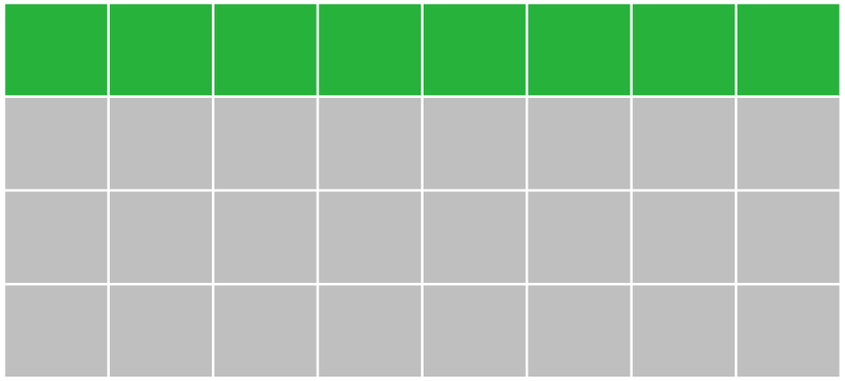
CUDA Core的矩阵乘法计算

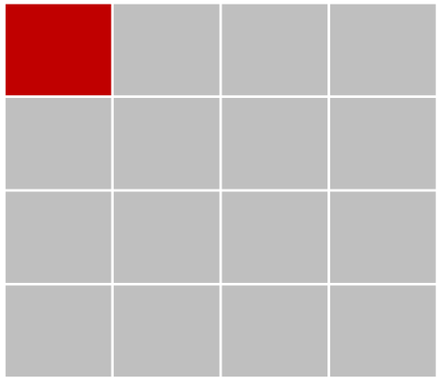
使用CUDA Core
计算 C = A * B

B(8 x 4)



A(4 x 8)





C(4 x 4)

(*)FMA: Fused Multiply-Add

CUDA Core的矩阵乘法计算

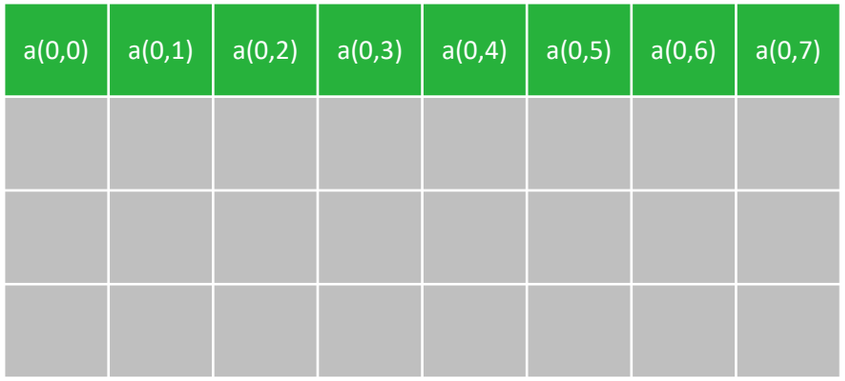
使用CUDA Core
计算 C = A * B

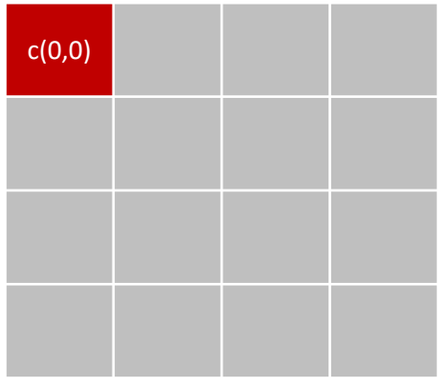
B(8 x 4)

$$
\begin{aligned}
c(0,0) &= a(0,0) * b(0,0) \\
&+ a(0,1) * b(1,0) \\
&+ a(0,2) * b(2,0) \\
&+ a(0,3) * b(3,0) \\
&+ a(0,4) * b(4,0) \\
&+ a(0,5) * b(5,0) \\
&+ a(0,6) * b(6,0) \\
&+ a(0,7) * b(7,0)
\end{aligned}
$$

需要8次FMA(*)，所以需要8个clk才可以完成一个c(0,0)

A(4 x 8)



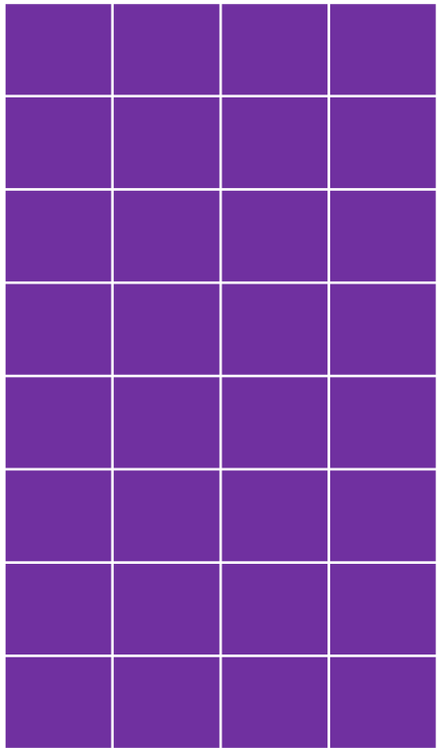


C(4 x 4)

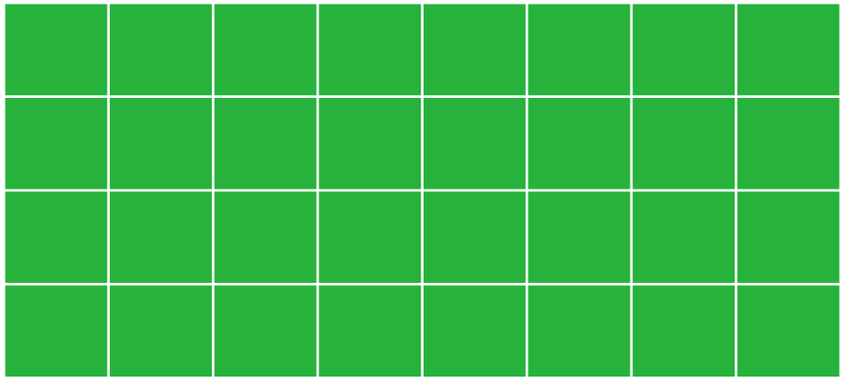
CUDA Core vs Tensor Core

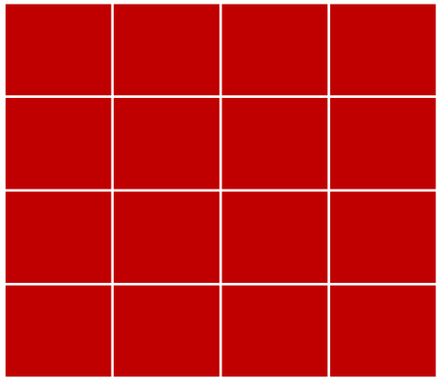
使用CUDA Core
计算 C = A * B

B(8 x 4)



A(4 x 8)





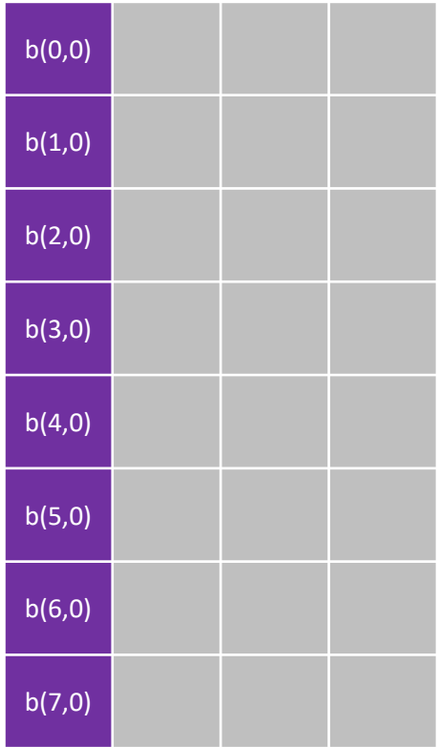
C(4 x 4)

要如果所有的thread依次执行的话，要完成4 x 8与8x 4 的计算，需要 8 * 16 = 128个clk才可以完成

CUDA Core的矩阵乘法计算

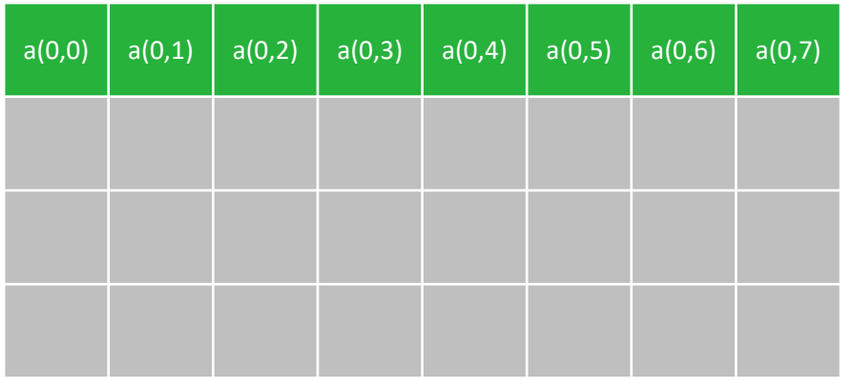
使用CUDA Core
计算 C = A * B

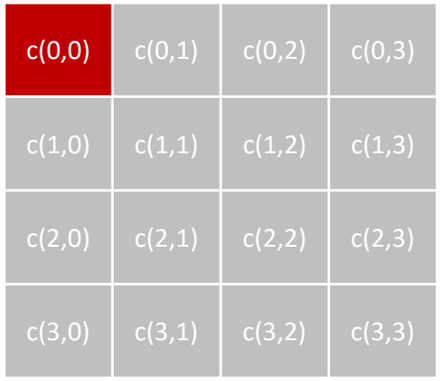
B(8 x 4)



$$
\begin{aligned}
c(0,0) &= a(0,0) * b(0,0) \\
&+ a(0,1) * b(1,0) \\
&+ a(0,2) * b(2,0) \\
&+ a(0,3) * b(3,0) \\
&+ a(0,4) * b(4,0) \\
&+ a(0,5) * b(5,0) \\
&+ a(0,6) * b(6,0) \\
&+ a(0,7) * b(7,0)
\end{aligned}
$$

A(4 x 8)





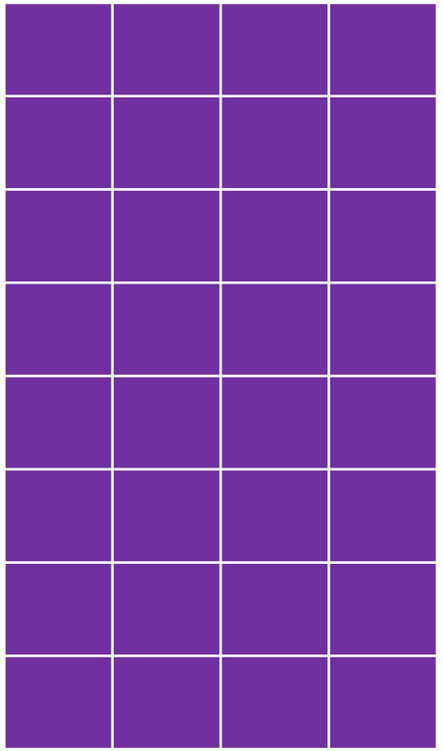
C(4 x 4)

要如果我们分配16个thread，每一个thread负责c中的一个元素...

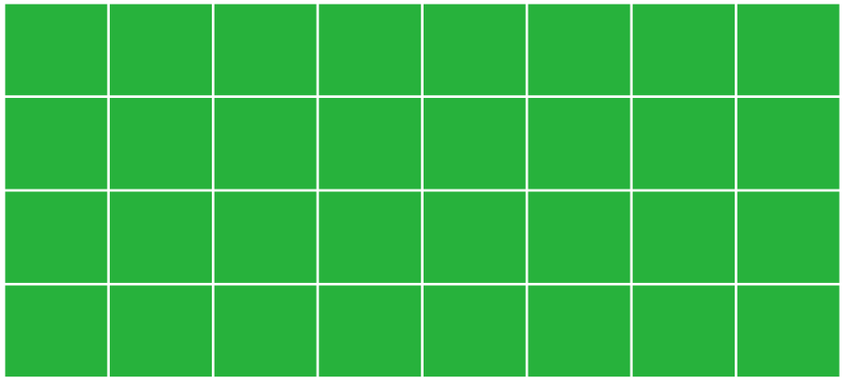
CUDA Core的矩阵乘法计算

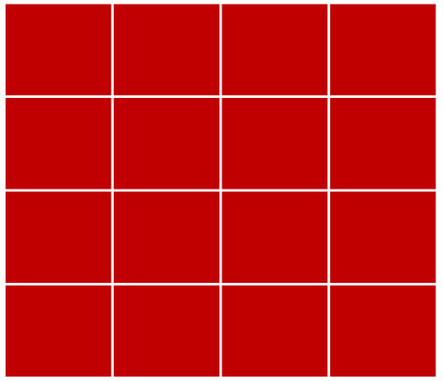
使用CUDA Core
计算 C = A * B

B(8 x 4)



A(4 x 8)





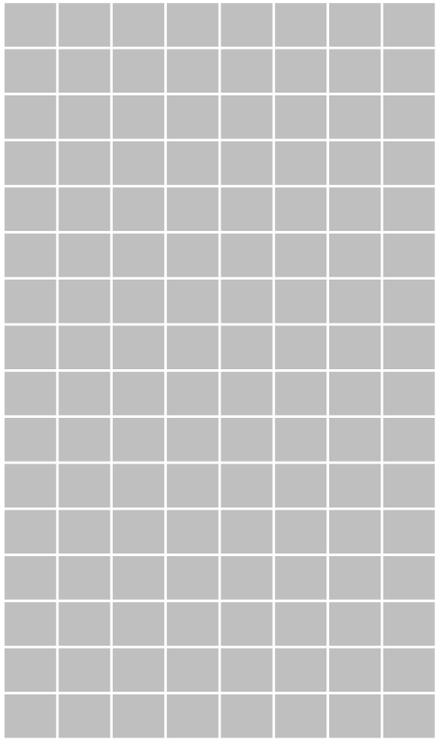
C(4 x 4)

所有的thread一起同时执行的话，要完成4 x 8与 8x 4 的计算，和一个1x8与8x1的计算所需要的时间是一样的。8个clk就可以完成了

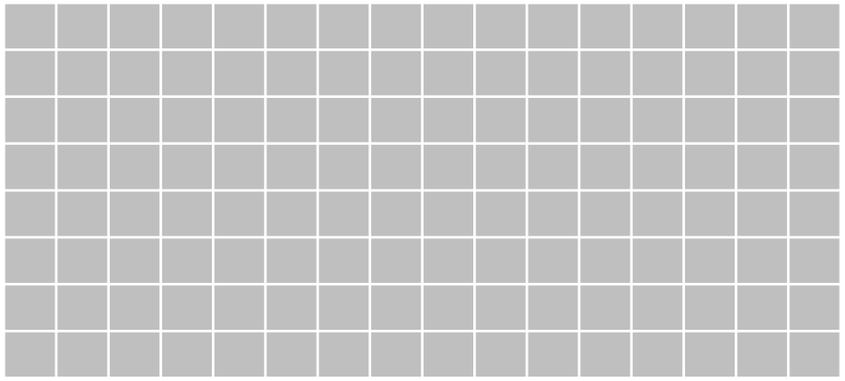
CUDA Core的矩阵乘法计算

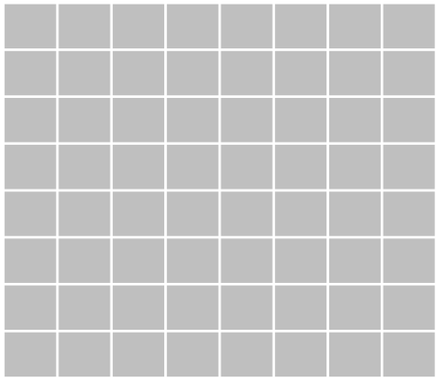
对于8x8的c，我们可以尝试这么切分：
blockDim(4, 4)
gridDim(2, 2)

B(16 x 8)



A(8 x 16)



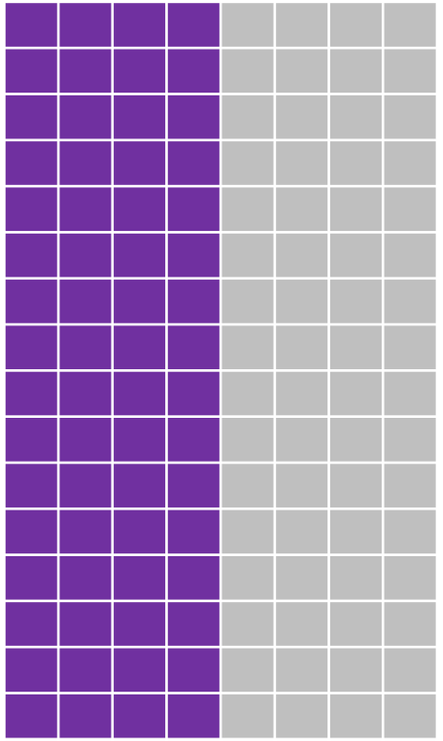


C(8 x 8)

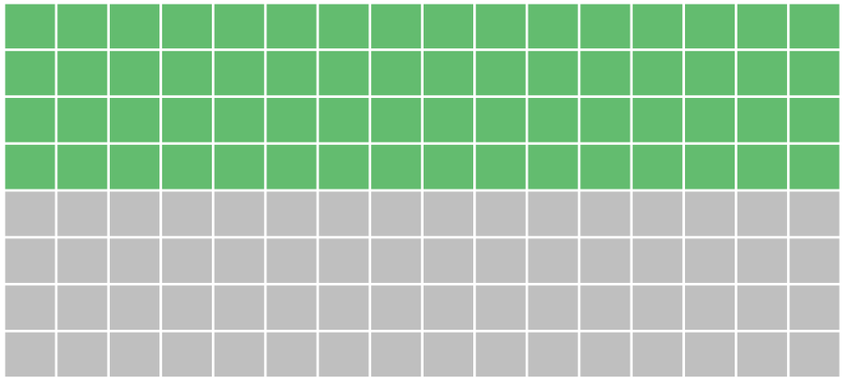
CUDA Core的矩阵乘法计算

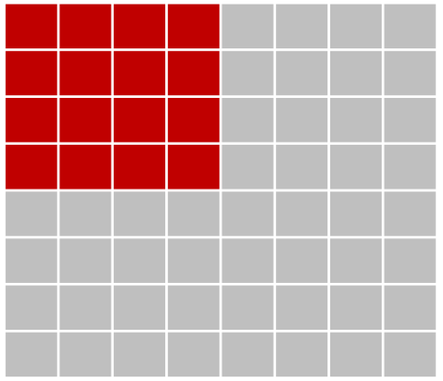
对于8x8的c，我们可以尝试这么切分：
blockDim(4, 4)
gridDim(2, 2)

B(16 x 8)



A(8 x 16)



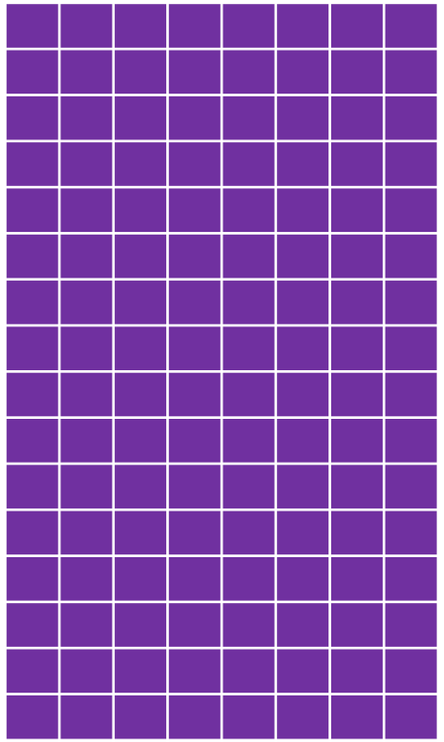


C(8 x 8)

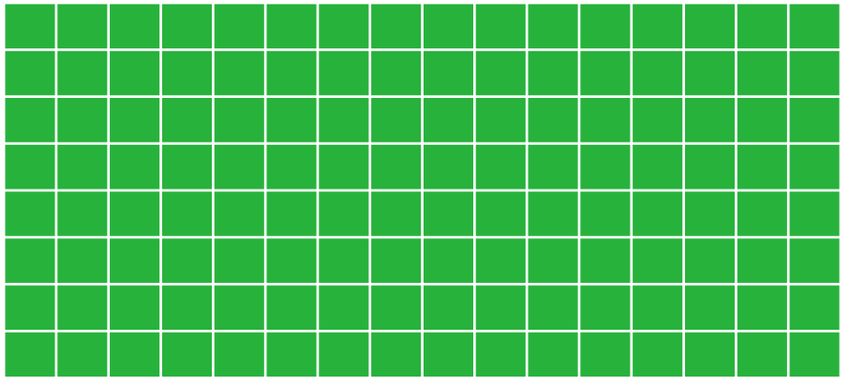
CUDA Core的矩阵乘法计算

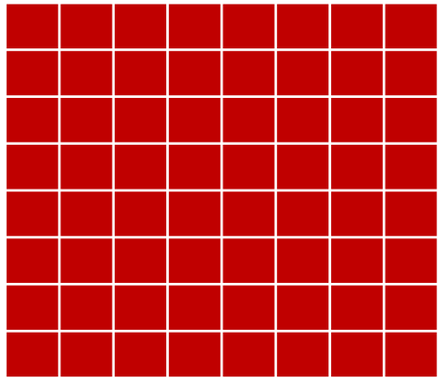
对于8x8的c，我们可以尝试这么切分：
blockDim(4, 4)
gridDim(2, 2)

B(16 x 8)



A(8 x 16)



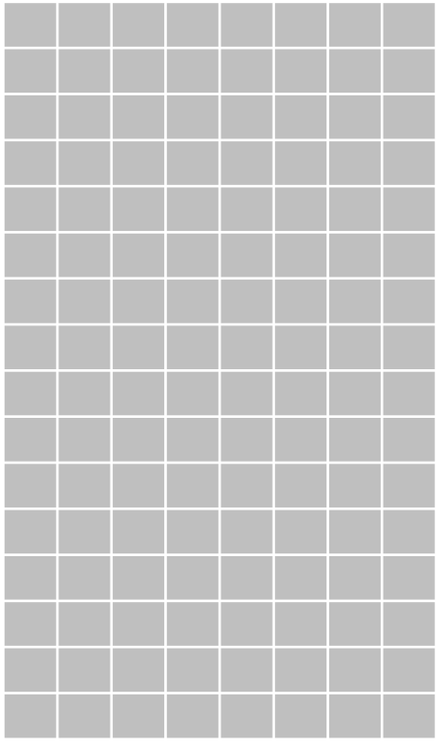


C(8 x 8)

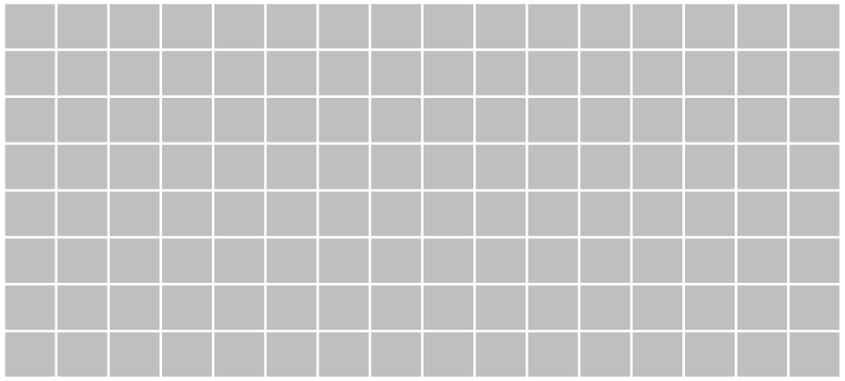
CUDA Core的矩阵乘法计算

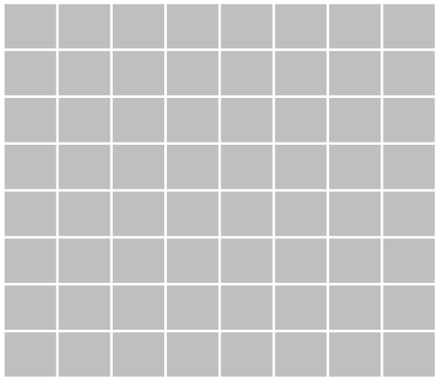
对于8x8的c，我们也可以尝试这么切分：
blockDim(8, 8)
gridDim(1, 1)

B(16 x 8)



A(8 x 16)





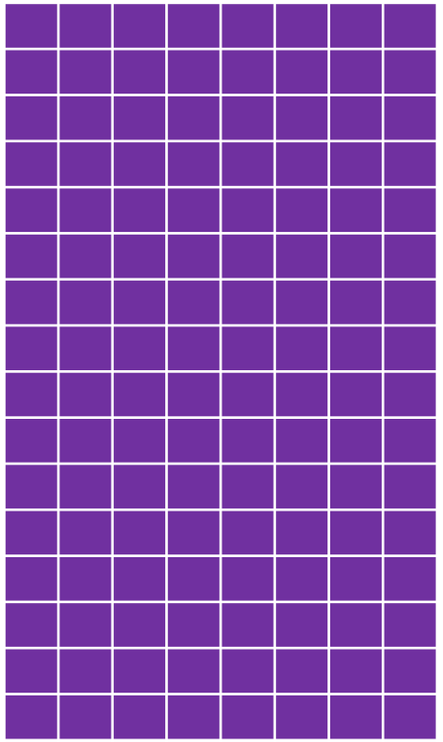
C(8 x 8)

CUDA Core的矩阵乘法计算

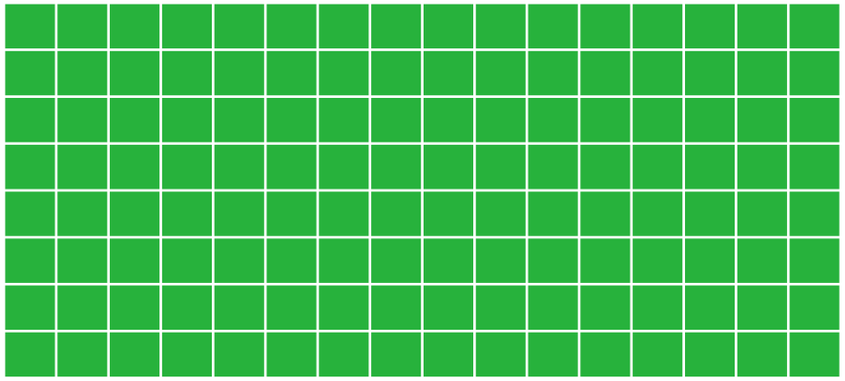
对于8x8的c，我们也可以尝试这么切分：
blockDim(8, 8)
gridDim(1, 1)

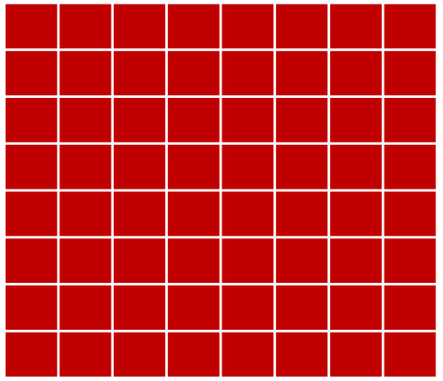
CUDA中有个规定，就是一个block中可以分配的thread的数量最大是1,024个线程。如果大于1,024会显示配置错误

B(16 x 8)



A(8 x 16)



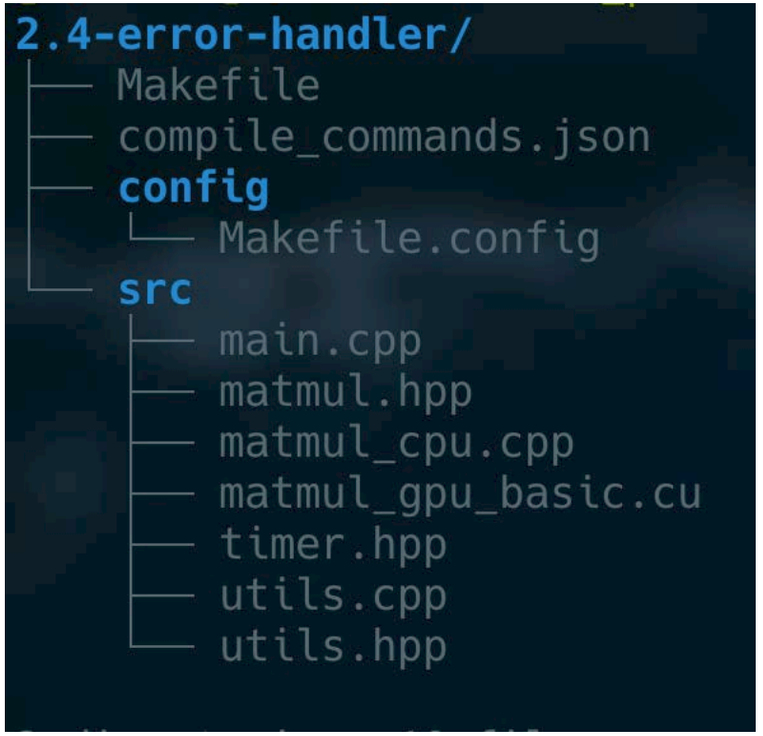


C(8 x 8)

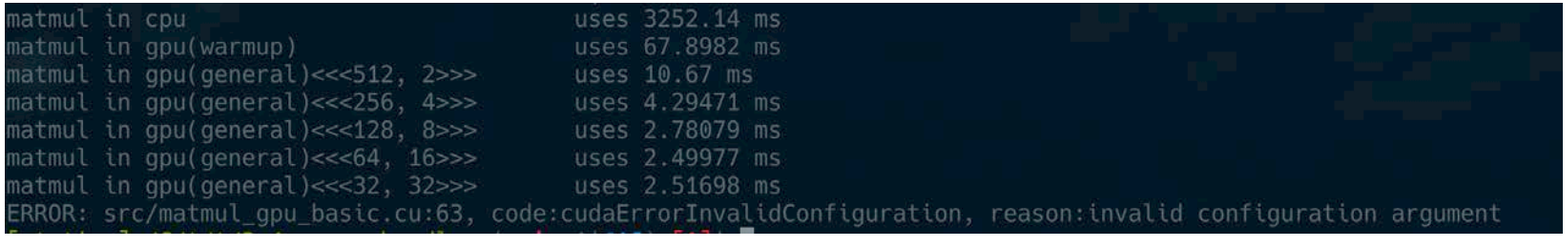
CUDA中的error handler

Goal: 养成使用cuda的error handler进行良好的编程习惯

执行一下我们的第四个CUDA程序



2.4-error-handler
(先不要管compile_commands.json，这个是neovim下进行c++函数定义声明调用跳转的东西。类似与.vscode)



make之后执行trt-cuda的一部分结果
(这里显示了当cuda配置错误的时候显示的错误提示信息)

执行一下我们的第四个CUDA程序

新添加的东西

```cpp
#define CUDA_CHECK(call)            __cudaCheck(call, __FILE__, __LINE__)
#define LAST_KERNEL_CHECK(call)     __kernelCheck(__FILE__, __LINE__)
#define BLOCKSIZE 16

static void __cudaCheck(cudaError_t err, const char* file, const int line) {
    if (err != cudaSuccess) {
        printf("ERROR: %s:%d, ", file, line);
        printf("code:%s, reason:%s\n", cudaGetErrorName(err), cudaGetErrorString(err));
        exit(1);
    }
}

static void __kernelCheck(const char* file, const int line) {
    /*
     * 在编写CUDA是，错误排查非常重要，默认的cuda runtime API中的函数都会返回cudaError_t类型的结果，
     * 但是在写kernel函数的时候，需要通过cudaPeekAtLastError或者cudaGetLastError来获取错误
     */
    cudaError_t err = cudaPeekAtLastError();
    if (err != cudaSuccess) {
        printf("ERROR: %s:%d, ", file, line);
        printf("code:%s, reason:%s\n", cudaGetErrorName(err), cudaGetErrorString(err));
        exit(1);
    }
}
```

utils.hpp
(这里添加了两个宏定义，一个是对cuda runtime api使用的error handling，一个是对最近的核函数的error handling)

```cpp
/* 分配M，N在GPU上的空间*/
float *M_device;
float *N_device;
CUDA_CHECK(cudaMalloc(&M_device, size));
CUDA_CHECK(cudaMalloc(&N_device, size));

/* 分配M，N拷贝到GPU上*/
CUDA_CHECK(cudaMemcpy(M_device, M_host, size, cudaMemcpyHos
CUDA_CHECK(cudaMemcpy(N_device, N_host, size, cudaMemcpyHos

/* 分配P在GPU上的空间*/
float *P_device;
CUDA_CHECK(cudaMalloc(&P_device, size));

/* 调用kernel来进行matmul计算，在这个例子中我们用的方案是：
dim3 dimBlock(blockSize, blockSize);
dim3 dimGrid(width / blockSize, width / blockSize);
MatmulKernel <<<dimGrid, dimBlock>>> (M_device, N_device, P

/* 将结果从device拷贝回host*/
CUDA_CHECK(cudaMemcpy(P_host, P_device, size, cudaMemcpyDev
CUDA_CHECK(cudaDeviceSynchronize());

/* 注意要在synchronization结束之后排查kernel的错误 */
LAST_KERNEL_CHECK();
```

matmul_gpu_basic.cu
(对于每一个cuda runtime api的时候都进行了错误检查，以及kernel执行结束以后对kernel进行错误排查)

为什么需要有error handler

```
blockSize = 64;
MatmulOnDevice(h_matM, h_matN, d_matP, width, blockSize);
timer.stop();
std::sprintf(str, "matmul in gpu(general)<<<%d, %d>>>", width / blockSize, blockSize);
timer.duration<Timer::ms>(str);
compareMat(h_matP, d_matP, size);
```

我们人为的设定了一个错误。block里面的thread数量是64*64 = 4,096

```
matmul in cpu                          uses 3329.74 ms
matmul in gpu(warmup)                  uses 58.8877 ms
matmul in gpu(general)<<<512, 2>>>     uses 10.4703 ms
matmul in gpu(general)<<<256, 4>>>     uses 4.10145 ms
matmul in gpu(general)<<<128, 8>>>     uses 2.63322 ms
matmul in gpu(general)<<<64, 16>>>     uses 2.35153 ms
matmul in gpu(general)<<<32, 32>>>     uses 2.39957 ms
matmul in gpu(general)<<<16, 64>>>     uses 1.17325 ms
Matmul result is different
cpu: 260.89050293, gpu: 0.00000000, cord:[0, 0]
```

没有error handling的时候

```
matmul in cpu                          uses 3208.02 ms
matmul in gpu(warmup)                  uses 59.477 ms
matmul in gpu(general)<<<512, 2>>>     uses 10.5102 ms
matmul in gpu(general)<<<256, 4>>>     uses 4.12098 ms
matmul in gpu(general)<<<128, 8>>>     uses 2.63845 ms
matmul in gpu(general)<<<64, 16>>>     uses 2.35508 ms
matmul in gpu(general)<<<32, 32>>>     uses 2.39826 ms
ERROR: src/matmul_gpu_basic.cu:63, code:cudaErrorInvalidConfiguration, reason:invalid configuration argument
```

有error handling的时候

为什么需要有error handler

一般来说，我们习惯把cuda的error handler定义成宏。因为这样可以避免在执行的时候发生调用error handler而引起的overhead。

- __FILE__: 编译器内部定义的一个宏。表示的是当前文件的文件名
- __LINE__: 编译器内部定义的一个宏。表示的是当前文件的行

一个良好的cuda编程习惯里，我们习惯在调用一个cuda runtime api时，我们就用error handler进行包装。这样可以方便我们排查错误的来源

```
#define CUDA_CHECK(call)              __cudaCheck(call, __FILE__, __LINE__)
#define LAST_KERNEL_CHECK(call)       __kernelCheck(__FILE__, __LINE__)
#define BLOCKSIZE 16

static void __cudaCheck(cudaError_t err, const char* file, const int line) {
    if (err != cudaSuccess) {
        printf("ERROR: %s:%d, ", file, line);
        printf("code:%s, reason:%s\n", cudaGetErrorName(err), cudaGetErrorString(err));
        exit(1);
    }
}

static void __kernelCheck(const char* file, const int line) {
    /*
     * 在编写CUDA是，错误排查非常重要，默认的cuda runtime API中的函数都会返回cudaError_t类型的结果，
     * 但是在写kernel函数的时候，需要通过cudaPeekAtLastError或者cudaGetLastError来获取错误
     */
    cudaError_t err = cudaPeekAtLastError();
    if (err != cudaSuccess) {
        printf("ERROR: %s:%d, ", file, line);
        printf("code:%s, reason:%s\n", cudaGetErrorName(err), cudaGetErrorString(err));
        exit(1);
    }
}
```

为什么需要有error handler

(*)“不可恢复的(non-recoverable/sticky)”，一般指的是核函数内部的执行错误，比较典型的例子就是内存访问越界。

相比之下，比如像block size以及shared memory size这种配置错误，是属于”可恢复的(recoverable/non-sticky)”错误

```
__host__ __device__ cudaError_t cudaGetLastError ( void )
```

Returns the last error from a runtime call.

Description

Returns the last error that has been produced by any of the runtime calls in the same instance of the CUDA Runtime library in the host thread and resets it to cudaSuccess.

```
__host__ __device__ cudaError_t cudaPeekAtLastError ( void )
```

Returns the last error from a runtime call.

Description

Returns the last error that has been produced by any of the runtime calls in the same instance of the CUDA Runtime library in the host thread. This call does not reset the error to cudaSuccess like cudaGetLastError().

NVIDIA官方对于两者的区别的解释[1]

两者差别在于错误是否传播。
对于不可恢复的错误(*)，如果发生了错误的话并且不把系统的状态进行reset的话，错误会一直传播下去，导致后面的即便正确的api使用也会产生同样的错误。

一般来说我们可以将错误分为：

- synchronous/asynchronous error
- sticky/non-sticky error

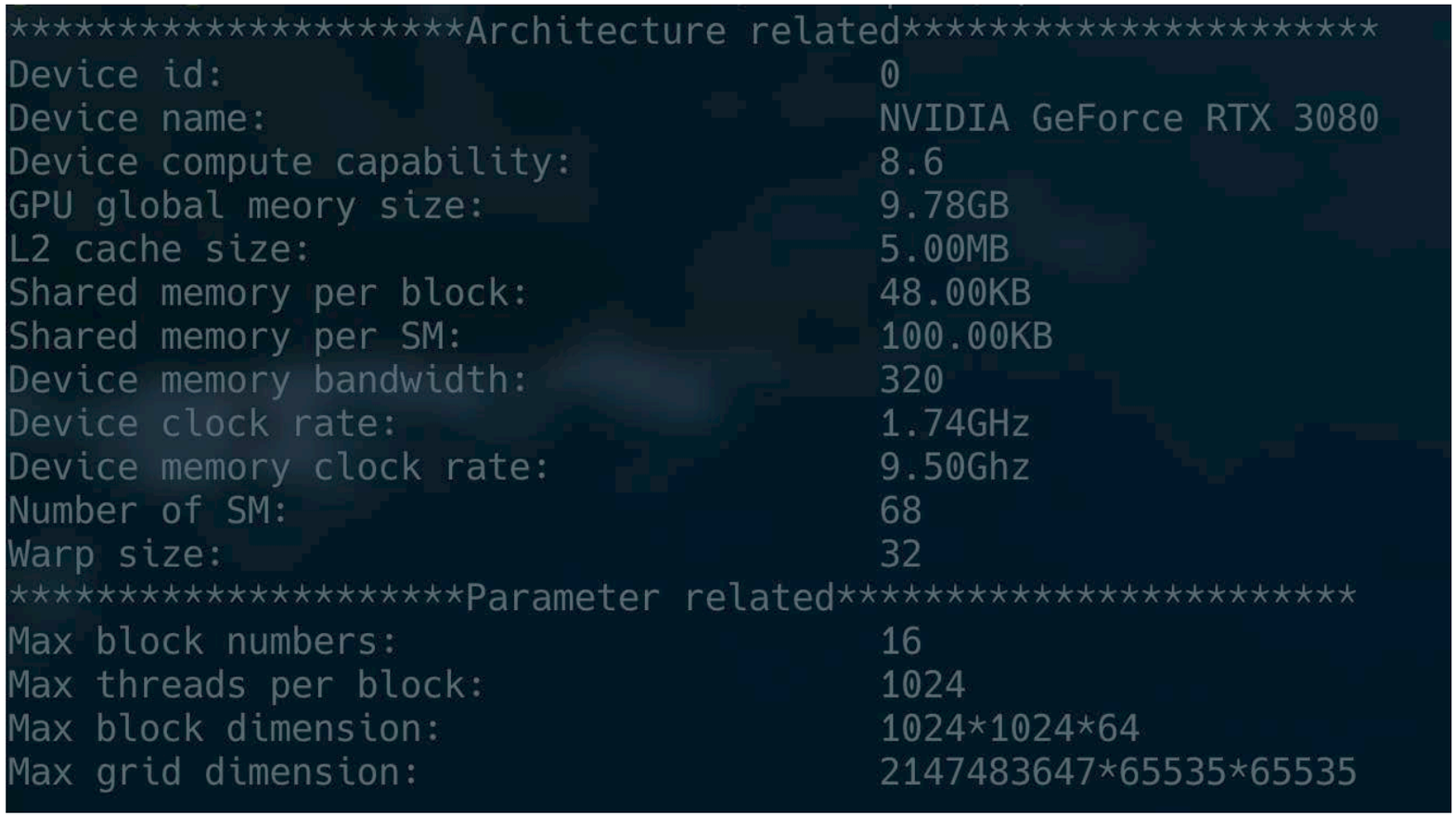
[1]NVIDIA, cuda toolkit documentation(cuda v12.2.0)

获取GPU的硬件信息

Goal: 学习如何使用cuda runtime api显示GPU硬件信息，
以及理解GPU硬件信息的重要性

执行一下我们的第五个CUDA程序

2.5-device-info
(先不要管compile_commands.json，这个是neovim下进行c++函数定义声明调用跳转的东西。类似与.vscode)



make之后执行trt-cuda的一部分结果
(这里显示了GPU当前的各种硬件信息)

执行一下我们的第五个CUDA程序

```cpp
int main(){
    int count;
    int index = 0;
    cudaGetDeviceCount(&count);
    while (index < count) {
        cudaSetDevice(index);
        cudaDeviceProp prop;
        cudaGetDeviceProperties(&prop, index);
        LOG("%-40s",              "********************Architecture related*********************");
        LOG("%-40s%d%s",          "Device id: ",                  index, "");
        LOG("%-40s%s%s",          "Device name: ",                prop.name, "");
        LOG("%-40s%.1f%s",        "Device compute capability: ",  prop.major + (float)prop.minor / 10, "");
        LOG("%-40s%.2f%s",        "GPU global meory size: ",      (float)prop.totalGlobalMem / (1<<30), "GB");
        LOG("%-40s%.2f%s",        "L2 cache size: ",              (float)prop.l2CacheSize / (1<<20), "MB");
        LOG("%-40s%.2f%s",        "Shared memory per block: ",    (float)prop.sharedMemPerBlock / (1<<10), "KB");
        LOG("%-40s%.2f%s",        "Shared memory per SM: ",       (float)prop.sharedMemPerMultiprocessor / (1<<10),
        LOG("%-40s%d%s",          "Device memory bandwidth: ",    prop.memoryBusWidth, "");
        LOG("%-40s%.2f%s",        "Device clock rate: ",          prop.clockRate*1E-6, "GHz");
        LOG("%-40s%.2f%s",        "Device memory clock rate: ",   prop.memoryClockRate*1E-6, "Ghz");
        LOG("%-40s%d%s",          "Number of SM: ",               prop.multiProcessorCount, "");
        LOG("%-40s%d%s",          "Warp size: ",                  prop.warpSize, "");

        LOG("%-40s",              "********************Parameter related***********************");
        LOG("%-40s%d%s",          "Max block numbers: ",          prop.maxBlocksPerMultiProcessor, "");
        LOG("%-40s%d%s",          "Max threads per block: ",      prop.maxThreadsPerBlock, "");
        LOG("%-40s%d*%d*%d%s",    "Max block dimension: ",        prop.maxThreadsDim[0], prop.maxThreadsDim[1], pro
        LOG("%-40s%d*%d*%d%s",    "Max grid dimension: ",         prop.maxGridSize[0], prop.maxGridSize[1], prop.ma
        index ++;
        printf("\n");
    }
    return 0;
```

main.cpp

(这里展示了一些比较重要的device info，比如说shared memory大小，
SM数量，thread per block的最大值，
还有很多其他的没有展示。建议大家去自行调查和使用一下)

执行一下我们的第五个CUDA程序

新添加的东西

```
#define LOG(...)                    __log_info(__VA_ARGS__)
```

```
31 static void __log_info(const char* format, ...) {
32     char msg[1000];
33     va_list args;
34     va_start(args, format);
35
36     vsnprintf(msg, sizeof(msg), format, args);
37
38     fprintf(stdout, "%s\n", msg);
39     va_end(args);
40 }
```

utils.hpp

(把打印日志的方法定义成了一个宏，方便使用)

__VA_ARGS__：编译器内部的定义的一个宏。表示的是变参。配合vsnprintf可以讲LOG中的变参信息存入到msg的这个buffer中。最终在一起打印出来

为什么要注意硬件信息

我们需要知道我们的GPU的compute capability。nvcc的参数中有时会需要

```
# GeForce RTX 3070, 3080, 3090
# ARCH= -gencode arch=compute_86,code=[sm_86,compute_86]

# Kepler GeForce GTX 770, GTX 760, GT 740
# ARCH= -gencode arch=compute_30,code=sm_30

# Tesla A100 (GA100), DGX-A100, RTX 3080
# ARCH= -gencode arch=compute_80,code=[sm_80,compute_80]

# Tesla V100
# ARCH= -gencode arch=compute_70,code=[sm_70,compute_70]

# GeForce RTX 2080 Ti, RTX 2080, RTX 2070, Quadro RTX 8000, Quadro RTX 6000, Quadro RTX 500
# ARCH= -gencode arch=compute_75,code=[sm_75,compute_75]

# Jetson XAVIER
# ARCH= -gencode arch=compute_72,code=[sm_72,compute_72]

# GTX 1080, GTX 1070, GTX 1060, GTX 1050, GTX 1030, Titan Xp, Tesla P40, Tesla P4
# ARCH= -gencode arch=compute_61,code=sm_61 -gencode arch=compute_61,code=compute_61

# GP100/Tesla P100 - DGX-1
# ARCH= -gencode arch=compute_60,code=sm_60

# For Jetson TX1, Tegra X1, DRIVE CX, DRIVE PX - uncomment:
# ARCH= -gencode arch=compute_53,code=[sm_53,compute_53]

# For Jetson Tx2 or Drive-PX2 uncomment:
# ARCH= -gencode arch=compute_62,code=[sm_62,compute_62]

# For Tesla GA10x cards, RTX 3090, RTX 3080, RTX 3070, RTX A6000, RTX A40 uncomment:
# ARCH= -gencode arch=compute_86,code=[sm_86,compute_86]
```

为什么要注意硬件信息

我们需要知道我们在启动核函数的时候，配置信息的规定都有哪些

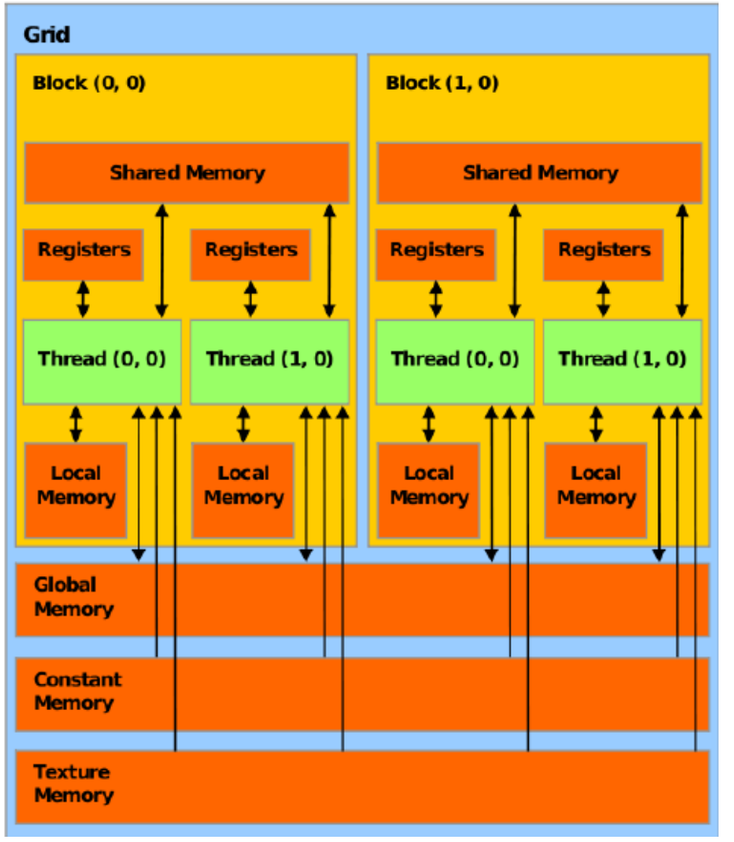
```
**********************Parameter related************************
Max block numbers:                              16
Max threads per block:                          1024
Max block dimension:                            1024*1024*64
Max grid dimension:                             2147483647*65535*65535
```

shared memory的使用对cuda程序的加速很重要。
当我们在使用shared memory是需要知道它的大小上限是多少。
这里需要注意的是，我们其实是可以调整shared memory和L1 cache在缓存中的空间。

```
Shared memory per block:                        48.00KB
Shared memory per SM:                           100.00KB
```

warp scheduler是cuda中对大量thread的一个调度器。
我们需要知道一个warp是由多少个thread组成的

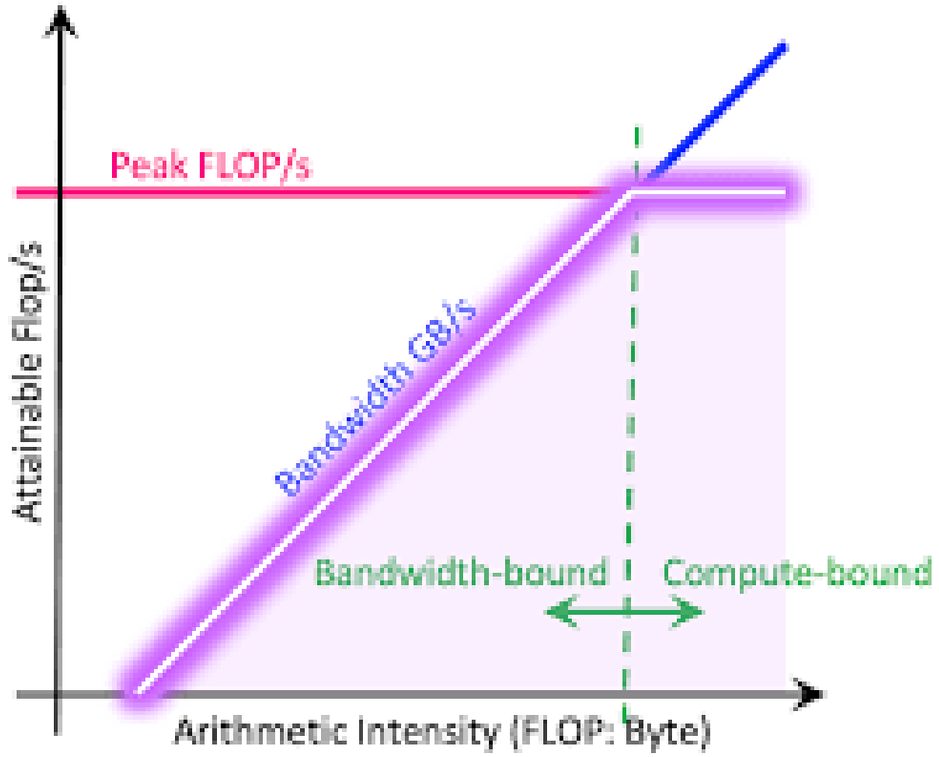
```
Warp size:                                      32
```



为什么要注意硬件信息

当我们在进行性能调优的时候，内存的大小和内存的带宽是我们需要考虑的一个很重要的因素。结合roofline model(后面的章节会讲)，我们需要寻找想要隐藏memory的数据传输所造成的overhead，需要多少的计算量和计算效率

```
Device memory bandwidth:                       320
Device clock rate:                             1.74GHz
Device memory clock rate:                      9.50Ghz
Number of SM:                                  68
```

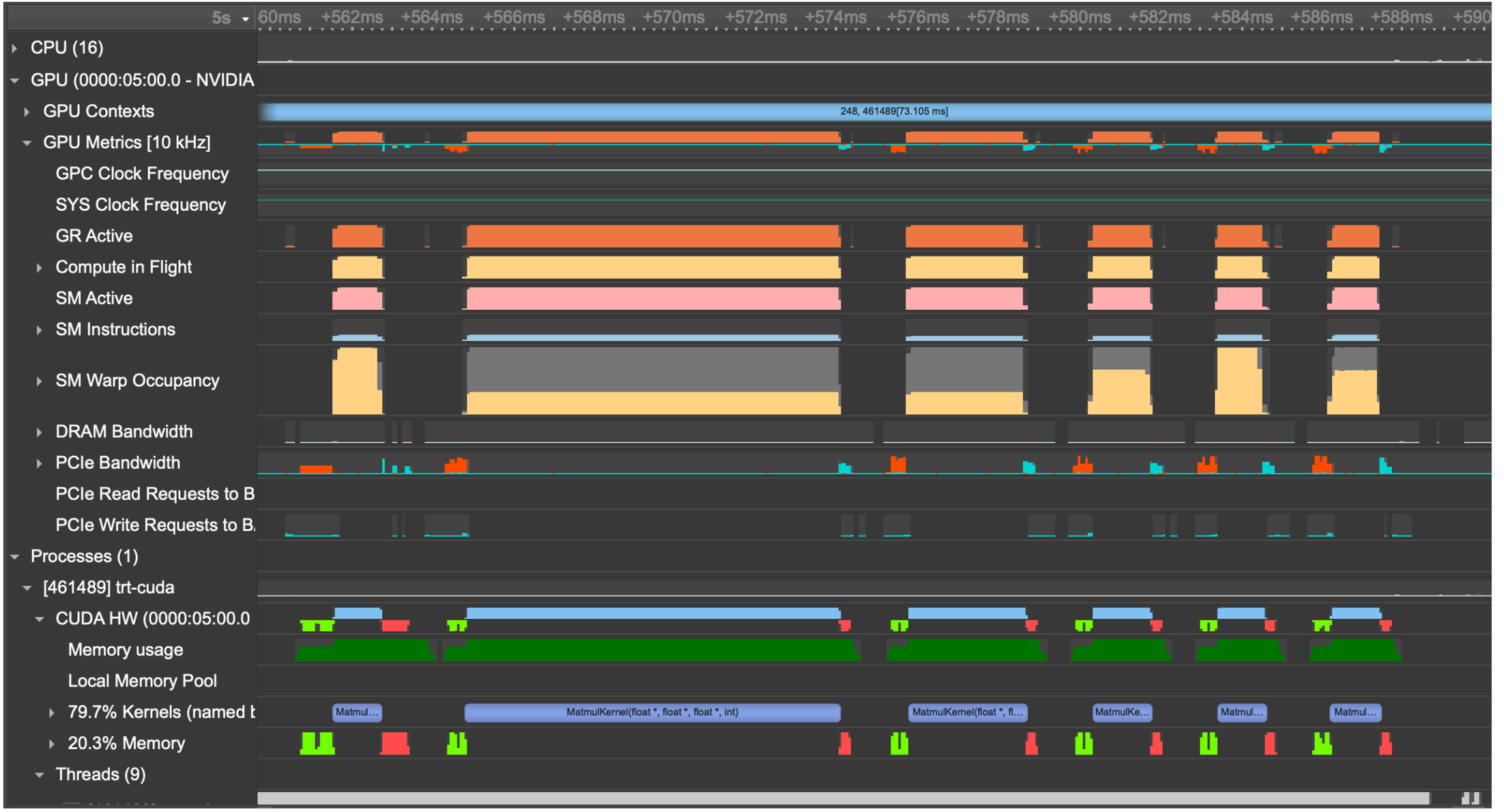


[1]Roofline Performance Model

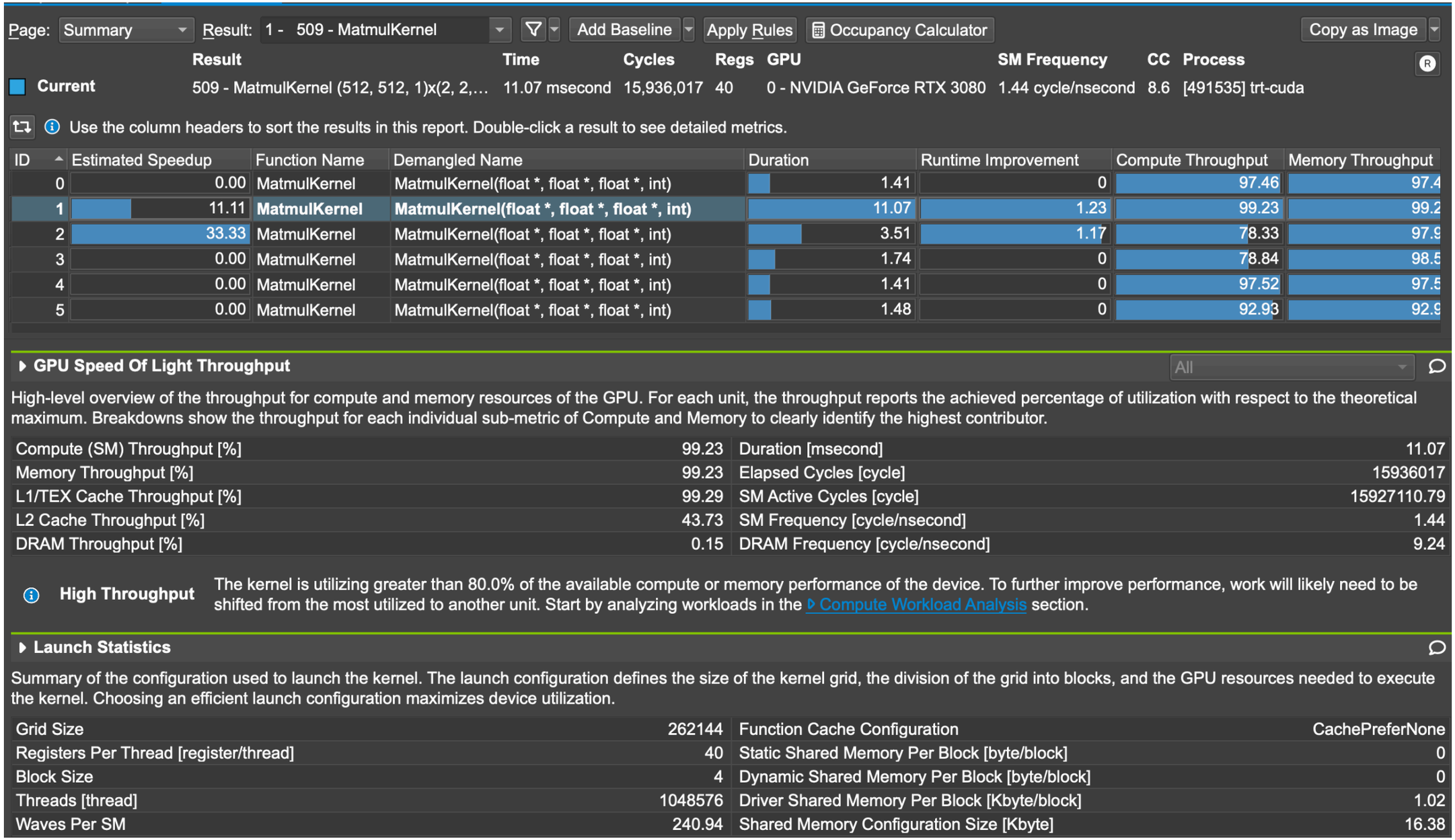
Nsight system and Nsight compute

Goal: 学习如何使用NVIDIA提供的Profiling tool进行性能分析

今天我们使用两个非常重要的软件(Nsight systems)



今天我们使用两个非常重要的软件(Nsight Compute)



Page: Summary | Result: 1 - 509 - MatmulKernel | Add Baseline | Apply Rules | Occupancy Calculator | Copy as Image

	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	509 - MatmulKernel (512, 512, 1)x(2, 2,…	11.07 msecond	15,936,017	40	0 - NVIDIA GeForce RTX 3080	1.44 cycle/nsecond	8.6	[491535] trt-cuda

Use the column headers to sort the results in this report. Double-click a result to see detailed metrics.

ID	Estimated Speedup	Function Name	Demangled Name	Duration	Runtime Improvement	Compute Throughput	Memory Throughput
0	0.00	MatmulKernel	MatmulKernel(float *, float *, float *, int)	1.41	0	97.46	97.4
1	11.11	**MatmulKernel**	**MatmulKernel(float *, float *, float *, int)**	11.07	1.23	99.23	99.2
2	33.33	MatmulKernel	MatmulKernel(float *, float *, float *, int)	3.51	1.17	78.33	97.9
3	0.00	MatmulKernel	MatmulKernel(float *, float *, float *, int)	1.74	0	78.84	98.5
4	0.00	MatmulKernel	MatmulKernel(float *, float *, float *, int)	1.41	0	97.52	97.5
5	0.00	MatmulKernel	MatmulKernel(float *, float *, float *, int)	1.48	0	92.93	92.9

GPU Speed Of Light Throughput

All

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor.

Compute (SM) Throughput [%]	99.23	Duration [msecond]	11.07
Memory Throughput [%]	99.23	Elapsed Cycles [cycle]	15936017
L1/TEX Cache Throughput [%]	99.29	SM Active Cycles [cycle]	15927110.79
L2 Cache Throughput [%]	43.73	SM Frequency [cycle/nsecond]	1.44
DRAM Throughput [%]	0.15	DRAM Frequency [cycle/nsecond]	9.24

High Throughput The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Compute Workload Analysis section.

Launch Statistics

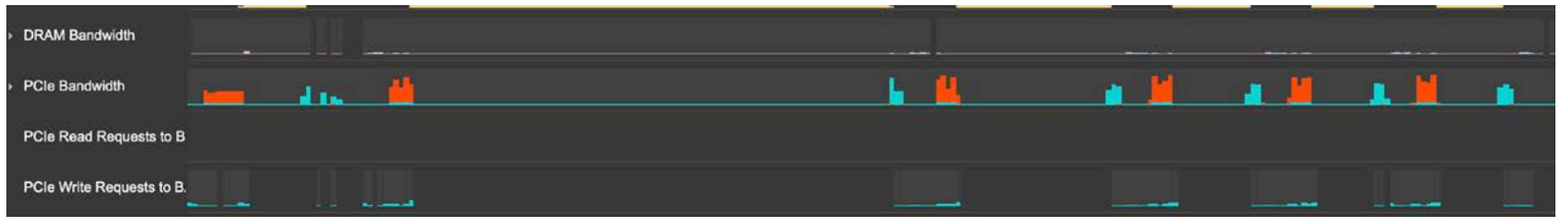
Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	262144	Function Cache Configuration	CachePreferNone
Registers Per Thread [register/thread]	40	Static Shared Memory Per Block [byte/block]	0
Block Size	4	Dynamic Shared Memory Per Block [byte/block]	0
Threads [thread]	1048576	Driver Shared Memory Per Block [Kbyte/block]	1.02
Waves Per SM	240.94	Shared Memory Configuration Size [Kbyte]	16.38

比较常用的分析

e.g. 对kernel执行和memory进行timeline分析，尝试寻找是否可以优化

- 隐藏memory access
- 多流调度
- 删除冗长的memory access
- 融合kernel减少kernel launch的overhead
- CPU与GPU的overlapping



e.g. 分析DRAM以及PCIe带宽的使用率

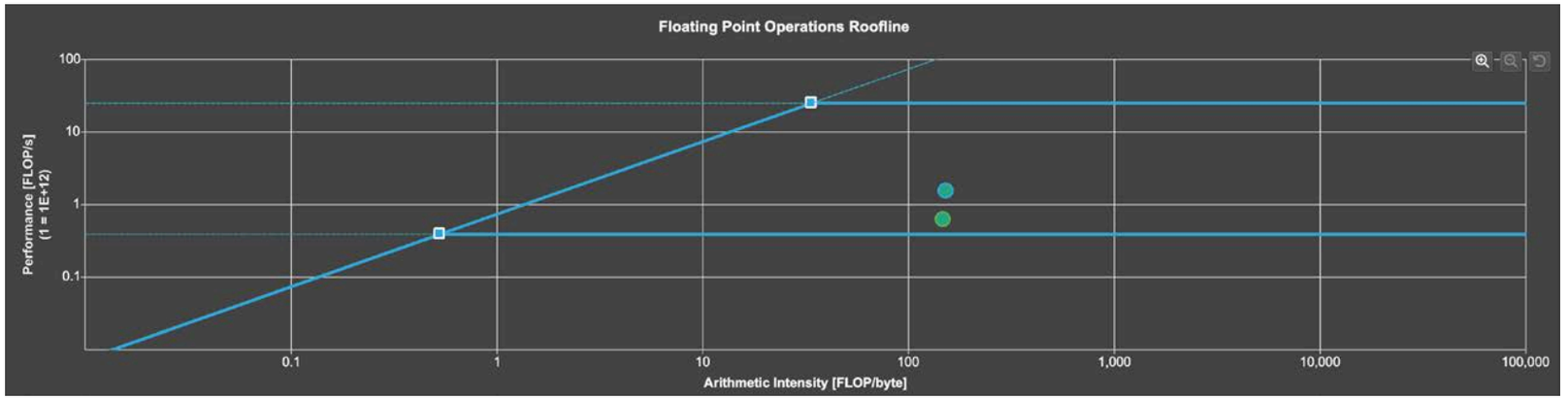
- 可以从中分析到哪些带宽没有被充分利用，从而进行优化

e.g. 分析SM中warp的占有率

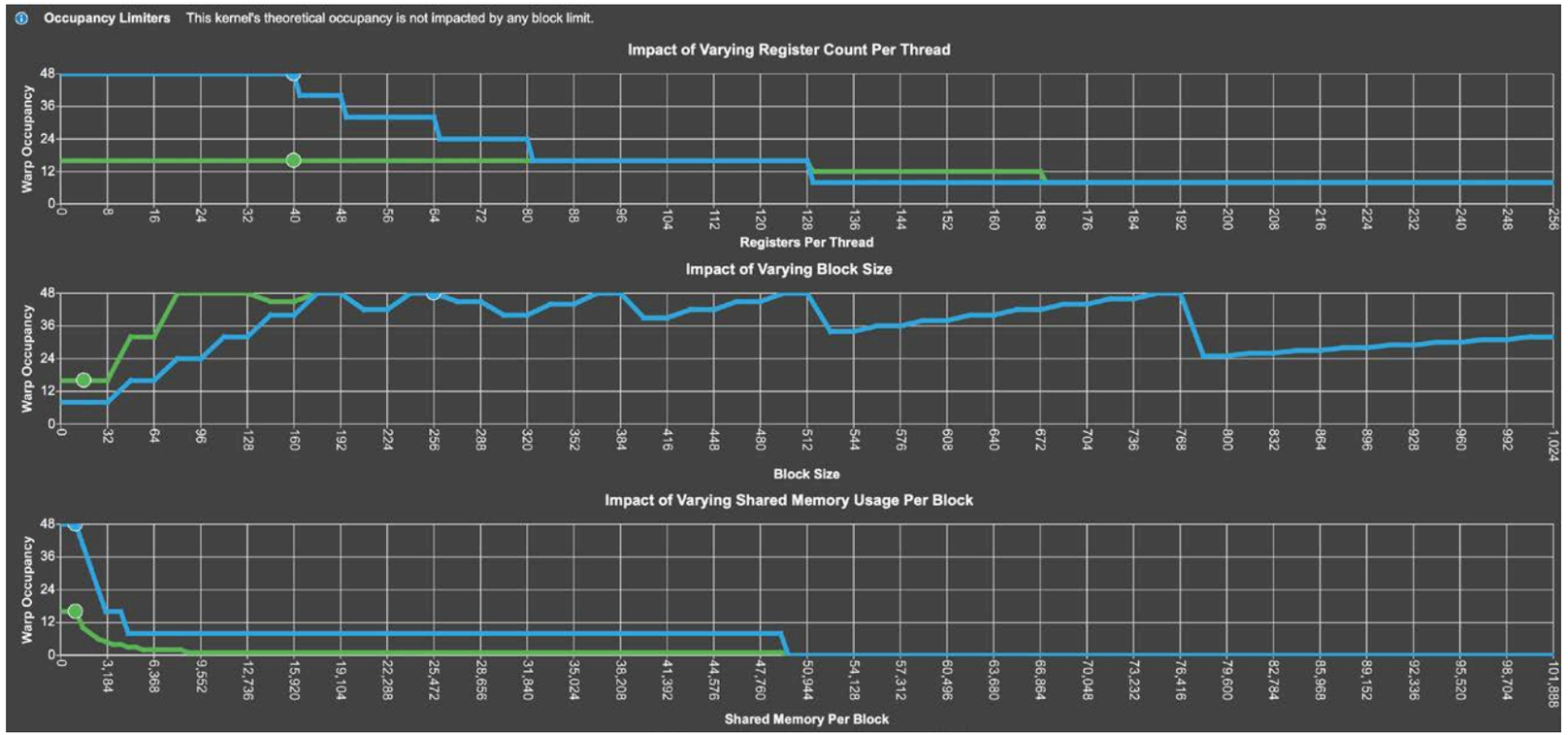
- 可以从中知道一个SM中资源是否被用满

比较常用的分析

baseline: matmulKernel<<<(256,256,1), (4,4,1)>>>
current: matmulKernel<<<(64, 64, 1), (16, 16, 1)>>>

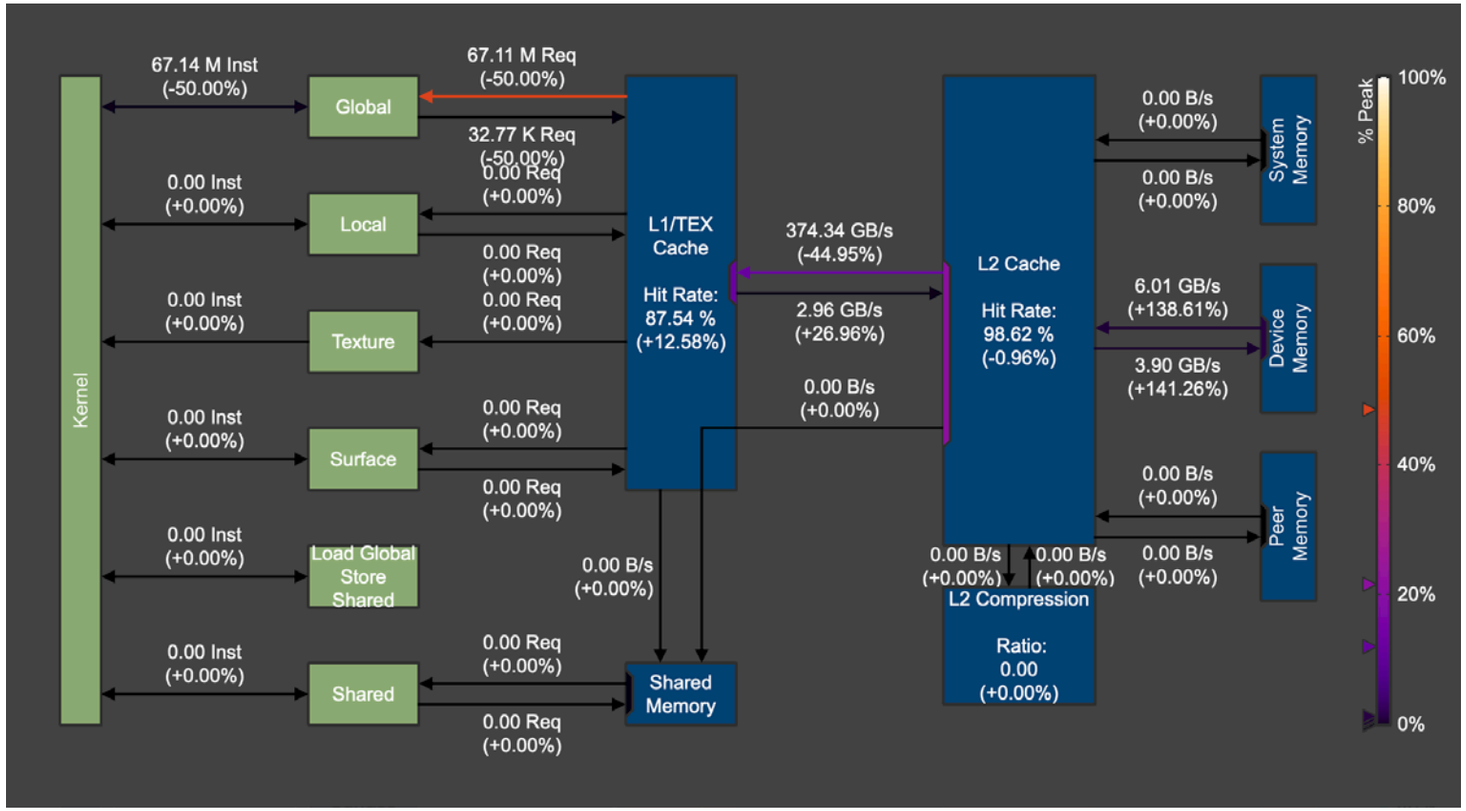


e.g. roofline analysis
对核函数进行roofline analysis，
并且根据base line进行优化比较



e.g. occupancy analysis
对核函数的各个指标进
行估算一个warp的占有
率的变化

比较常用的分析



e.g. memory bindwidth analysis
针对核函数中对各个memory的数据传输的带宽进行分析。可以比较好的理解memory架构

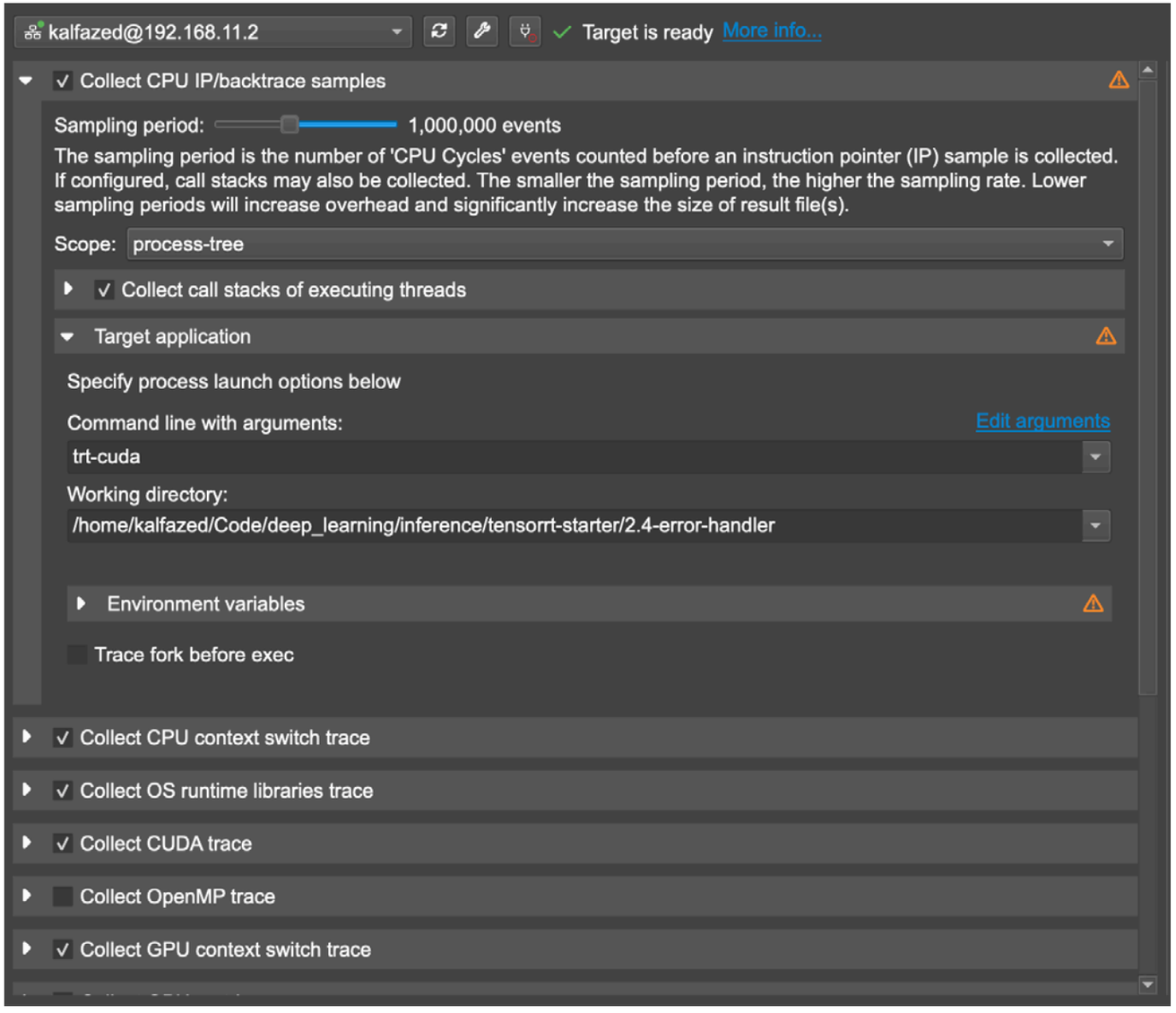
Shared Memory

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
Shared Load Matrix	0 (+0.00%)	0 (+0.00%)			
Shared Store	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
Shared Store From Global Load	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
Shared Atomic	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
Other	-	-	65536 (-50.00%)	0.05 (+24.50%)	0 (+0.00%)
Total	0 (+0.00%)	0 (+0.00%)	65536 (-50.00%)	0.05 (+24.50%)	0 (+0.00%)

e.g. shared memory analysis
针对核函数中对shared memory访问以及使用效率的分析

如何使用(推荐使用ssh远程)

(*)主要用在当host端的nsight不能够直接ssh到target device情况下使用。比如说Jetson的profiling



方法一：在host端使用Nsight进行ssh远程profiling
方法二：在remote端使用Nsight进行直接profiling

方法三：在remote端通过CUI获取statistics之后传输到host端进行查看(*)
方法四：在remote端直接使用CUI进行分析(*)

两者的不同

Nsight systems

- 偏重于可视化application的整体的profiling以及各个细节指标，比如说
  - PCIe bindwidth
  - DRAM bindwidth
  - SM Warp occupancy
  - 所有核函数的调度信息
  - 所有核函数的执行时间，以及占用整体时间的比例
  - 多个Stream之间的调度信息
  - 同一个stream中的多个队列的调度信息
  - CPU和GPU之间的数据传输耗时
  - Application整体上的各个核函数以及操作的消耗时间排序
  - 捕捉同一个stream中的多个event

- 整体上会提供一些比较全面的信息，我们一般会从这里得到很多信息进而进行优化

两者的不同

Nsight compute

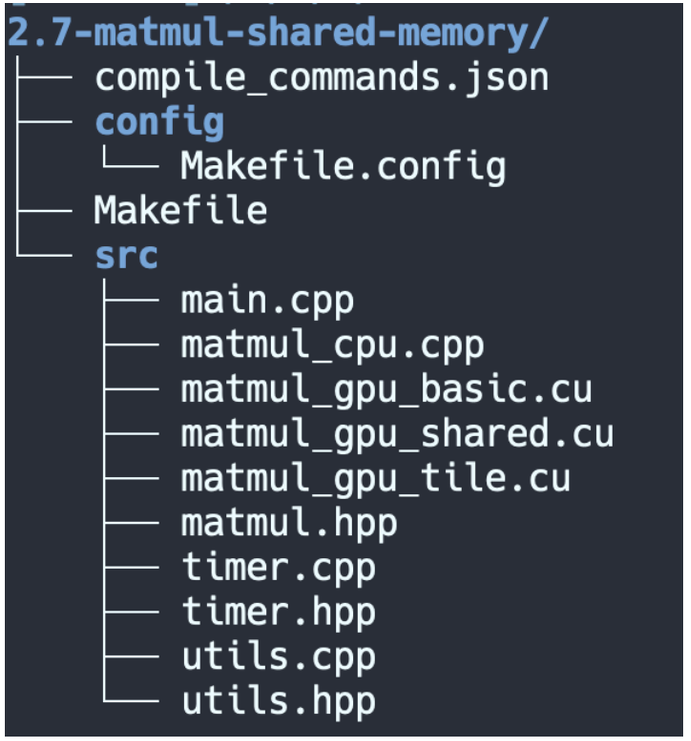
- 偏重于可视化每一个CUDA kernel的profiling以及各个细节指标，比如说
  - SM中计算吞吐量
  - L1 cache数据传输吞吐量
  - L2 cache数据传输吞吐量
  - DRAM数据传输吞吐量
  - 当前核函数属于计算密集型还是访存密集型
    - Roofline model分析
  - 核函数中的L1 cache的cache hit几率, cache miss几率的多少
  - 核函数中各个代码部分的延迟
    - 精确到代码部分进行highlight
  - 核函数的load bandwidth, store bandwith, load次数, store次数
  - L1 cache/shared memory, L2 cache, global memory中的memory access scheduling
  - 设置baseline，来进行核函数的优化前后的效率对比

- 整体上能够得到一个针对某一个kernel的非常精确的profiling，源码级别的性能捕捉，以及优化推荐
- (个人经验而言，NVIDIA在这些profiling tool上非常的下功夫，使用过其他SDK的人通过对比会深有感触)

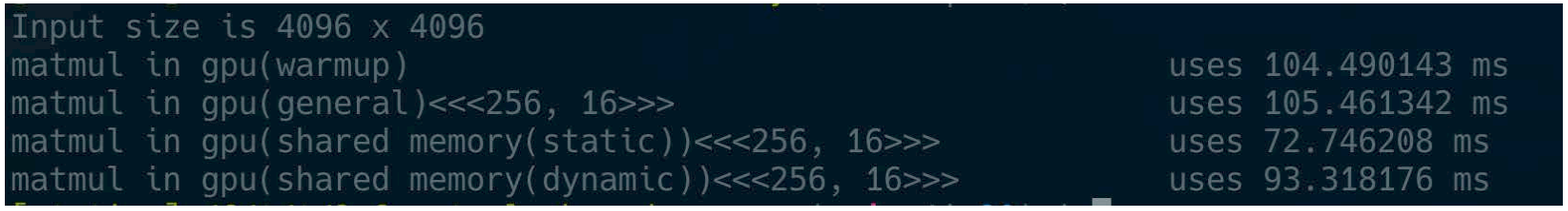
shared memory

Goal: 理解如何使用shared memory，为什么使用shared memory会有加速效果
以及在shared memory中使用动态/静态变量的注意事项

执行一下我们的第七个CUDA程序



2.7-matmul-shared-memory
(先不要管compile_commands.json，这个是neovim下进行c++函数定义声明调用跳转的东西。类似与.vscode)



make之后执行trt-cuda的一部分结果
(这里显示了对于一个4096x4096大小的矩阵进行mamtul时，使用shared memory的加速效果比较)

执行一下我们的第六个CUDA程序

```
/* GPU warmup */
timer.start_gpu();
MatmulOnDevice(h_matM, h_matN, h_matP, width, blockSize);
timer.stop_gpu();
timer.duration_gpu("matmul in gpu(warmup)");

/* GPU general implementation <<<256, 16>>>*/
timer.start_gpu();
MatmulOnDevice(h_matM, h_matN, d_matP, width, blockSize);
timer.stop_gpu();
std::sprintf(str, "matmul in gpu(general)<<<%d, %d>>>", width / blockSize, blockSize);
timer.duration_gpu(str);
compareMat(h_matP, d_matP, size);

// /* GPU general implementation <<<256, 16>>>*/
timer.start_gpu();
MatmulSharedOnDevice(h_matM, h_matN, d_matP, width, blockSize, statMem);
timer.stop_gpu();
std::sprintf(str, "matmul in gpu(shared memory(static))<<<%d, %d>>>", width / blockSize, blockSize);
timer.duration_gpu(str);
compareMat(h_matP, d_matP, size);

/* GPU general implementation <<<256, 16>>>*/
statMem = false;
timer.start_gpu();
MatmulSharedOnDevice(h_matM, h_matN, d_matP, width, blockSize, statMem);
timer.stop_gpu();
std::sprintf(str, "matmul in gpu(shared memory(dynamic))<<<%d, %d>>>", width / blockSize, blockSize);
timer.duration_gpu(str);
compareMat(h_matP, d_matP, size);
```

main.cpp

(使用cuda以<<<256, 16>>>的大小执行matmul的三种不同方式)

- 普通的不使用shared memory
- 使用shared memory，并使用静态变量
- 使用shared memory，并使用动态变量

执行一下我们的第六个CUDA程序

新添加的东西

```
public:
    Timer();
    ~Timer();

public:
    void start_cpu();
    void start_gpu();
    void stop_cpu();
    void stop_gpu();

    template <typename span>
    void duration_cpu(std::string msg);

    void duration_gpu(std::string msg);

private:
    std::chrono::time_point<std::chrono::high_resolution_clock> _cStart;
    std::chrono::time_point<std::chrono::high_resolution_clock> _cStop;
    cudaEvent_t _gStart;
    cudaEvent_t _gStop;
    float _timeElasped;
};
```

timer.hpp

```
23 void Timer::start_gpu() {
24     cudaEventRecord(_gStart, 0);
25 }
26
27 void Timer::stop_gpu() {
28     cudaEventRecord(_gStop, 0);
29 }
```

```
39 void Timer::duration_gpu(std::string msg){
40     CUDA_CHECK(cudaEventSynchronize(_gStart));
41     CUDA_CHECK(cudaEventSynchronize(_gStop));
42     cudaEventElapsedTime(&_timeElasped, _gStart, _gStop);
43
44     // cudaDeviceSynchronize();
45     // LAST_KERNEL_CHECK();
46
47     LOG("%-60s uses %.6lf ms", msg.c_str(), _timeElasped);
48 }
```

timer.cpp

为了实现更加精准的kernel测速，这里采用了event进行标记。
event中文名字叫做”事件”，可以用来标记cuda的stream中某一个执行点。一般用在多个stream同步或者监听某个stream的执行(有关stream和event，在之后的案例中会细讲)

执行一下我们的第六个CUDA程序

新添加的东西

```cpp
/*
    使用shared memory把计算一个tile所需要的数据分块存储到访问速度快的memory中
*/
__global__ void MatmulSharedStaticKernel(float *M_device, float *N_device, float *P_device, int width){
    __shared__ float M_deviceShared[BLOCKSIZE][BLOCKSIZE];
    __shared__ float N_deviceShared[BLOCKSIZE][BLOCKSIZE];
    /*
        对于x和y，根据blockID，tile大小和threadID进行索引
    */
    int x = blockIdx.x * BLOCKSIZE + threadIdx.x;
    int y = blockIdx.y * BLOCKSIZE + threadIdx.y;

    float P_element = 0.0;

    int ty = threadIdx.y;
    int tx = threadIdx.x;
    /* 对于每一个P的元素，我们只需要循环遍历width / tile_width 次就okay了，这里有点绕，画图理解一下*/
    for (int m = 0; m < width / BLOCKSIZE; m ++) {
        M_deviceShared[ty][tx] = M_device[y * width + (m * BLOCKSIZE + tx)];
        N_deviceShared[ty][tx] = N_device[(m * BLOCKSIZE + ty)* width + x];
        __syncthreads();

        for (int k = 0; k < BLOCKSIZE; k ++) {
            P_element += M_deviceShared[ty][k] * N_deviceShared[k][tx];
        }
        __syncthreads();
    }

    P_device[y * width + x] = P_element;
}
```

matmul_gpu_shared.cu
(使用静态共享变量)

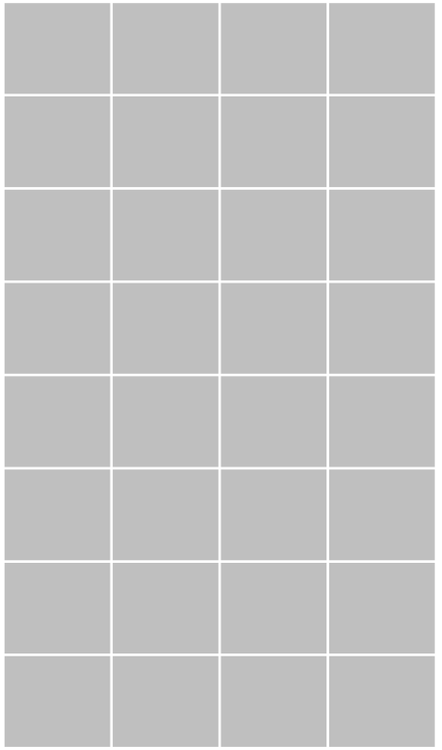
```cpp
__global__ void MatmulSharedDynamicKernel(float *M_device, float *N_device, float *P_device, int width, int
    /*
        声明动态共享变量的时候需要加extern，同时需要是一维的
        注意这里有个坑，不能够像这样定义：
            __shared__ float M_deviceShared[];
            __shared__ float N_deviceShared[];
        因为在cuda中定义动态共享变量的话，无论定义多少个他们的地址都是一样的。
        所以如果想要像上面这样使用的话，需要用两个指针分别指向shared memory的不同位置才行
    */

    extern __shared__ float deviceShared[];
    int stride = blockSize * blockSize;
    /*
        对于x和y，根据blockID，tile大小和threadID进行索引
    */
    int x = blockIdx.x * blockSize + threadIdx.x;
    int y = blockIdx.y * blockSize + threadIdx.y;

    float P_element = 0.0;

    int ty = threadIdx.y;
    int tx = threadIdx.x;
    /* 对于每一个P的元素，我们只需要循环遍历width / tile_width 次就okay了 */
    for (int m = 0; m < width / blockSize; m ++) {
        deviceShared[ty * blockSize + tx] = M_device[y * width + (m * blockSize + tx)];
        deviceShared[stride + (ty * blockSize + tx)] = N_device[(m * blockSize + ty)* width + x];
        __syncthreads();

        for (int k = 0; k < blockSize; k ++) {
            P_element += deviceShared[ty * blockSize + k] * deviceShared[stride + (k * blockSize + tx)];
        }
        __syncthreads();
    }

    if (y < width && x < width) {
        P_device[y * width + x] = P_element;
    }
}
```

matmul_gpu_shared.cu
(使用动态共享变量)

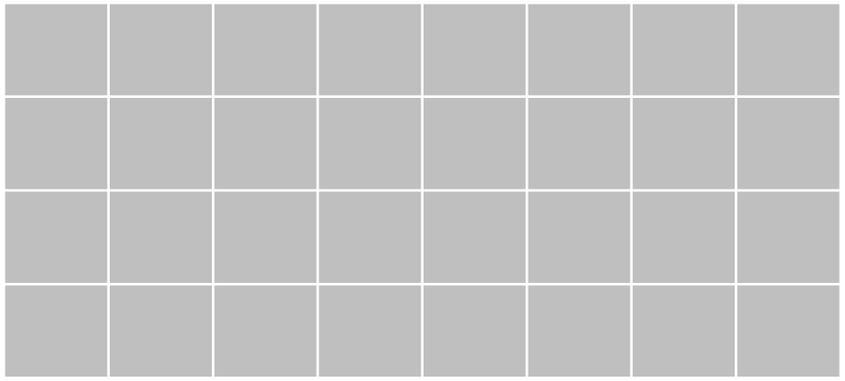
CUDA Core的矩阵乘法计算

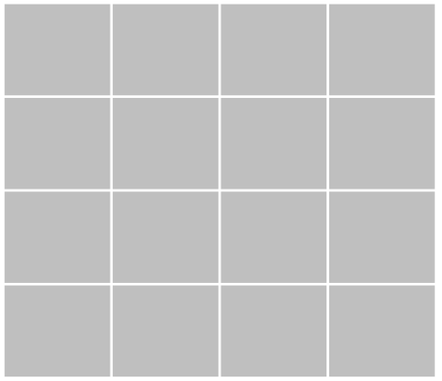
使用CUDA Core
计算 C = A * B

B(8 x 4)



A(4 x 8)



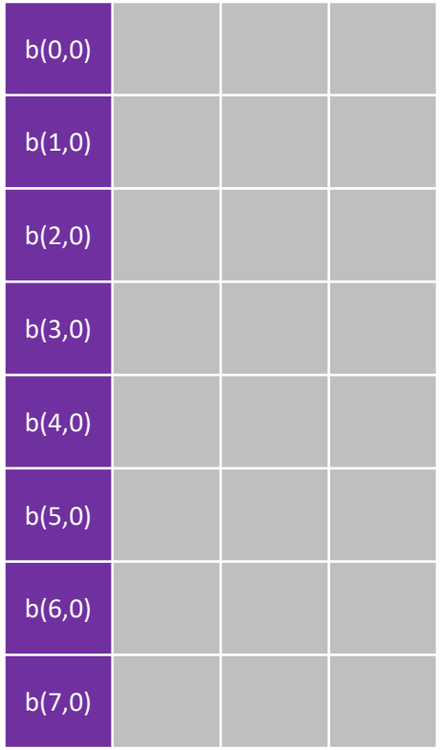


C(4 x 4)

CUDA Core的矩阵乘法计算

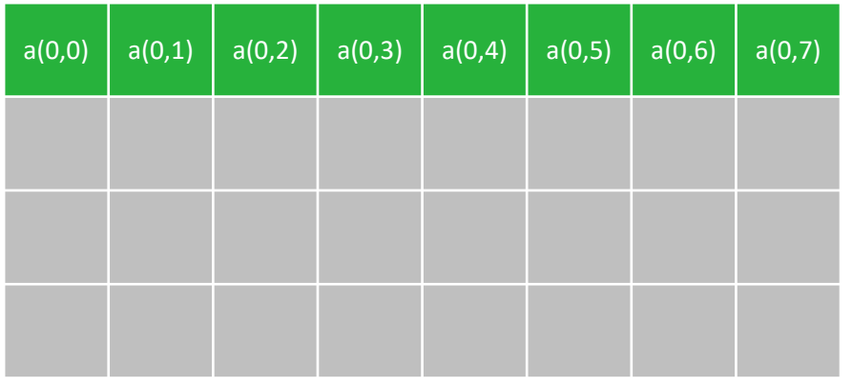
使用CUDA Core
计算 C = A * B

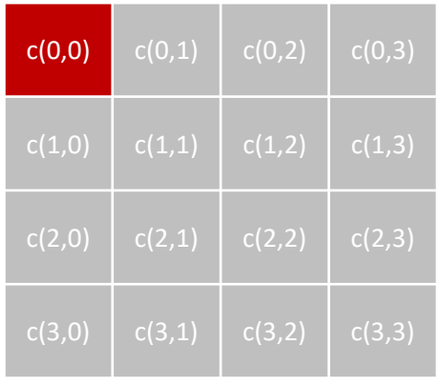
B(8 x 4)



$$
\begin{aligned}
c(0,0) &= a(0,0) * b(0,0) \\
&+ a(0,1) * b(1,0) \\
&+ a(0,2) * b(2,0) \\
&+ a(0,3) * b(3,0) \\
&+ a(0,4) * b(4,0) \\
&+ a(0,5) * b(5,0) \\
&+ a(0,6) * b(6,0) \\
&+ a(0,7) * b(7,0)
\end{aligned}
$$

A(4 x 8)





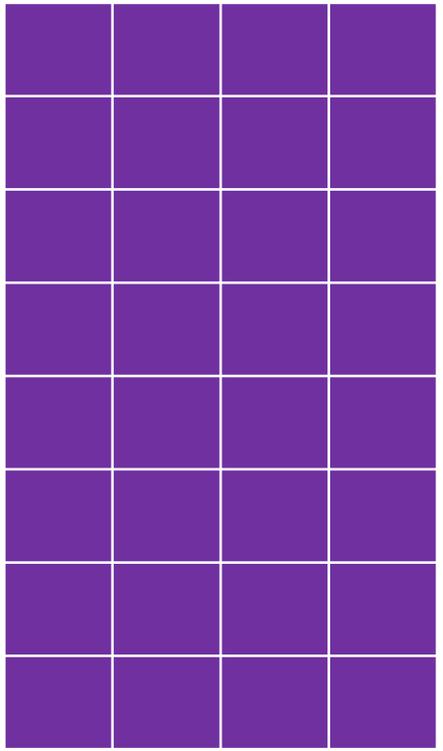
C(4 x 4)

要如果我们分配16个thread，每一个thread负责c中的一个元素...

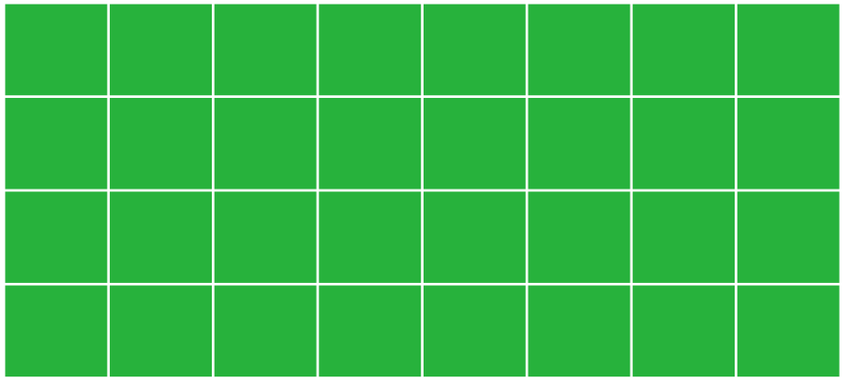
CUDA Core的矩阵乘法计算

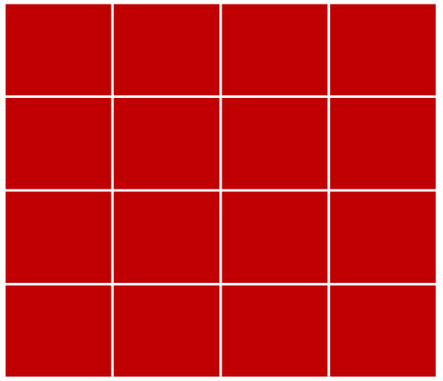
使用CUDA Core
计算 C = A * B

B(8 x 4)



A(4 x 8)





C(4 x 4)

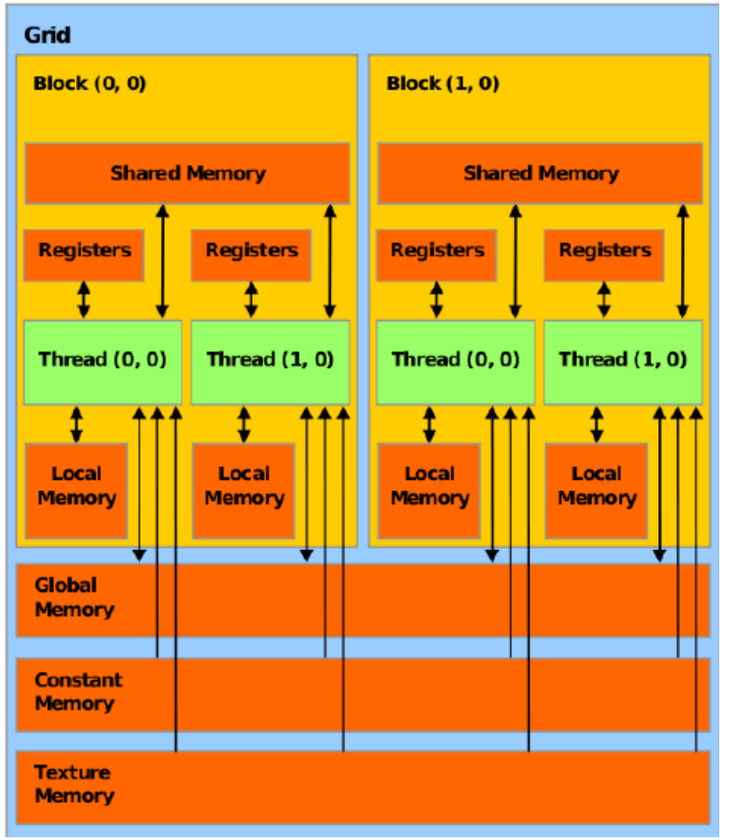
所有的thread一起同时执行的话，要完成4 x 8与 8x 4 的计算，和一个1x8与8x1的计算所需要的时间是一样的。8个clk就可以完成了

CUDA Core的矩阵乘法计算

使用CUDA Core
计算 C = A * B

B(8 x 4)

b(0,0)
b(1,0)
b(2,0)
b(3,0)
b(4,0)
b(5,0)
b(6,0)
b(7,0)



A(4 x 8)

a(0,0)	a(0,1)	a(0,2)	a(0,3)	a(0,4)	a(0,5)	a(0,6)	a(0,7)

c(0,0)	c(0,1)	c(0,2)	c(0,3)
c(1,0)	c(1,1)	c(1,2)	c(1,3)
c(2,0)	c(2,1)	c(2,2)	c(2,3)
c(3,0)	c(3,1)	c(3,2)	c(3,3)

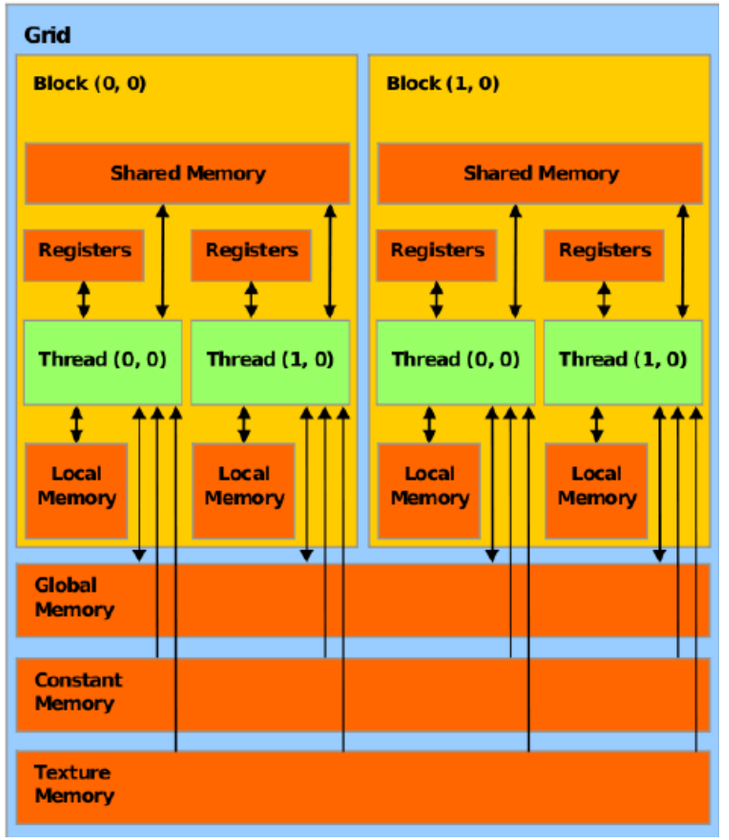
C(4 x 4)

CUDA Core的矩阵乘法计算

使用CUDA Core
计算 C = A * B

B(8 x 4)

b(0,0)	b(0,1)		
b(1,0)	b(1,1)		
b(2,0)	b(2,1)		
b(3,0)	b(3,1)		
b(4,0)	b(4,1)		
b(5,0)	b(5,1)		
b(6,0)	b(6,1)		
b(7,0)	b(7,1)		



A(4 x 8)

a(0,0)	a(0,1)	a(0,2)	a(0,3)	a(0,4)	a(0,5)	a(0,6)	a(0,7)

c(0,0)	c(0,1)	c(0,2)	c(0,3)
c(1,0)	c(1,1)	c(1,2)	c(1,3)
c(2,0)	c(2,1)	c(2,2)	c(2,3)
c(3,0)	c(3,1)	c(3,2)	c(3,3)

C(4 x 4)

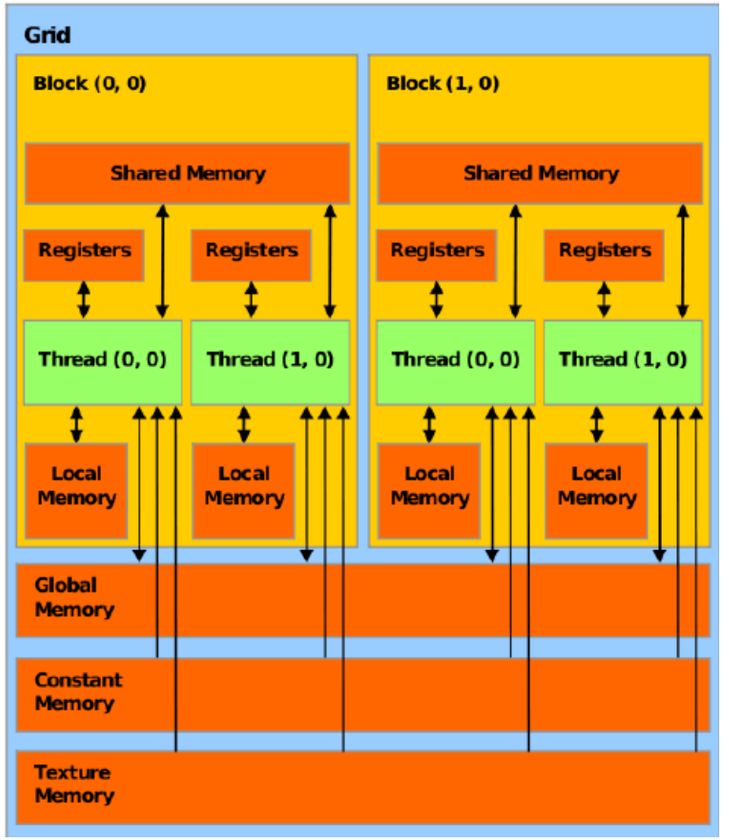
从global memory中重复的去loadA的同一块空间

CUDA Core的矩阵乘法计算

使用CUDA Core
计算 C = A * B

B(8 x 4)

b(0,0)	b(0,1)	b(0,2)	
b(1,0)	b(1,1)	b(1,2)	
b(2,0)	b(2,1)	b(2,2)	
b(3,0)	b(3,1)	b(3,2)	
b(4,0)	b(4,1)	b(4,2)	
b(5,0)	b(5,1)	b(5,2)	
b(6,0)	b(6,1)	b(6,2)	
b(7,0)	b(7,1)	b(7,2)	



从global memory中重复的去loadA的同一块空间

A(4 x 8)

a(0,0)	a(0,1)	a(0,2)	a(0,3)	a(0,4)	a(0,5)	a(0,6)	a(0,7)

c(0,0)	c(0,1)	c(0,2)	c(0,3)
c(1,0)	c(1,1)	c(1,2)	c(1,3)
c(2,0)	c(2,1)	c(2,2)	c(2,3)
c(3,0)	c(3,1)	c(3,2)	c(3,3)

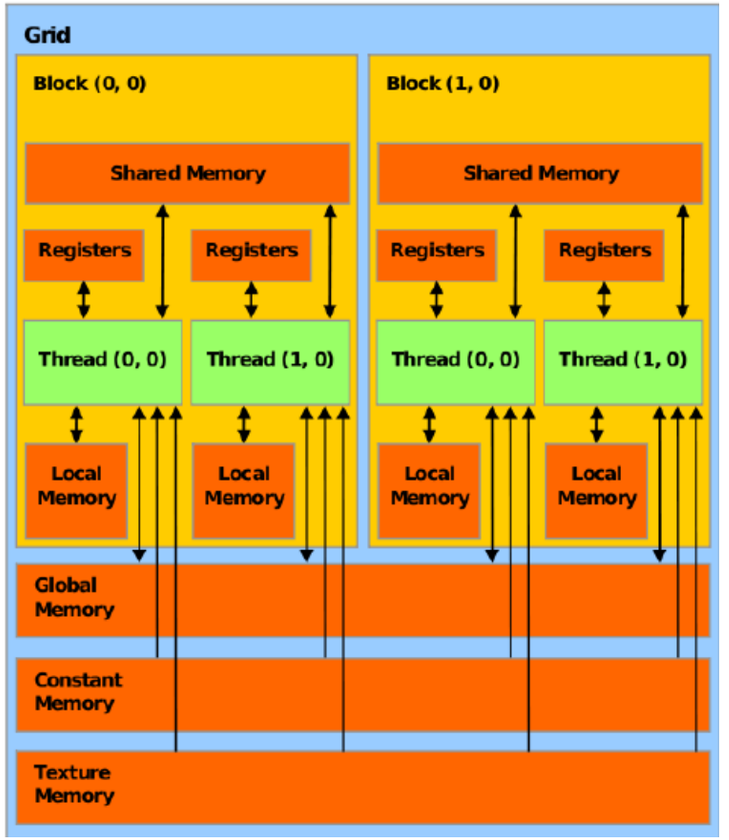
C(4 x 4)

CUDA Core的矩阵乘法计算

使用CUDA Core
计算 C = A * B

B(8 x 4)

b(0,0)	b(0,1)	b(0,2)	b(0,3)
b(1,0)	b(1,1)	b(1,2)	b(1,3)
b(2,0)	b(2,1)	b(2,2)	b(2,3)
b(3,0)	b(3,1)	b(3,2)	b(3,3)
b(4,0)	b(4,1)	b(4,2)	b(4,3)
b(5,0)	b(5,1)	b(5,2)	b(5,3)
b(6,0)	b(6,1)	b(6,2)	b(6,3)
b(7,0)	b(7,1)	b(7,2)	b(7,3)



从global memory中重复的去loadA的同一块空间

A(4 x 8)

a(0,0)	a(0,1)	a(0,2)	a(0,3)	a(0,4)	a(0,5)	a(0,6)	a(0,7)

c(0,0)	c(0,1)	c(0,2)	c(0,3)
c(1,0)	c(1,1)	c(1,2)	c(1,3)
c(2,0)	c(2,1)	c(2,2)	c(2,3)
c(3,0)	c(3,1)	c(3,2)	c(3,3)

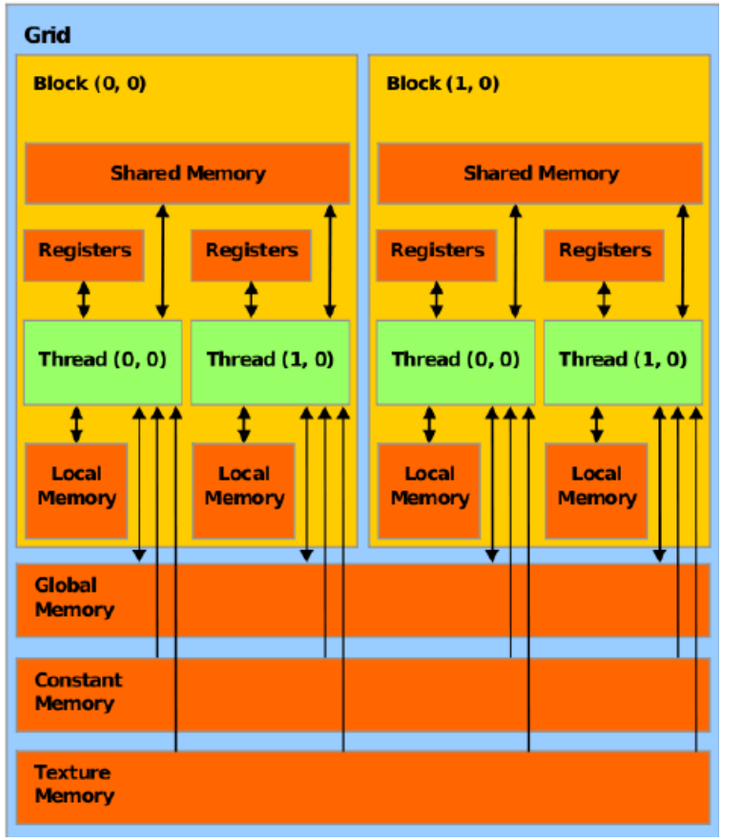
C(4 x 4)

CUDA Core的矩阵乘法计算

使用CUDA Core
计算 C = A * B

B(8 x 4)

b(0,0)
b(1,0)
b(2,0)
b(3,0)
b(4,0)
b(5,0)
b(6,0)
b(7,0)



A(4 x 8)

a(0,0)	a(0,1)	a(0,2)	a(0,3)	a(0,4)	a(0,5)	a(0,6)	a(0,7)

c(0,0)	c(0,1)	c(0,2)	c(0,3)
c(1,0)	c(1,1)	c(1,2)	c(1,3)
c(2,0)	c(2,1)	c(2,2)	c(2,3)
c(3,0)	c(3,1)	c(3,2)	c(3,3)

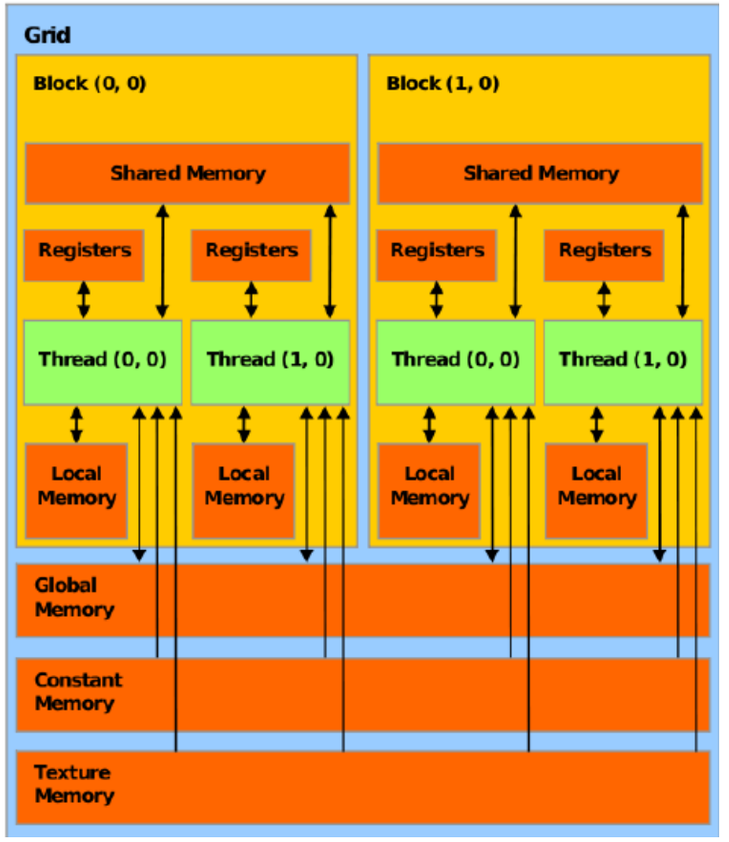
C(4 x 4)

CUDA Core的矩阵乘法计算

使用CUDA Core
计算 C = A * B

B(8 x 4)

b(0,0)			
b(1,0)			
b(2,0)			
b(3,0)			
b(4,0)			
b(5,0)			
b(6,0)			
b(7,0)			



从global memory中重复的去loadB的同一块空间

A(4 x 8)

a(0,0)	a(0,1)	a(0,2)	a(0,3)	a(0,4)	a(0,5)	a(0,6)	a(0,7)
a(1,0)	a(1,1)	a(1,2)	a(1,3)	a(1,4)	a(1,5)	a(1,6)	a(1,7)

c(0,0)	c(0,1)	c(0,2)	c(0,3)
c(1,0)	c(1,1)	c(1,2)	c(1,3)
c(2,0)	c(2,1)	c(2,2)	c(2,3)
c(3,0)	c(3,1)	c(3,2)	c(3,3)

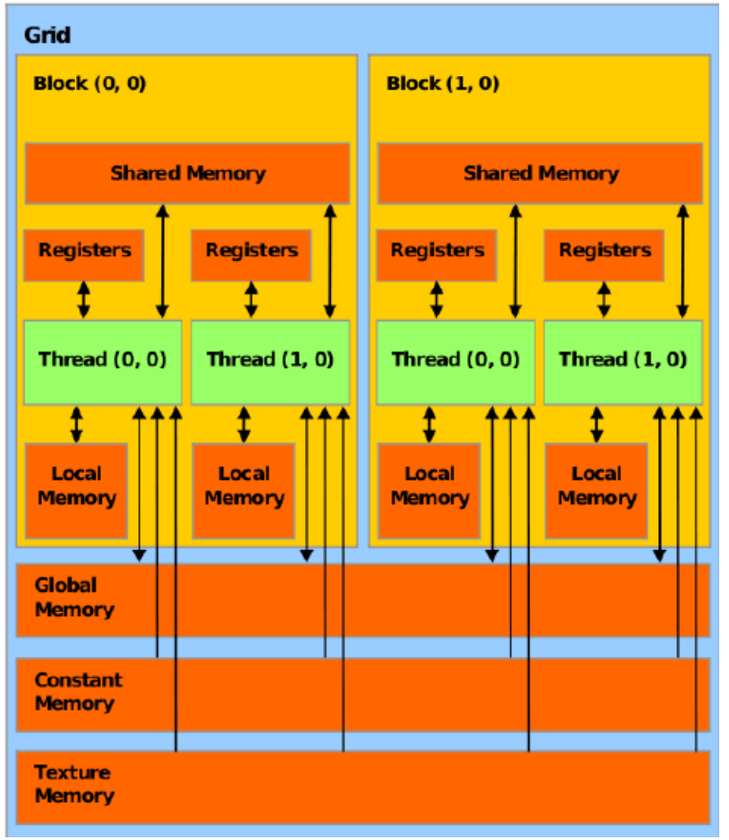
C(4 x 4)

CUDA Core的矩阵乘法计算

使用CUDA Core
计算 C = A * B

B(8 x 4)

b(0,0)
b(1,0)
b(2,0)
b(3,0)
b(4,0)
b(5,0)
b(6,0)
b(7,0)



从global memory中重复的去loadB的同一块空间

A(4 x 8)

a(0,0)	a(0,1)	a(0,2)	a(0,3)	a(0,4)	a(0,5)	a(0,6)	a(0,7)
a(1,0)	a(1,1)	a(1,2)	a(1,3)	a(1,4)	a(1,5)	a(1,6)	a(1,7)
a(2,0)	a(2,1)	a(2,2)	a(2,3)	a(2,4)	a(2,5)	a(2,6)	a(2,7)

c(0,0)	c(0,1)	c(0,2)	c(0,3)
c(1,0)	c(1,1)	c(1,2)	c(1,3)
c(2,0)	c(2,1)	c(2,2)	c(2,3)
c(3,0)	c(3,1)	c(3,2)	c(3,3)

C(4 x 4)

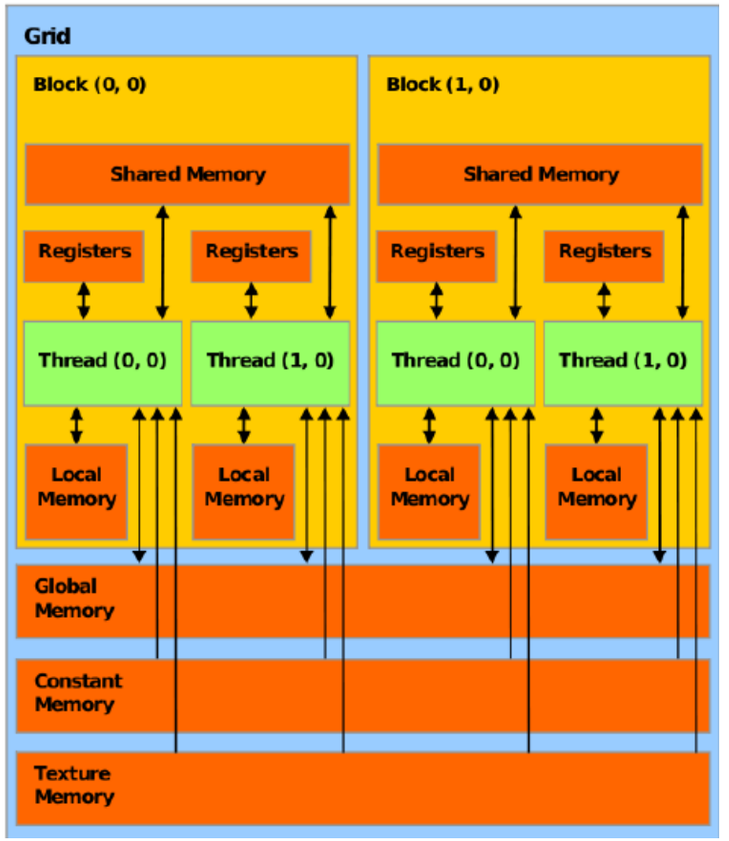
CUDA Core的矩阵乘法计算

使用CUDA Core
计算 C = A * B

思考：这个过程是否存在冗长的操作？如果有怎么办？

B(8 x 4)

b(0,0)
b(1,0)
b(2,0)
b(3,0)
b(4,0)
b(5,0)
b(6,0)
b(7,0)



从global memory中重复的去loadB的同一块空间

A(4 x 8)

a(0,0)	a(0,1)	a(0,2)	a(0,3)	a(0,4)	a(0,5)	a(0,6)	a(0,7)
a(1,0)	a(1,1)	a(1,2)	a(1,3)	a(1,4)	a(1,5)	a(1,6)	a(1,7)
a(2,0)	a(2,1)	a(2,2)	a(2,3)	a(2,4)	a(2,5)	a(2,6)	a(2,7)
a(2,0)	a(2,1)	a(2,2)	a(2,3)	a(2,4)	a(2,5)	a(2,6)	a(2,7)

c(0,0)	c(0,1)	c(0,2)	c(0,3)
c(1,0)	c(1,1)	c(1,2)	c(1,3)
c(2,0)	c(2,1)	c(2,2)	c(2,3)
c(3,0)	c(3,1)	c(3,2)	c(3,3)

C(4 x 4)

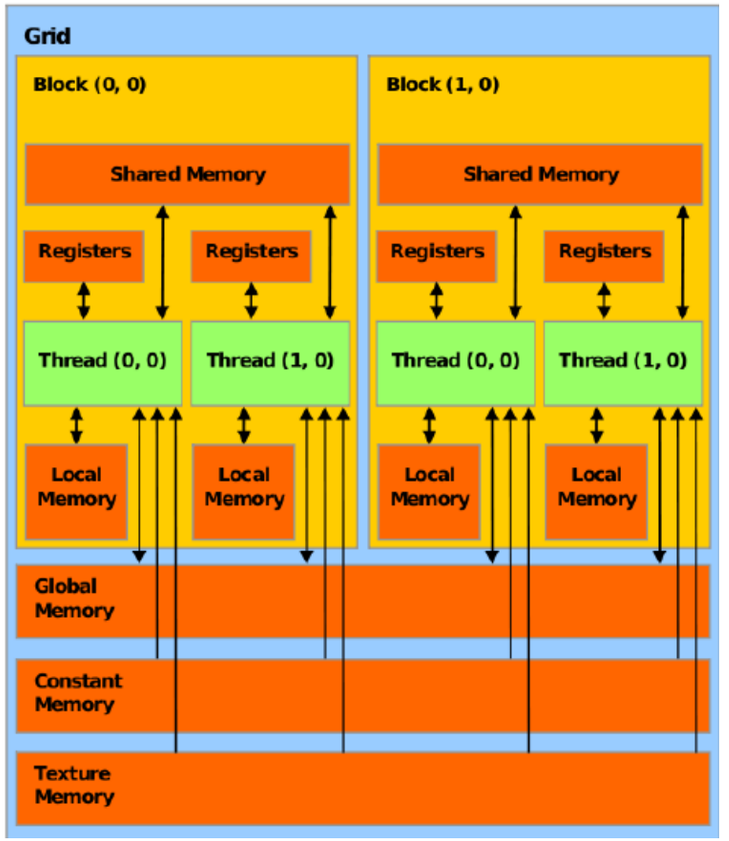
CUDA Core的矩阵乘法计算

使用CUDA Core
计算 C = A * B

一直访问glogal memory的话有点慢
如果数据可以在多次被复用的话，可以把他们放在可以快速访问的shared memory中

B(8 x 4)

b(0,0)
b(1,0)
b(2,0)
b(3,0)
b(4,0)
b(5,0)
b(6,0)
b(7,0)



从global memory中重复的去loadB的同一块空间

A(4 x 8)

a(0,0)	a(0,1)	a(0,2)	a(0,3)	a(0,4)	a(0,5)	a(0,6)	a(0,7)
a(1,0)	a(1,1)	a(1,2)	a(1,3)	a(1,4)	a(1,5)	a(1,6)	a(1,7)
a(2,0)	a(2,1)	a(2,2)	a(2,3)	a(2,4)	a(2,5)	a(2,6)	a(2,7)
a(2,0)	a(2,1)	a(2,2)	a(2,3)	a(2,4)	a(2,5)	a(2,6)	a(2,7)

c(0,0)	c(0,1)	c(0,2)	c(0,3)
c(1,0)	c(1,1)	c(1,2)	c(1,3)
c(2,0)	c(2,1)	c(2,2)	c(2,3)
c(3,0)	c(3,1)	c(3,2)	c(3,3)

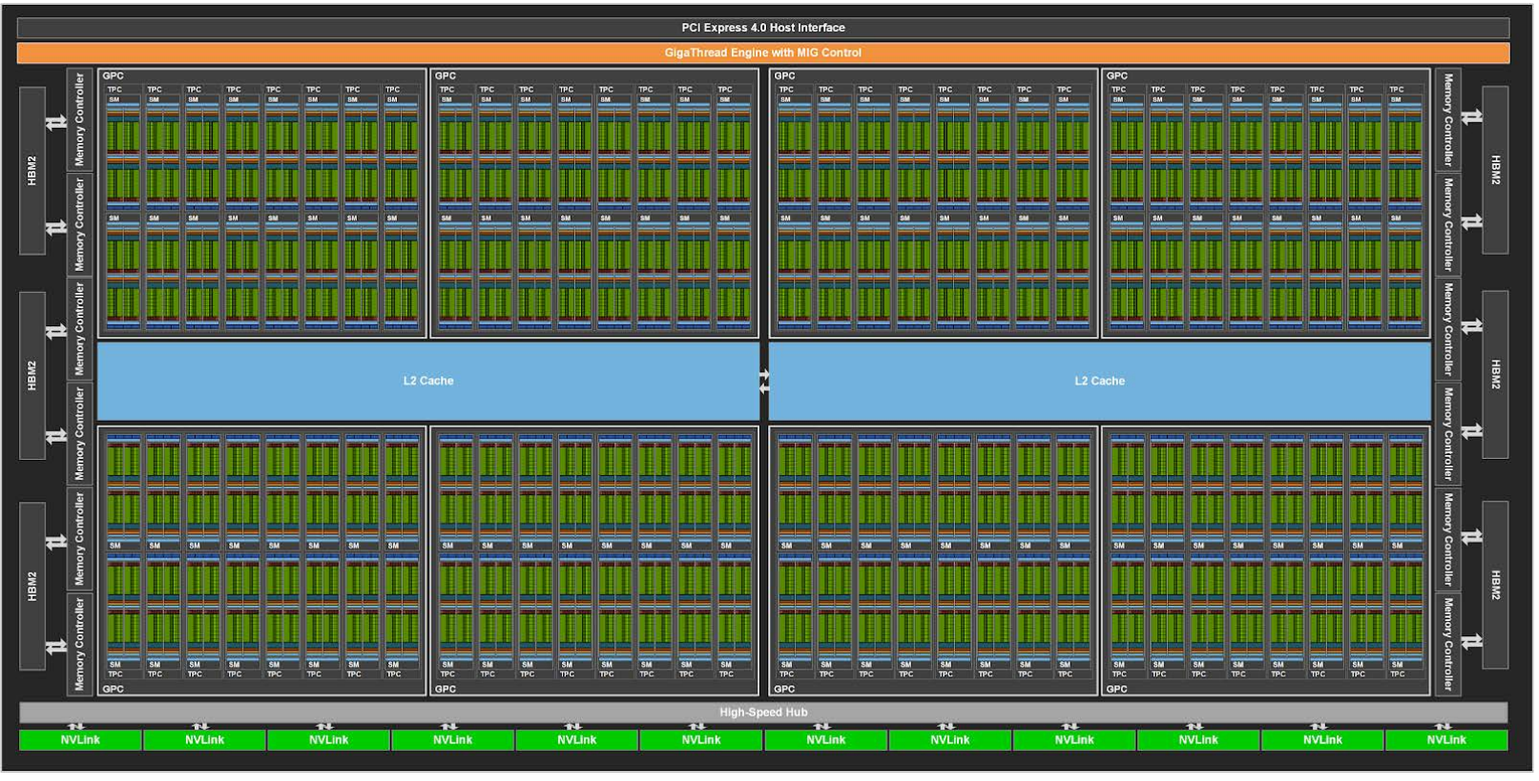
C(4 x 4)

Shared memory vs Global memory

Shared memory: on-chip memory
Global memory: off-chip memory (DRAM)

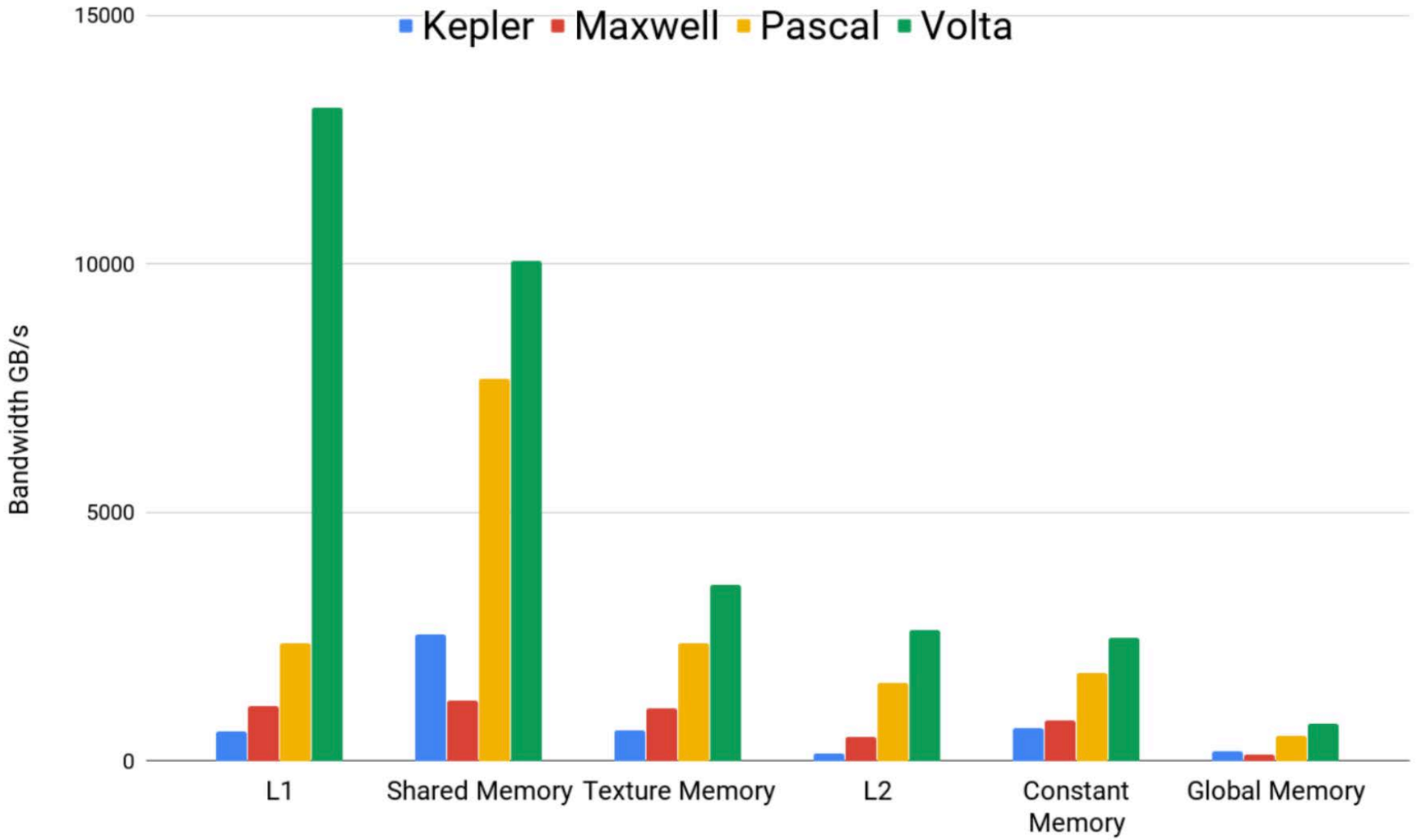
L1/L2 cache和shared memory都是属于on-chip memory，memory load/store的overhead会比较小，是可以高速访问的memory

Global memory的延迟是最高的，我们一般在cudaMalloc时都是在global memory上进行访问的

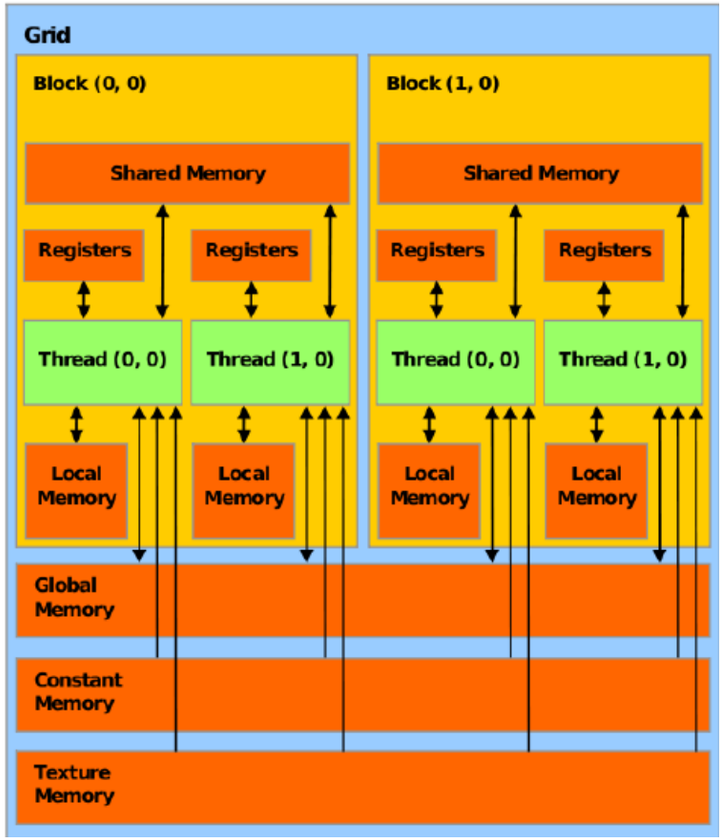


[1]NVIDIA Ampere Architecture In-Depth

Shared memory vs Global memory



不同架构的不同memory的bandwidth的不同[1]



[1]Is Data Placement Optimization Still Relevant On Newer GPUs, SC'18

Shared memory vs Global memory

register: 线程独享，访问最快大小最小，on-chip
L1 cache: SM内共享, on-chip
shared memory: SM内共享, on-chip
L2 cache: SM间共享, on-chip

local memory: 线程独享，寄存器不足的时候使用，off-chip
Global memory: 设备内所有线程共享, off-chip
constant memory: 只读内存，用与避免线程独数据冲突，off-chip
texture memory: 只读内存，tex可以实现硬件插值, off-chip

CUDA Core的矩阵乘法计算

```cpp
/* matmul的函数实现*/
__global__ void MatmulKernel(float *M_device, float *N_device,
    /*
        我们设定每一个thread负责P中的一个坐标的matmul
        所以一共有width * width个thread并行处理P的计算
    */
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    float P_element = 0;

    /* 对于每一个P的元素，我们只需要循环遍历width次M和N中的元素
    for (int k = 0; k < width; k ++){
        float M_element = M_device[y * width + k];
        float N_element = N_device[k * width + x];
        P_element += M_element * N_element;
    }

    P_device[y * width + x] = P_element;
}
```

matmul_gpu_basic.cu
(不使用shared memory)

```cpp
__global__ void MatmulSharedStaticKernel(float *M_device, float *N_device, flo
    __shared__ float M_deviceShared[BLOCKSIZE][BLOCKSIZE];
    __shared__ float N_deviceShared[BLOCKSIZE][BLOCKSIZE];
    /*
        对于x和y，根据blockID，tile大小和threadID进行索引
    */
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    float P_element = 0.0;

    int ty = threadIdx.y;
    int tx = threadIdx.x;
    /* 对于每一个P的元素，我们只需要循环遍历width / tile_width 次就okay了，这里
    for (int m = 0; m < width / BLOCKSIZE; m ++) {
        M_deviceShared[ty][tx] = M_device[y * width + (m * BLOCKSIZE + tx)];
        N_deviceShared[ty][tx] = N_device[(m * BLOCKSIZE + ty)* width + x];
        __syncthreads();

        for (int k = 0; k < BLOCKSIZE; k ++) {
            P_element += M_deviceShared[ty][k] * N_deviceShared[k][tx];
        }
        __syncthreads();
    }

    P_device[y * width + x] = P_element;
}
```

matmul_gpu_shared.cu
(使用shared memory)

两者主要在遍历M(A), 和N(B)的方式不同

CUDA Core的矩阵乘法计算

```cpp
__global__ void MatmulSharedStaticKernel(float *M_device, float *N_device, flo
    __shared__ float M_deviceShared[BLOCKSIZE][BLOCKSIZE];
    __shared__ float N_deviceShared[BLOCKSIZE][BLOCKSIZE];
    /*
        对于x和y，根据blockID，tile大小和threadID进行索引
    */
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    float P_element = 0.0;

    int ty = threadIdx.y;
    int tx = threadIdx.x;
    /* 对于每一个P的元素，我们只需要循环遍历width / tile_width 次就okay了，这里
    for (int m = 0; m < width / BLOCKSIZE; m ++) {
        M_deviceShared[ty][tx] = M_device[y * width + (m * BLOCKSIZE + tx)];
        N_deviceShared[ty][tx] = N_device[(m * BLOCKSIZE + ty)* width + x];
        __syncthreads();

        for (int k = 0; k < BLOCKSIZE; k ++) {
            P_element += M_deviceShared[ty][k] * N_deviceShared[k][tx];
        }
        __syncthreads();
    }

    P_device[y * width + x] = P_element;
}
```

A(4 x 8)

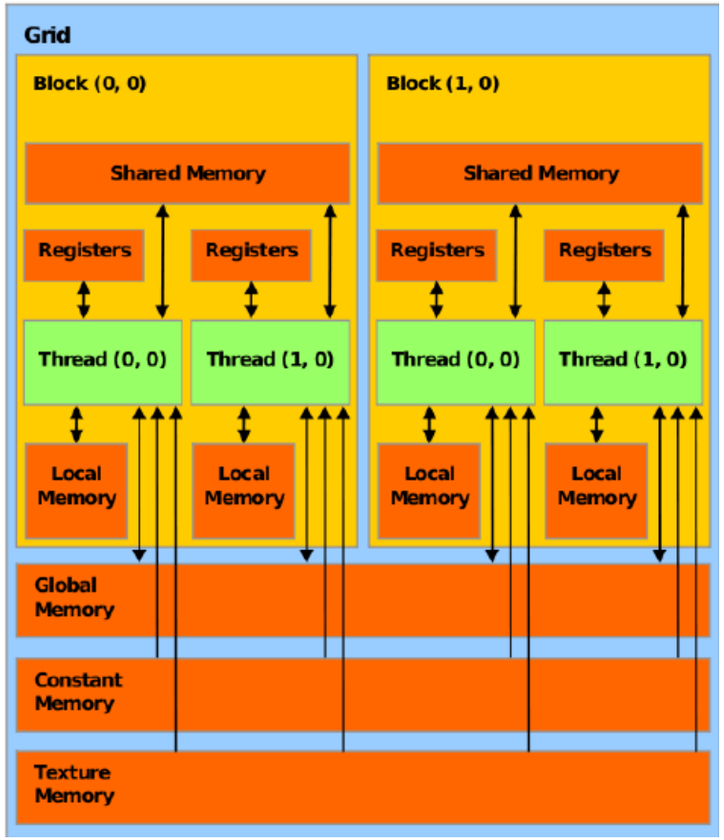
a(0,0)	a(0,1)	a(0,2)	a(0,3)	a(0,4)	a(0,5)	a(0,6)	a(0,7)
a(1,0)	a(1,1)	a(1,2)	a(1,3)	a(1,4)	a(1,5)	a(1,6)	a(1,7)
a(2,0)	a(2,1)	a(2,2)	a(2,3)	a(2,4)	a(2,5)	a(2,6)	a(2,7)
a(3,0)	a(3,1)	a(3,2)	a(3,3)	a(3,4)	a(3,5)	a(3,6)	a(3,7)

B(8 x 4)

b(0,0)	b(0,1)	b(0,2)	b(0,3)
b(1,0)	b(1,1)	b(1,2)	b(1,3)
b(2,0)	b(2,1)	b(2,2)	b(2,3)
b(3,0)	b(3,1)	b(3,2)	b(3,3)
b(4,0)	b(4,1)	b(4,2)	b(4,3)
b(5,0)	b(5,1)	b(5,2)	b(5,3)
b(6,0)	b(6,1)	b(6,2)	b(6,3)
b(7,0)	b(7,1)	b(7,2)	b(7,3)

c(0,0)	c(0,1)	c(0,2)	c(0,3)
c(1,0)	c(1,1)	c(1,2)	c(1,3)
c(2,0)	c(2,1)	c(2,2)	c(2,3)
c(3,0)	c(3,1)	c(3,2)	c(3,3)

C(4 x 4)

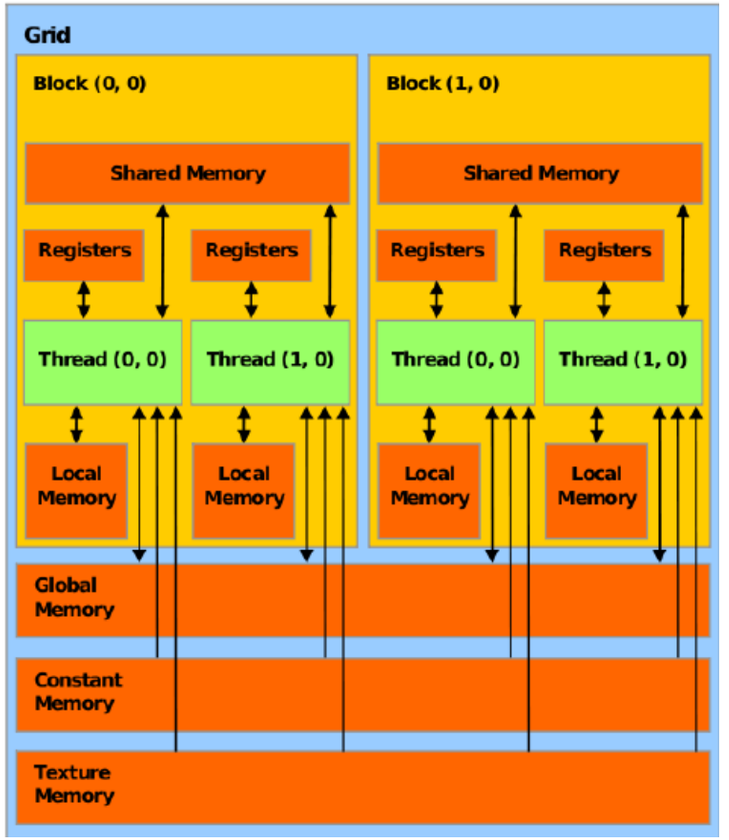


CUDA Core的矩阵乘法计算

```cpp
__global__ void MatmulSharedStaticKernel(float *M_device, float *N_device, flo
    __shared__ float M_deviceShared[BLOCKSIZE][BLOCKSIZE];
    __shared__ float N_deviceShared[BLOCKSIZE][BLOCKSIZE];
    /*
        对于x和y，根据blockID，tile大小和threadID进行索引
    */
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    float P_element = 0.0;

    int ty = threadIdx.y;
    int tx = threadIdx.x;
    /* 对于每一个P的元素，我们只需要循环遍历width / tile_width 次就okay了，这里
    for (int m = 0; m < width / BLOCKSIZE; m ++) {
        M_deviceShared[ty][tx] = M_device[y * width + (m * BLOCKSIZE + tx)];
        N_deviceShared[ty][tx] = N_device[(m * BLOCKSIZE + ty)* width + x];
        __syncthreads();

        for (int k = 0; k < BLOCKSIZE; k ++) {
            P_element += M_deviceShared[ty][k] * N_deviceShared[k][tx];
        }
        __syncthreads();
    }

    P_device[y * width + x] = P_element;
}
```

B(8 x 4)

b(0,0)	b(0,1)	b(0,2)	b(0,3)
b(1,0)	b(1,1)	b(1,2)	b(1,3)
b(2,0)	b(2,1)	b(2,2)	b(2,3)
b(3,0)	b(3,1)	b(3,2)	b(3,3)
b(4,0)	b(4,1)	b(4,2)	b(4,3)
b(5,0)	b(5,1)	b(5,2)	b(5,3)
b(6,0)	b(6,1)	b(6,2)	b(6,3)
b(7,0)	b(7,1)	b(7,2)	b(7,3)



A(4 x 8)

a(0,0)	a(0,1)	a(0,2)	a(0,3)	a(0,4)	a(0,5)	a(0,6)	a(0,7)
a(1,0)	a(1,1)	a(1,2)	a(1,3)	a(1,4)	a(1,5)	a(1,6)	a(1,7)
a(2,0)	a(2,1)	a(2,2)	a(2,3)	a(2,4)	a(2,5)	a(2,6)	a(2,7)
a(3,0)	a(3,1)	a(3,2)	a(3,3)	a(3,4)	a(3,5)	a(3,6)	a(3,7)

c(0,0)	c(0,1)	c(0,2)	c(0,3)
c(1,0)	c(1,1)	c(1,2)	c(1,3)
c(2,0)	c(2,1)	c(2,2)	c(2,3)
c(3,0)	c(3,1)	c(3,2)	c(3,3)

C(4 x 4)

CUDA Core的矩阵乘法计算

```
__global__ void MatmulSharedStaticKernel(float *M_device, float *N_device, flo
    __shared__ float M_deviceShared[BLOCKSIZE][BLOCKSIZE];
    __shared__ float N_deviceShared[BLOCKSIZE][BLOCKSIZE];
    /*
        对于x和y，根据blockID，tile大小和threadID进行索引
    */
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    float P_element = 0.0;

    int ty = threadIdx.y;
    int tx = threadIdx.x;
    /* 对于每一个P的元素，我们只需要循环遍历width / tile_width 次就okay了，这
    for (int m = 0; m < width / BLOCKSIZE; m ++) {
        M_deviceShared[ty][tx] = M_device[y * width + (m * BLOCKSIZE + tx)];
        N_deviceShared[ty][tx] = N_device[(m * BLOCKSIZE + ty)* width + x];
        __syncthreads();

        for (int k = 0; k < BLOCKSIZE; k ++) {
            P_element += M_deviceShared[ty][k] * N_deviceShared[k][tx];
        }
        __syncthreads();
    }

    P_device[y * width + x] = P_element;
}
```

A(4 x 8)

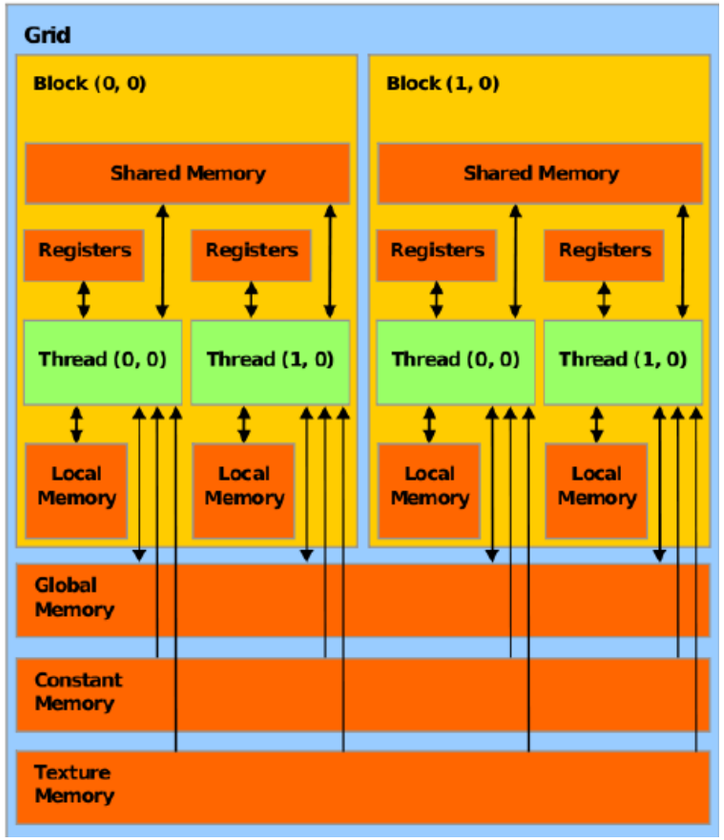
a(0,0)	a(0,1)	a(0,2)	a(0,3)	a(0,4)	a(0,5)	a(0,6)	a(0,7)
a(1,0)	a(1,1)	a(1,2)	a(1,3)	a(1,4)	a(1,5)	a(1,6)	a(1,7)
a(2,0)	a(2,1)	a(2,2)	a(2,3)	a(2,4)	a(2,5)	a(2,6)	a(2,7)
a(3,0)	a(3,1)	a(3,2)	a(3,3)	a(3,4)	a(3,5)	a(3,6)	a(3,7)

B(8 x 4)

b(0,0)	b(0,1)	b(0,2)	b(0,3)
b(1,0)	b(1,1)	b(1,2)	b(1,3)
b(2,0)	b(2,1)	b(2,2)	b(2,3)
b(3,0)	b(3,1)	b(3,2)	b(3,3)
b(4,0)	b(4,1)	b(4,2)	b(4,3)
b(5,0)	b(5,1)	b(5,2)	b(5,3)
b(6,0)	b(6,1)	b(6,2)	b(6,3)
b(7,0)	b(7,1)	b(7,2)	b(7,3)

c(0,0)	c(0,1)	c(0,2)	c(0,3)
c(1,0)	c(1,1)	c(1,2)	c(1,3)
c(2,0)	c(2,1)	c(2,2)	c(2,3)
c(3,0)	c(3,1)	c(3,2)	c(3,3)

C(4 x 4)



CUDA Core的矩阵乘法计算

如果使用动态共享变量，方法流程跟静态是一样的，但需要注意几个点：

- 动态申请的时候需要是一维的
- 动态申请的变量地址都是一样的

使用动态共享变量速度会慢一点

```
__global__ void MatmulSharedDynamicKernel(float *M_device, float *N_device, float *P_device, int width, int bl
    /*
        声明动态共享变量的时候需要加extern，同时需要是一维的
        注意这里有个坑，不能够像这样定义：
            __shared__ float M_deviceShared[];
            __shared__ float N_deviceShared[];
        因为在cuda中定义动态共享变量的话，无论定义多少个他们的地址都是一样的。
        所以如果想要像上面这样使用的话，需要用两个指针分别指向shared memory的不同位置才行
    */

    extern __shared__ float deviceShared[];
    int stride = blockSize * blockSize;
    /*
        对于x和y，根据blockID，tile大小和threadID进行索引
    */
    int x = blockIdx.x * blockSize + threadIdx.x;
    int y = blockIdx.y * blockSize + threadIdx.y;

    float P_element = 0.0;

    int ty = threadIdx.y;
    int tx = threadIdx.x;
    /* 对于每一个P的元素，我们只需要循环遍历width / tile_width 次就okay了 */
    for (int m = 0; m < width / blockSize; m ++) {
        deviceShared[ty * blockSize + tx] = M_device[y * width + (m * blockSize + tx)];
        deviceShared[stride + (ty * blockSize + tx)] = N_device[(m * blockSize + ty)* width + x];
        __syncthreads();

        for (int k = 0; k < blockSize; k ++) {
            P_element += deviceShared[ty * blockSize + k] * deviceShared[stride + (k * blockSize + tx)];
        }
        __syncthreads();
    }

    if (y < width && x < width) {
        P_device[y * width + x] = P_element;
    }
}
```

bank conflict

Goal: 理解什么在shared memory中的bank是什么，
什么时候会发生bank conflict，以及如何减缓bank conflict

执行一下我们的第八个CUDA程序


```
2.8-bank-conflict/
├── compile_commands.json
├── config
│   └── Makefile.config
├── Makefile
└── src
    ├── main.cpp
    ├── matmul_cpu.cpp
    ├── matmul_gpu_bank_conflict.cu
    ├── matmul_gpu_bank_conflict_pad.cu
    ├── matmul_gpu_basic.cu
    ├── matmul_gpu_shared.cu
    ├── matmul.hpp
    ├── timer.cpp
    ├── timer.hpp
    ├── utils.cpp
    └── utils.hpp
```

2.8-bank-conflict
(先不要管compile_commands.json，这个是neovim下进行c++函数定义声明调用跳转的东西。类似与.vscode)

```
Input size is 4096 x 4096
matmul in gpu(warmup)                                  uses 104.839714 ms
matmul in gpu(general)                                 uses 105.241791 ms
matmul in gpu(shared memory(static))                   uses 68.399872 ms
matmul in gpu(shared memory(static, bank conf))        uses 153.316193 ms
matmul in gpu(shared memory(static, pad resolve bank conf))  uses 114.701599 ms
matmul in gpu(shared memory(dynamic))                  uses 92.915970 ms
matmul in gpu(shared memory(dynamic, bank conf)        uses 215.873444 ms
matmul in gpu(shared memory(dynamic, pad resolve bank conf)) uses 114.717789 ms
```

make之后执行trt-cuda的一部分结果
(这里显示了使用shared memory中没有bank conflict和有bank conflict时的速度差异)

执行一下我们的第七个CUDA程序

```cpp
/*
    使用shared memory把计算一个tile所需要的数据分块存储到访问速度快的memory中
*/
__global__ void MatmulSharedStaticConflictKernel(float *M_device, float *N_devic
    __shared__ float M_deviceShared[BLOCKSIZE][BLOCKSIZE];
    __shared__ float N_deviceShared[BLOCKSIZE][BLOCKSIZE];
    /*
        对于x和y，根据blockID，tile大小和threadID进行索引
    */
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    float P_element = 0.0;

    int ty = threadIdx.y;
    int tx = threadIdx.x;
    /* 对于每一个P的元素，我们只需要循环遍历width / tile_width 次就okay了，这里有
    for (int m = 0; m < width / BLOCKSIZE; m ++) {
        /* 这里为了实现bank conflict，把tx与tx的顺序颠倒，同时索引也改变了*/
        M_deviceShared[tx][ty] = M_device[x * width + (m * BLOCKSIZE + ty)];
        N_deviceShared[tx][ty] = M_device[(m * BLOCKSIZE + tx)* width + y];

        __syncthreads();

        for (int k = 0; k < BLOCKSIZE; k ++) {
            P_element += M_deviceShared[tx][k] * N_deviceShared[k][ty];
        }
        __syncthreads();
    }

    /* 列优先 */
    P_device[x * width + y] = P_element;
```

```cpp
__global__ void MatmulSharedDynamicKernel(float *M_device, float *N_device, float *P_device,
    /*
        声明动态共享变量的时候需要加extern，同时需要是一维的
        注意这里有个坑，不能够像这样定义：
            __shared__ float M_deviceShared[];
            __shared__ float N_deviceShared[];
        因为在cuda中定义动态共享变量的话，无论定义多少个他们的地址都是一样的。
        所以如果想要像上面这样使用的话，需要用两个指针分别指向shared memory的不同位置才行
    */

    extern __shared__ float deviceShared[];
    int stride = blockSize * blockSize;
    /*
        对于x和y，根据blockID，tile大小和threadID进行索引
    */
    int x = blockIdx.x * blockSize + threadIdx.x;
    int y = blockIdx.y * blockSize + threadIdx.y;

    float P_element = 0.0;

    int ty = threadIdx.y;
    int tx = threadIdx.x;
    /* 对于每一个P的元素，我们只需要循环遍历width / tile_width 次就okay了 */
    for (int m = 0; m < width / blockSize; m ++) {
        deviceShared[ty * blockSize + tx] = M_device[y * width + (m * blockSize + tx)];
        deviceShared[stride + (ty * blockSize + tx)] = N_device[(m * blockSize + ty)* width +
        __syncthreads();

        for (int k = 0; k < blockSize; k ++) {
            P_element += deviceShared[ty * blockSize + k] * deviceShared[stride + (k * blockS
        }
        __syncthreads();
    }

    P_device[y * width + x] = P_element;
}
```

matmul_gpu_bank_conflict.cu (左:静态, 右: 动态)
(在matmul的过程中，矩阵的遍历方式从行优先转为列优先。从而
人为的让matmul的过程发生bank conflict)

执行一下我们的第七个CUDA程序

```cuda
/*
    使用shared memory把计算一个tile所需要的数据分块存储到访问速度快的memory中
*/
__global__ void MatmulSharedStaticConflictPadKernel(float *M_device, float *N_dev
    /* 添加一个padding, 可以防止bank conflict发生, 结合图理解一下*/
    __shared__ float M_deviceShared[BLOCKSIZE][BLOCKSIZE + 1];
    __shared__ float N_deviceShared[BLOCKSIZE][BLOCKSIZE + 1];
    /*
        对于x和y, 根据blockID, tile大小和threadID进行索引
    */
    int x = blockIdx.x * BLOCKSIZE + threadIdx.x;
    int y = blockIdx.y * BLOCKSIZE + threadIdx.y;

    float P_element = 0.0;

    int ty = threadIdx.y;
    int tx = threadIdx.x;
    /* 对于每一个P的元素, 我们只需要循环遍历width / tile_width 次就okay了, 这里有
    for (int m = 0; m < width / BLOCKSIZE; m ++) {
        /* 这里为了实现bank conflict, 把tx与tx的顺序颠倒, 同时索引也改变了*/
        M_deviceShared[tx][ty] = M_device[x * width + (m * BLOCKSIZE + ty)];
        N_deviceShared[tx][ty] = M_device[(m * BLOCKSIZE + tx)* width + y];

        __syncthreads();

        for (int k = 0; k < BLOCKSIZE; k ++) {
            P_element += M_deviceShared[tx][k] * N_deviceShared[k][ty];
        }
        __syncthreads();
    }

    /* 列优先 */
    P_device[x * width + y] = P_element;
}
```

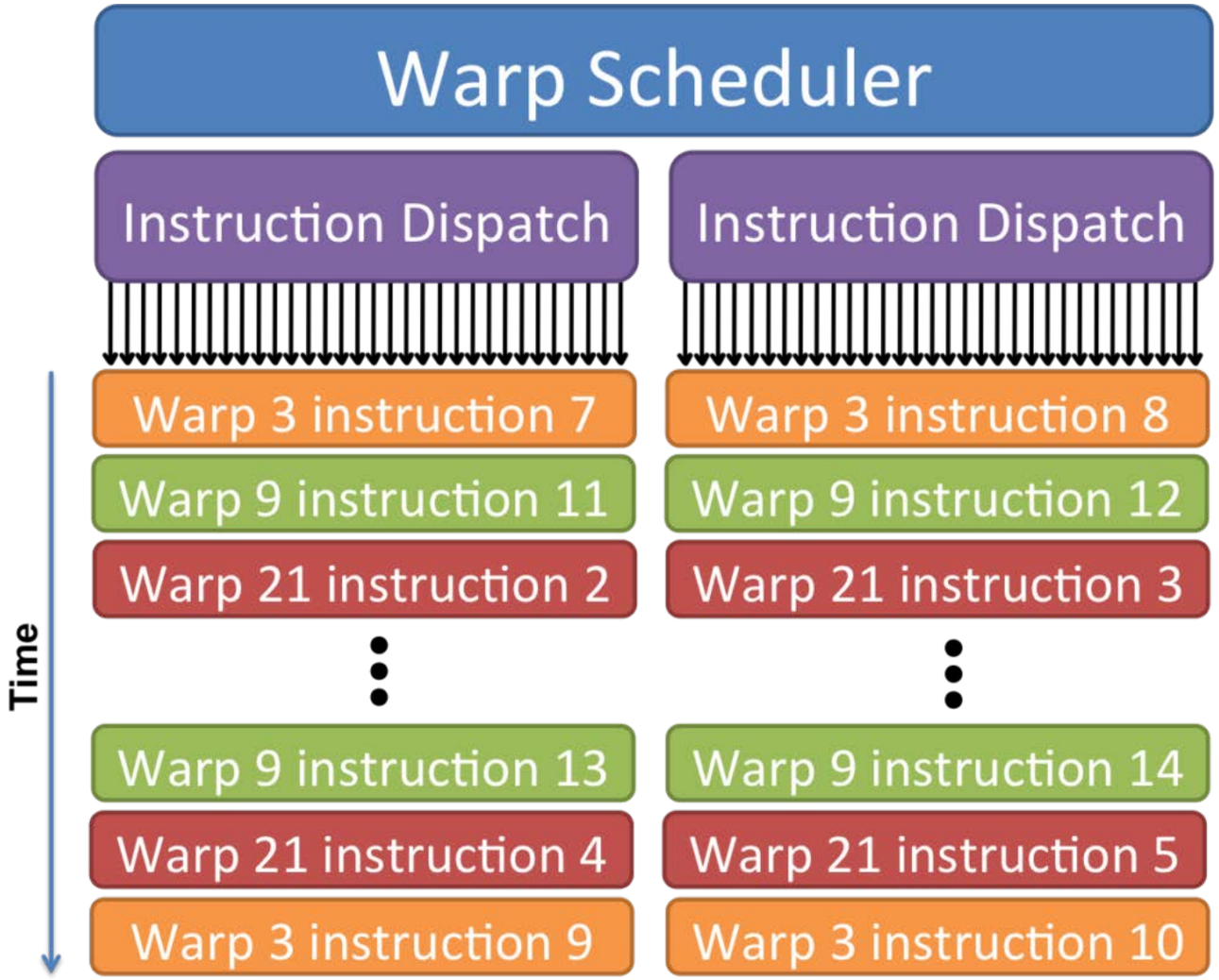
```cuda
__global__ void MatmulSharedDynamicConflictPadKernel(float *M_device, float *N_device, f
    /*
        声明动态共享变量的时候需要加extern, 同时需要是一维的
        注意这里有个坑, 不能够像这样定义:
            __shared__ float M_deviceShared[];
            __shared__ float N_deviceShared[];
        因为在cuda中定义动态共享变量的话, 无论定义多少个他们的地址都是一样的。
        所以如果想要像上面这样使用的话, 需要用两个指针分别指向shared memory的不同位置才行
    */

    extern __shared__ float deviceShared[];
    int stride = (blockSize + 1) * blockSize;
    /*
        对于x和y, 根据blockID, tile大小和threadID进行索引
    */
    int x = blockIdx.x * blockSize + threadIdx.x;
    int y = blockIdx.y * blockSize + threadIdx.y;

    float P_element = 0.0;

    int ty = threadIdx.y;
    int tx = threadIdx.x;
    /* 对于每一个P的元素, 我们只需要循环遍历width / tile_width 次就okay了 */
    for (int m = 0; m < width / blockSize; m ++) {
        /* 这里为了实现bank conflict, 把tx与tx的顺序颠倒, 同时索引也改变了*/
        deviceShared[tx * (blockSize + 1) + ty]            = M_device[x * width + (m *
        deviceShared[stride + (tx * (blockSize + 1) + ty)] = N_device[(m * blockSize +

        __syncthreads();

        for (int k = 0; k < blockSize; k ++) {
            P_element += deviceShared[tx * (blockSize + 1) + k] * deviceShared[stride +
        }
        __syncthreads();
    }

    /* 列优先 */
    P_device[x * width + y] = P_element;
}
```

matmul_gpu_bank_conflict_pad.cu (左:静态, 右: 动态)

(通过在shared memory申请内存时人为的进行padding，从而防止bank conflict发生)

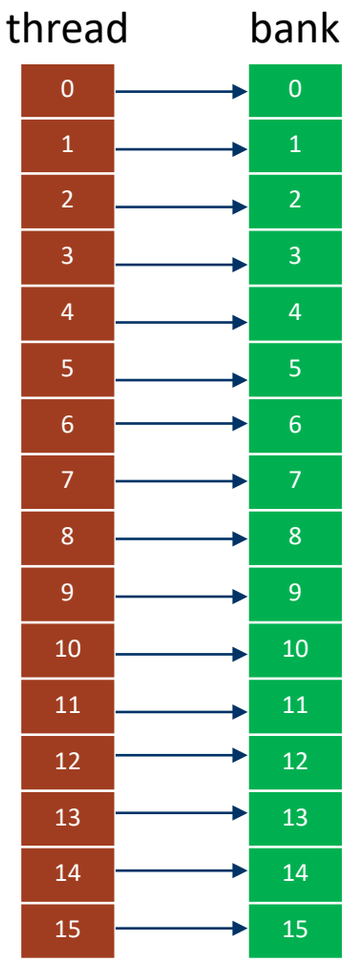
shared memory中存放数据的特殊方式

回顾一下：
在cuda编程中，32个threads组成一个warp，一般程序在执行的时候是以warp为单位去执行，也就是说每32个threads一起执行同一个指令



[1]CUDA – Streaming Multiprocessors

shared memory中存放数据的特殊方式



(*)由于PPT空间有限，
这个图以16个bank为例子

所以，为了能够高效的访存，shared memory中也对应了分成了32个存储体。我们称之为“bank”，分别对应warp中32个线程

bank的宽度，代表的是一个bank所存储的数据的大小宽度。

- 可以是4个字节(32 bit，单精度浮点数)
- 也可以是8个字节(64bit，双精度浮点数)

[1]avoiding bank conflicts in shared memory
[2]Shared Memory Banks and Conflicts

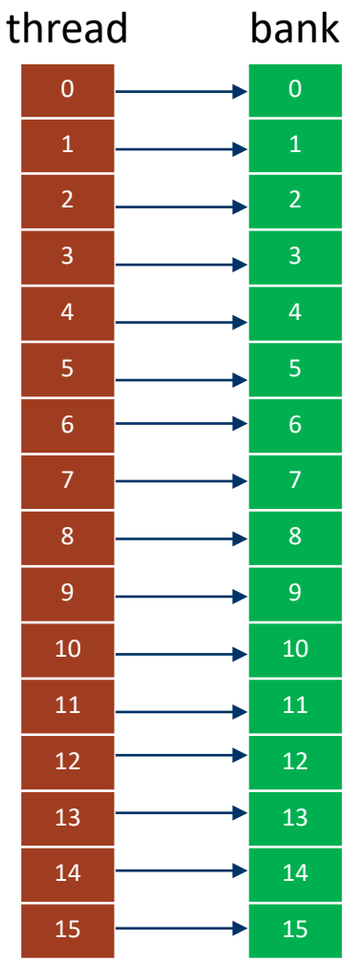
shared memory中存放数据的特殊方式

每31个bank，就会进行一次stride。
比如说bank的宽度是4字节。我们在shared memory中申请了float A[256]大小的空间，那么
A[0], A[1], …, A[31]分别在bank0, bank1, …, bank31中
A[32], A[33], …, A[63]也分在了bank0, bank1, …, bank31中

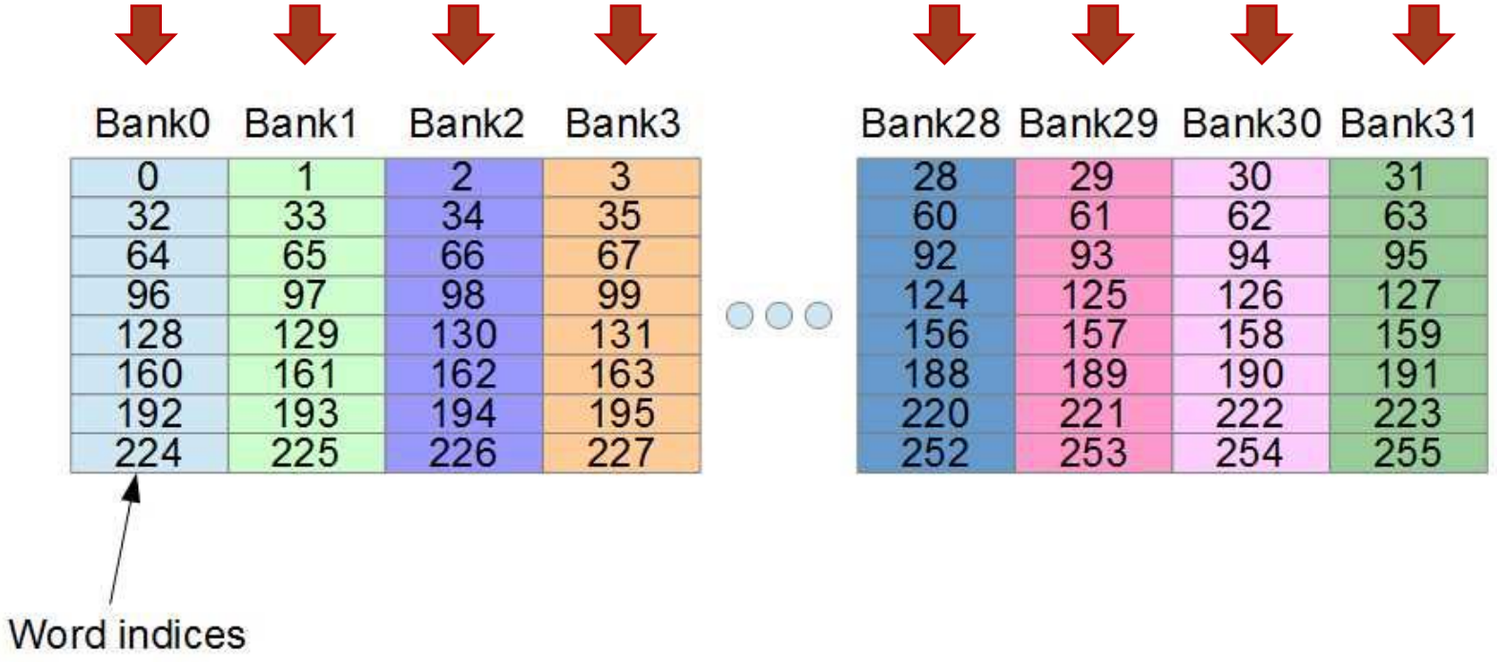
所以A[0], A[32]是共享同一个bank的

[1]avoiding bank conflicts in shared memory
[2]Shared Memory Banks and Conflicts

bank conflict

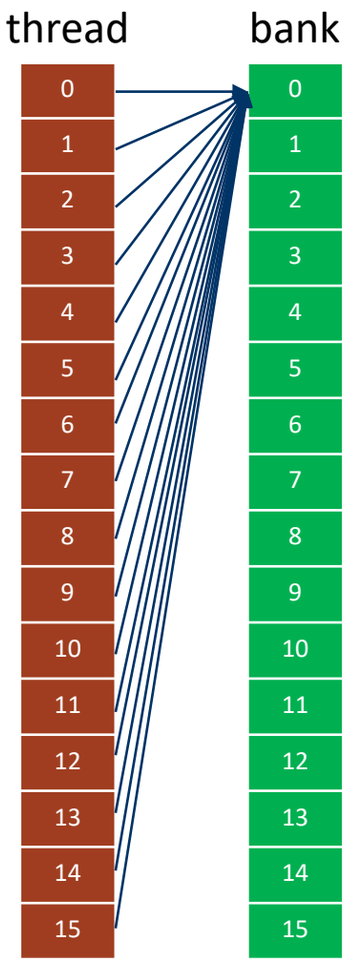


一个很理想的情况就是，32个thread，分别访问shared memory中的32个不同的bank
没有bank conflict，一个memory周期完成所有的memory read/write
(row major/行优先矩阵访问)

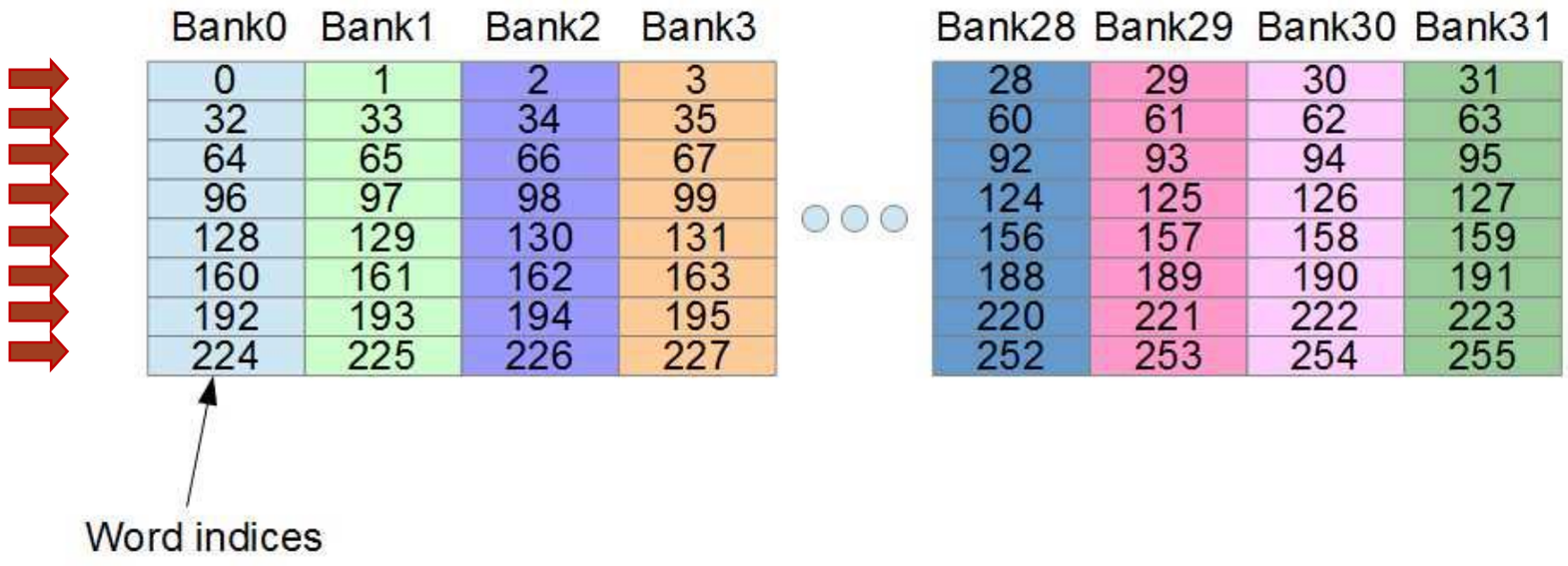


[1]avoiding bankconflicts in shared memory
[2]Shared Memory Banks and Conflicts

bank conflict



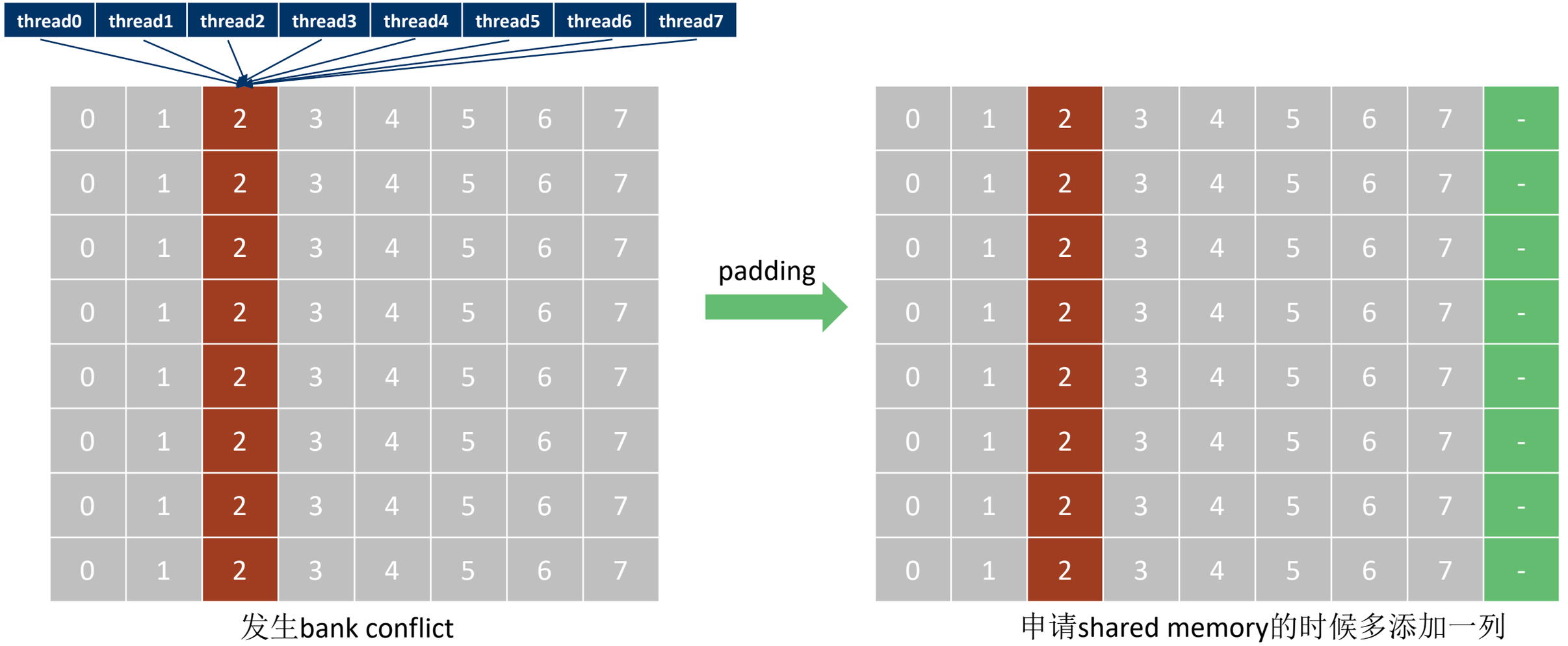
一个最不理想的情况就是，32个thread，访问shared memory中的同一个bank
bank conflict最大化，需要32个memory周期才能完成所有的memory read/write
(column major列优先矩阵访问)



[1]avoiding bankconflicts in shared memory
[2]Shared Memory Banks and Conflicts

使用padding缓解bank conflict

为了方便解释，这里使用了8个bank一次stride进行举例。
在实际CUDA设计时，依然是32个bank一次stride

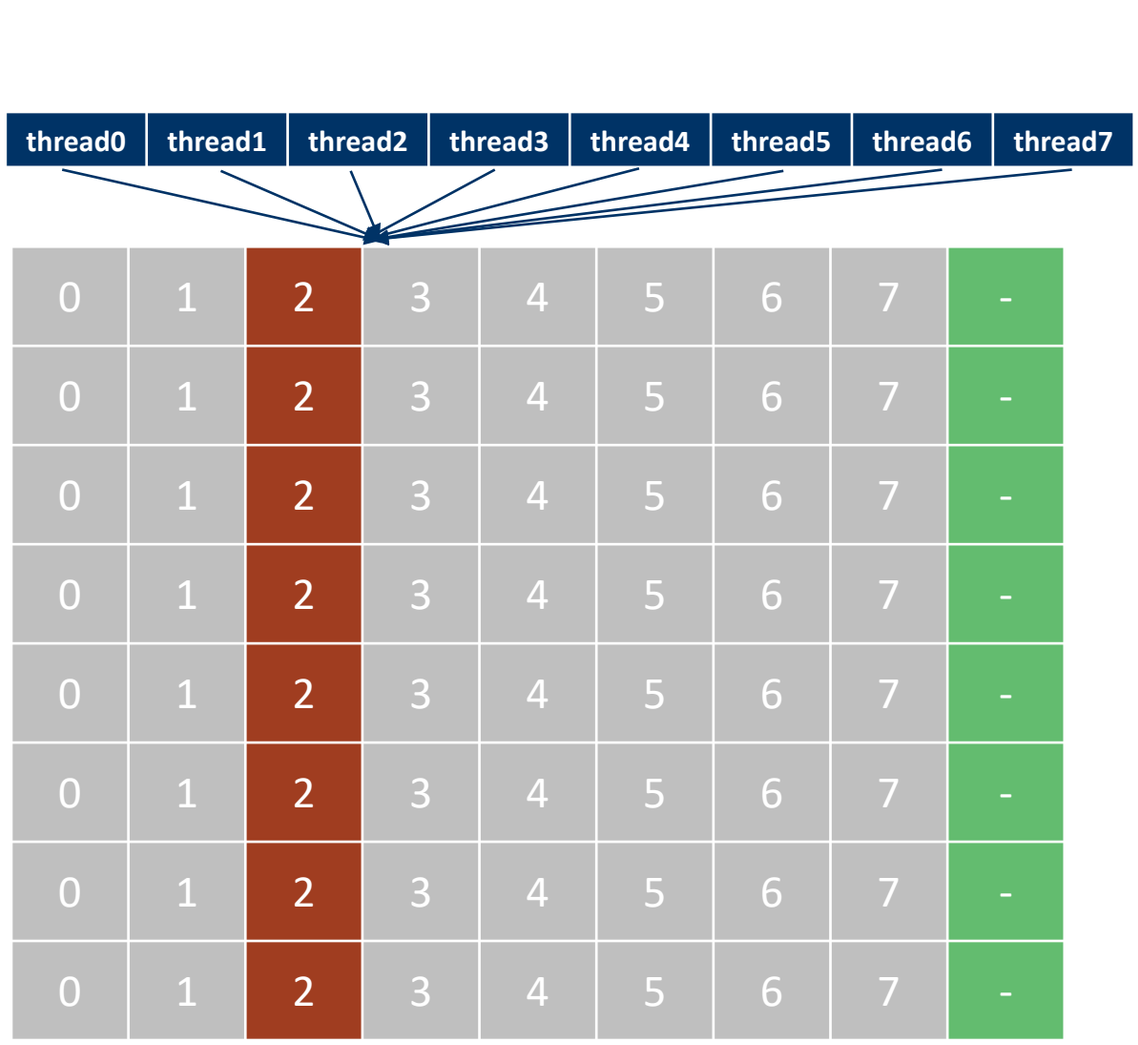


发生bank conflict

申请shared memory的时候多添加一列

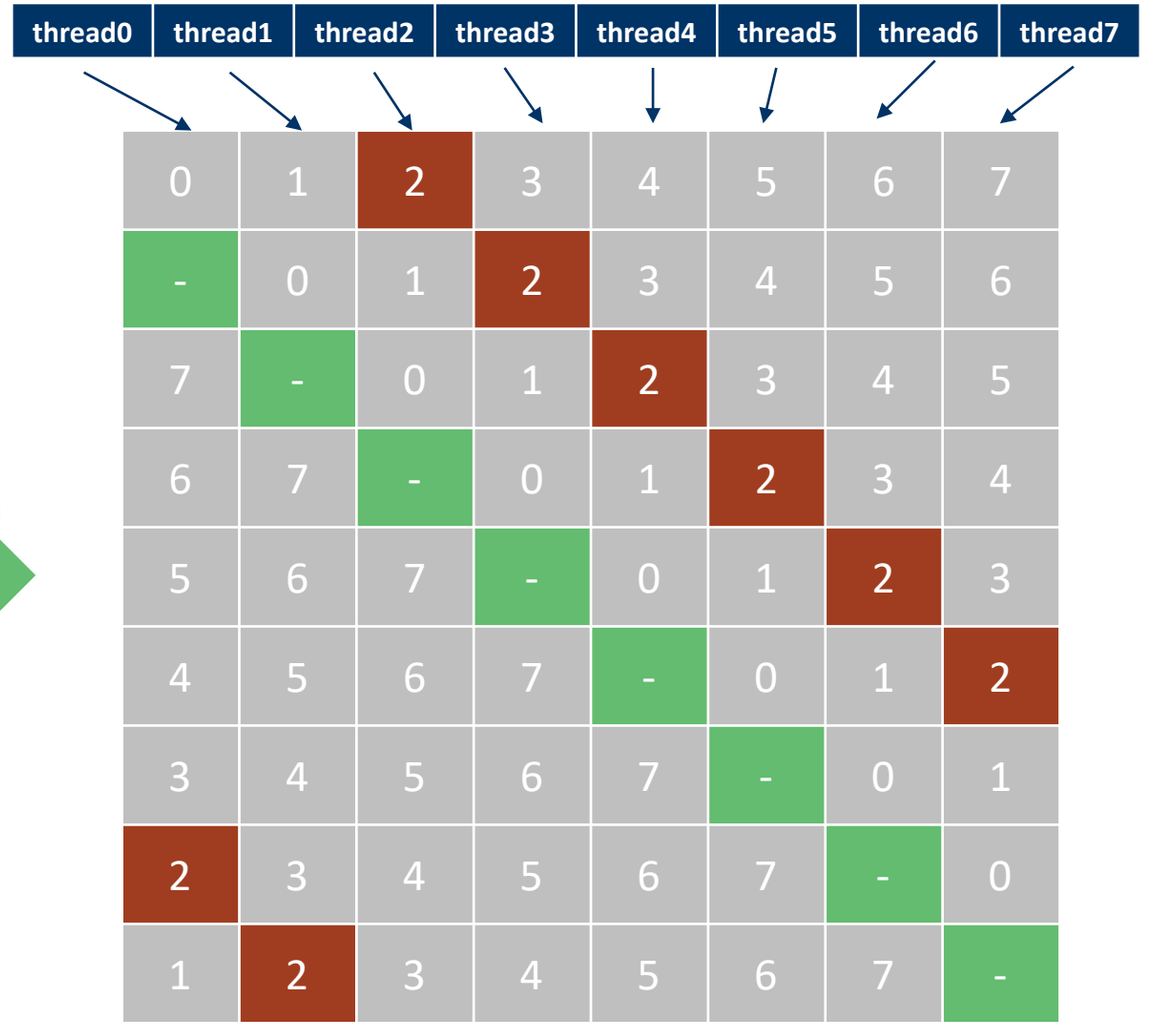
使用padding缓解bank conflict

为了方便解释，这里使用了8个bank一次stride进行举例。
在实际CUDA设计时，依然是32个bank一次stride



发生bank conflict

更改布局



每8个bank进行一次stride，所以最终memory的布局发生改变

Stream and Event

Goal: 理解什么是stream，cuda编程中的显示隐式同步，
以及如何利用多流进行隐藏访存和核函数执行延迟的调度

执行一下我们的第九个CUDA程序


```
2.9-stream-and-event/
├── compile_commands.json
├── config
│   └── Makefile.config
├── Makefile
└── src
    ├── gelu.cu
    ├── gelu.hpp
    ├── main.cpp
    ├── matmul_cpu.cpp
    ├── matmul_gpu_basic.cu
    ├── matmul_gpu_shared.cu
    ├── matmul_gpu_stream.cu
    ├── matmul_gpu_tile.cu
    ├── matmul.hpp
    ├── stream.cu
    ├── stream.hpp
    ├── timer.cpp
    ├── timer.hpp
    ├── utils.cpp
    └── utils.hpp
```

2.9-stream-and-event

```
device supports overlap
Input size is 1048576
sleep <<<(64,64), (16,16)>>>,  1 stream,  1 memcpy,  5 kernel uses 8.762592 ms
sleep <<<(64,64), (16,16)>>>,  5 stream,  1 memcpy,  5 kernel uses 4.012448 ms
```

make之后执行trt-cuda的一部分结果
(这里显示了单流以及多流的实验结果，可以根据block size, grid size进行内部修改)

执行一下我们的第九个CUDA程序

```
// 为了能够体现延迟，这里特意使用clock64()来进行模拟sleep
// 否则如果kernel计算太快，而无法观测到kernel在multi stream中的并发
__global__ void SleepKernel(
    int64_t num_cycles)
{
    int64_t cycles = 0;
    int64_t start = clock64();
    while(cycles < num_cycles) {
        cycles = clock64() - start;
    }
}
```

为了能够平衡mempry以及kernel的执行，这里在kernel内部设定了等待

```
    /* 先把所需要的stream创建出来 */
    cudaStream_t stream[count];
    for (int i = 0; i < count ; i++) {
        CUDA_CHECK(cudaStreamCreate(&stream[i]));
    }

    for (int i = 0; i < count ; i++) {
        for (int j = 0; j < 1; j ++)
            CUDA_CHECK(cudaMemcpyAsync(src_device, src_host, size, cudaMemcpyHostToDevice, stream[i]));
        dim3 dimBlock(blockSize, blockSize);
        dim3 dimGrid(width / blockSize, width / blockSize);

        /* 这里面我们把参数写全了 <<<dimGrid, dimBlock, sMemSize, stream>>> */
        SleepKernel <<<dimGrid, dimBlock, 0, stream[i]>>> (MAX_ITER);
        CUDA_CHECK(cudaMemcpyAsync(src_host, src_device, size, cudaMemcpyDeviceToHost, stream[i]));
    }


    CUDA_CHECK(cudaDeviceSynchronize());


    cudaFree(tar_device);
    cudaFree(src_device);

    for (int i = 0; i < count ; i++) {
        // 使用完了以后不要忘记释放
        cudaStreamDestroy(stream[i]);
    }
```

多流进行异步的计算流程

执行一下我们的第九个CUDA程序

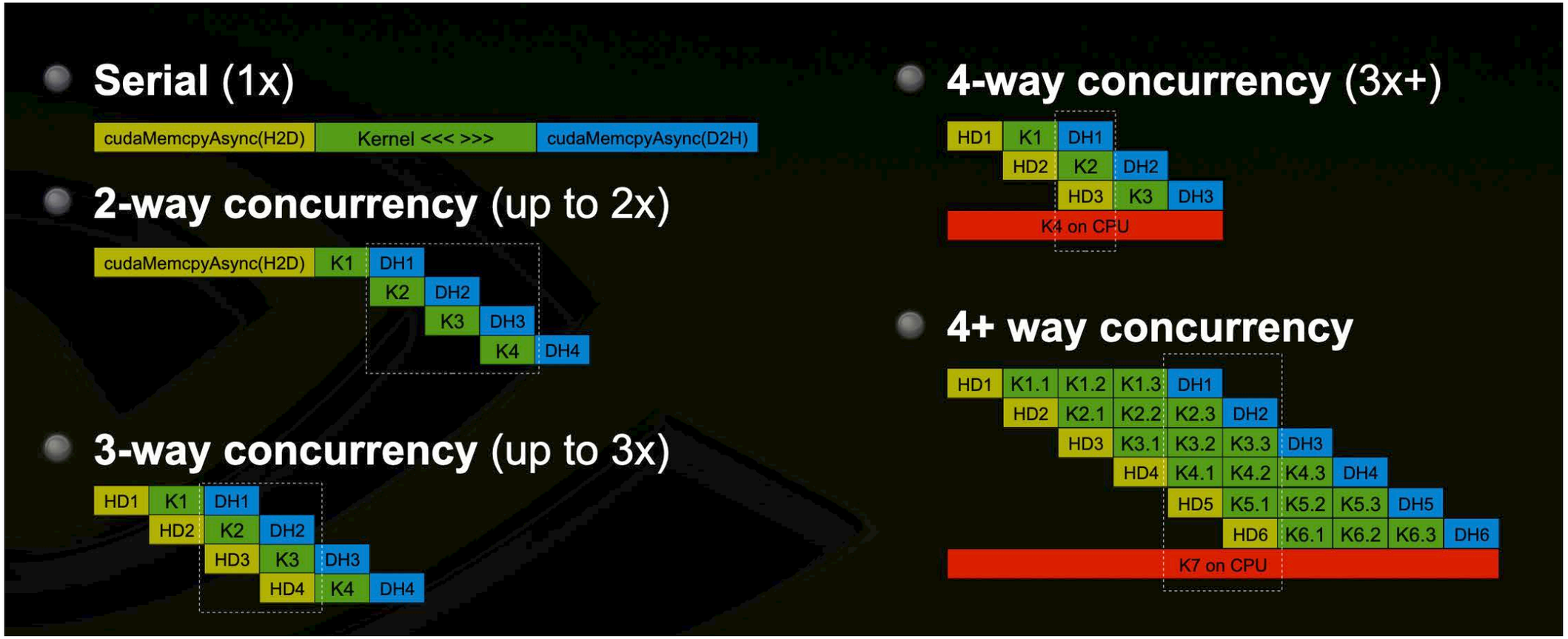
```
/*
    GELU的计算公式(input tensor: X)
    GELU(x)
        = x * Phi(x)
        = x * (0.5 * (1 + tanh[sqrt(2.0 / M_PI) * ( x + 0.44716 * x ^ 3)])

    我们可以提前把这个公式里面的某些值给提前计算出来，以免产生额外的计算，比如：
    - A = 0.5
    - B = sqrt(2.0 / M_PI)
    - C = sqrt(2.0 / M_PI) * 0.44716

    那么这个公式就可以变成
    GELU(x)
        = x * (A + A * tanh(x * (B + C * x * x)))
*/

constexpr float A = 0.5f;
constexpr float B = 0.7978845608028654f;   // sqrt(2.0/M_PI)
constexpr float C = 0.035677408136300125f; // 0.044715 * sqrt(2.0/M_PI)

__global__ void geluKernel(
    const float a, const float b, const float c,
    const float* input, float* output)
{
    const int idx = blockIdx.x * blockDim.x + threadIdx.x;
    const float in  = input[idx];
    const float cdf = a + a * tanh(in * (b + c * in * in));

    output[idx] = in * cdf;
}
```

添加了一个GELU的CUDA实现，为后面课程中搭建Plugin做铺垫

Stream是什么

“A sequence of operation that execute in issue-order in GPU”
同一个流的执行顺序和各个kernel以及mempry operation的启动的顺序是一致的
但是，只要资源没有被占用，不同流之间的执行是可以overlap的。

- PCIe是共享的，所以memcpy只能够在同一个时间执行一个
- SM计算资源是有限的，所以如果计算资源占满了，多流和单流是差不多的



Stream是什么

[1]NVIDIA: Stream and concurrency webinar

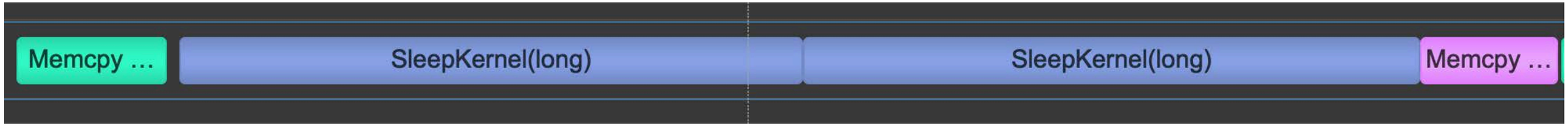
默认流

- 当我们不指定核函数以及memcpy的流式，cuda会使用默认流(default stream)

```
    CUDA_CHECK(cudaMemcpy(src_device, src_host, size, cudaMemcpyHostToDevice));

dim3 dimBlock(blockSize, blockSize);
dim3 dimGrid(width / blockSize, width / blockSize);

SleepKernel <<<dimGrid, dimBlock >>> (MAX_ITER);
CUDA_CHECK(cudaMemcpy(src_host, src_device, size, cudaMemcpyDeviceToHost));
```

HtoD和DtoH是在同一个stream中的不同队列

```
    CUDA_CHECK(cudaMemcpy(src_device, src_host, size, cudaMemcpyHostToDevice));

dim3 dimBlock(blockSize, blockSize);
dim3 dimGrid(width / blockSize, width / blockSize);

SleepKernel <<<dimGrid, dimBlock >>> (MAX_ITER);
SleepKernel <<<dimGrid, dimBlock >>> (MAX_ITER);
CUDA_CHECK(cudaMemcpy(src_host, src_device, size, cudaMemcpyDeviceToHost));
```



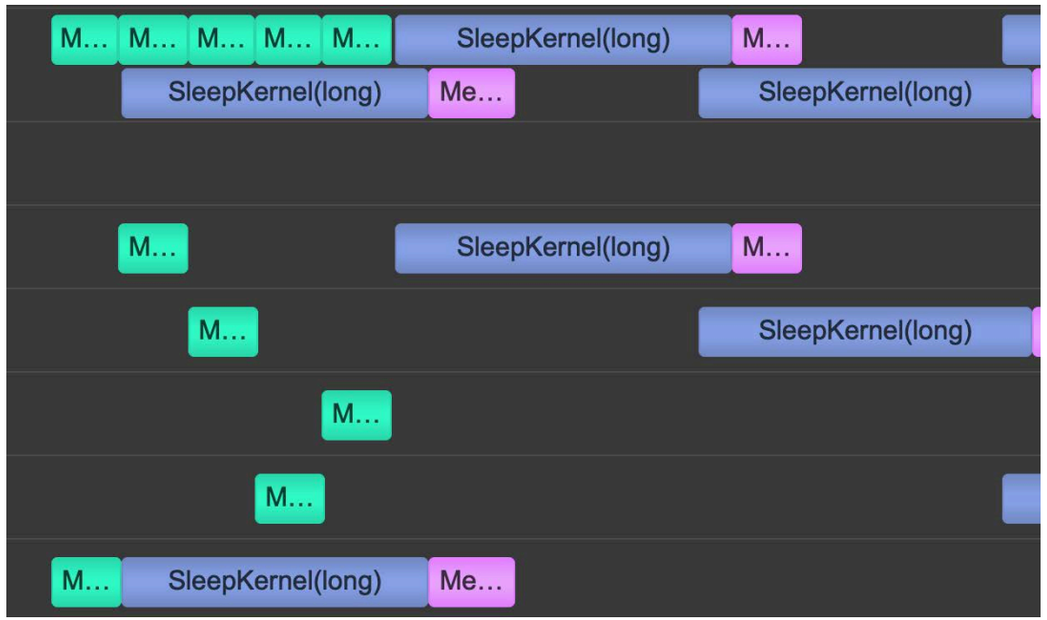
Stream是什么

[1]NVIDIA: Stream and concurrency webinar

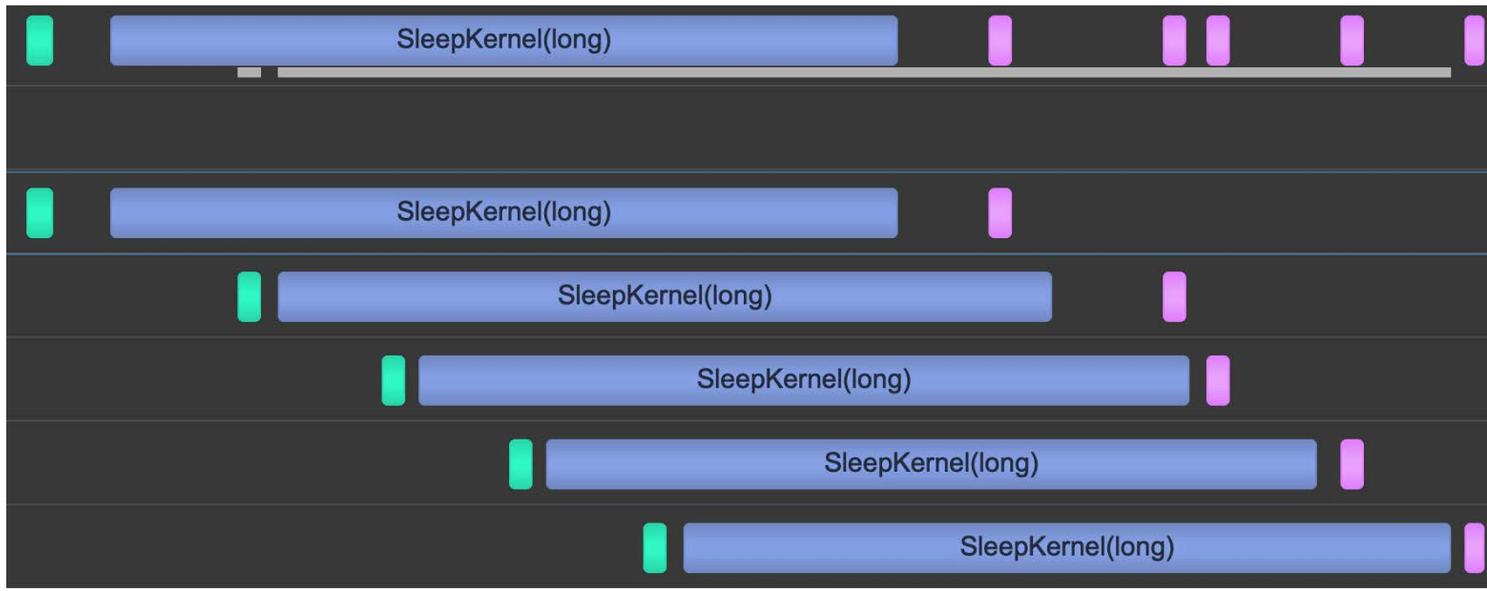
显示的指定流进行操作

- 当我们显式的指定流的时候，核函数是可能overlap

```
    CUDA_CHECK(cudaMemcpyAsync(src_device, src_host, size, cudaMemcpyHostToDevice, stream[i]));
dim3 dimBlock(blockSize, blockSize);
dim3 dimGrid(width / blockSize, width / blockSize);

/* 这里面我们把参数写全了 <<<dimGrid, dimBlock, sMemSize, stream>>> */
SleepKernel <<<dimGrid, dimBlock, 0, stream[i]>>> (MAX_ITER);
CUDA_CHECK(cudaMemcpyAsync(src_host, src_device, size, cudaMemcpyDeviceToHost, stream[i]));
```



计算资源被占满的时候

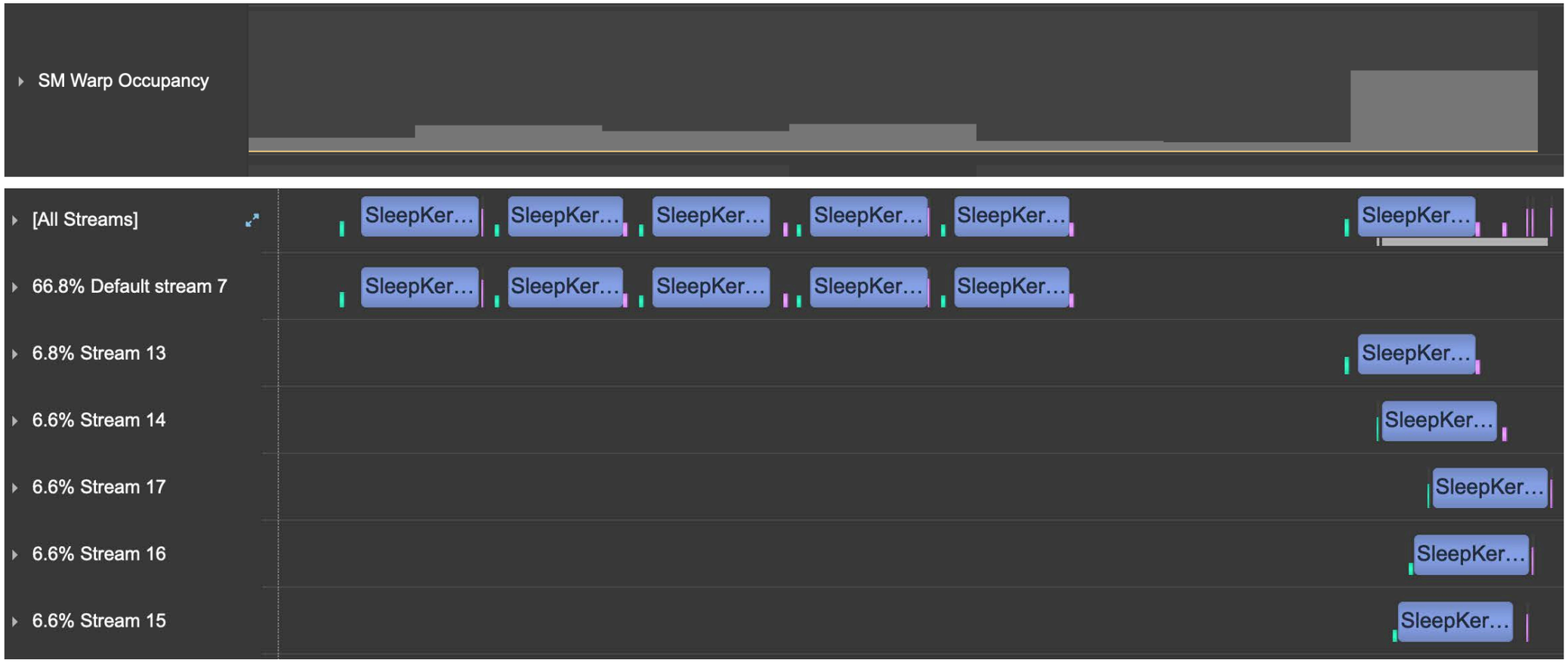


计算资源没有被占满的时候

Streams实验(单流vs多流)

```
device supports overlap
Input size is 256
sleep <<<( 4, 4), ( 4, 4)>>>,  1 stream,  1 memcpy,  5 kernel uses 0.448544 ms
sleep <<<( 4, 4), ( 4, 4)>>>,  5 stream,  1 memcpy,  5 kernel uses 0.193248 ms
```

5 steams: 2.32x speedup



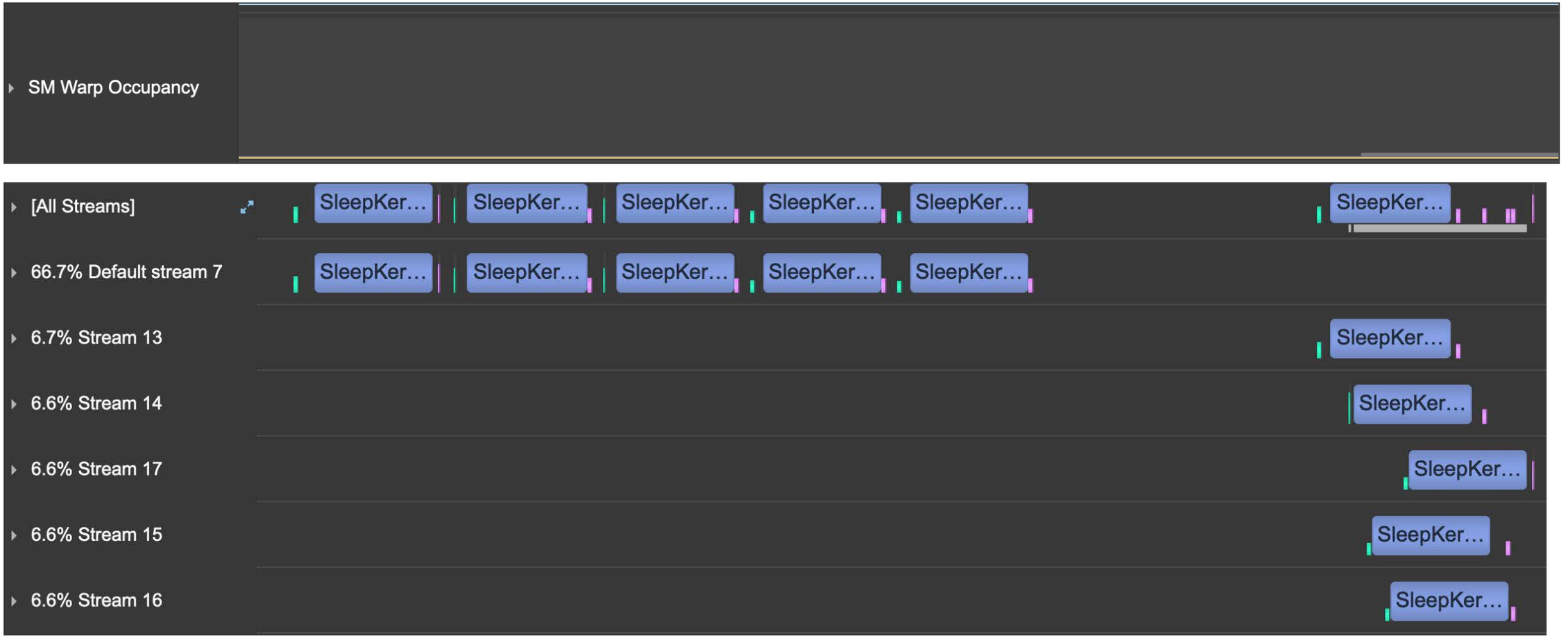
使用nsight进行单流多流的分析(左:单流, 右: 多流)

(这个时候grid, block都很小，GPU上有空余的资源，所以多流有效)

Streams实验(单流vs多流)

```
device supports overlap
Input size is 256
sleep <<<( 1, 1), (16,16)>>>,  1 stream,  1 memcpy,  5 kernel uses 0.441536 ms
sleep <<<( 1, 1), (16,16)>>>,  5 stream,  1 memcpy,  5 kernel uses 0.195968 ms
```

5 steams: 2.25x speedup



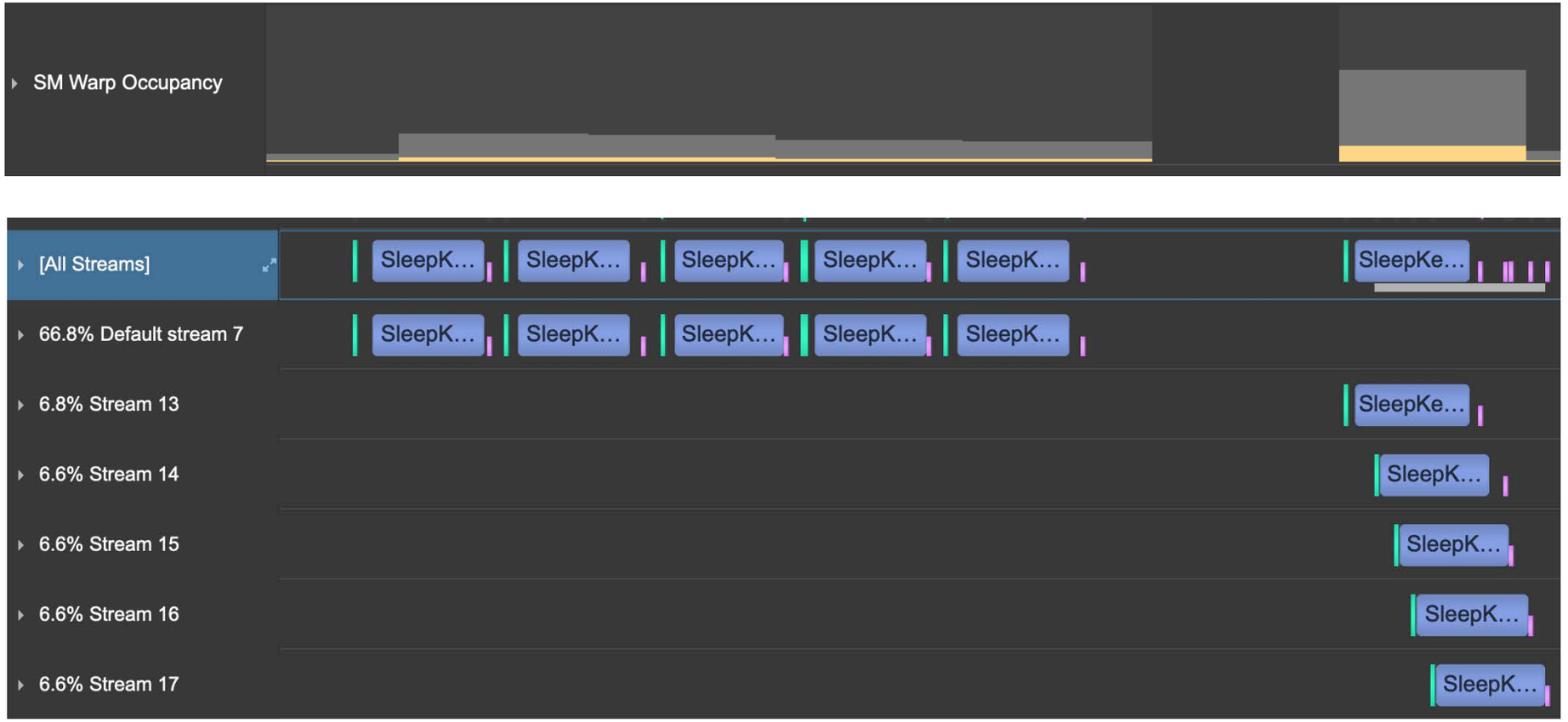
使用nsight进行单流多流的分析(左:单流, 右: 多流)

(同理，扩大了block size可以让warp的填充率提升，但计算资源还是有很多，所以速度跟刚才是一样的)

Streams实验(单流vs多流)

```
device supports overlap
Input size is 4096
sleep <<<( 4, 4), (16,16)>>>,  1 stream,  1 memcpy,  5 kernel uses 0.525696 ms
sleep <<<( 4, 4), (16,16)>>>,  5 stream,  1 memcpy,  5 kernel uses 0.212704 ms
```

5 steams: 2.47x speedup



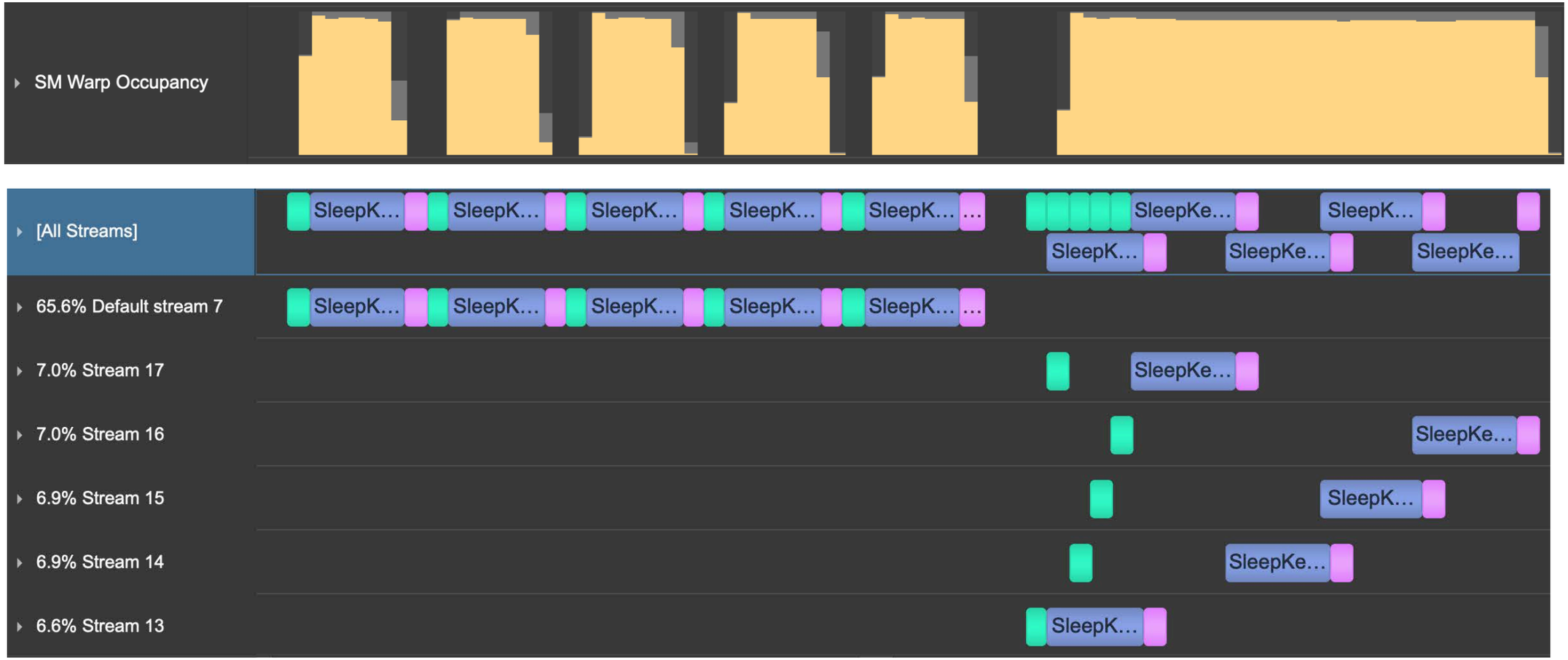
使用nsight进行单流多流的分析(左:单流, 右: 多流)

(同理，扩大了block size可以让warp的填充率提升，资源占有率继续提升)

Streams实验(单流vs多流)

```
device supports overlap
Input size is 1048576
sleep <<<(64,64), (16,16)>>>,  1 stream,  1 memcpy,  5 kernel uses 5.352544 ms
sleep <<<(64,64), (16,16)>>>,  5 stream,  1 memcpy,  5 kernel uses 4.002528 ms
```

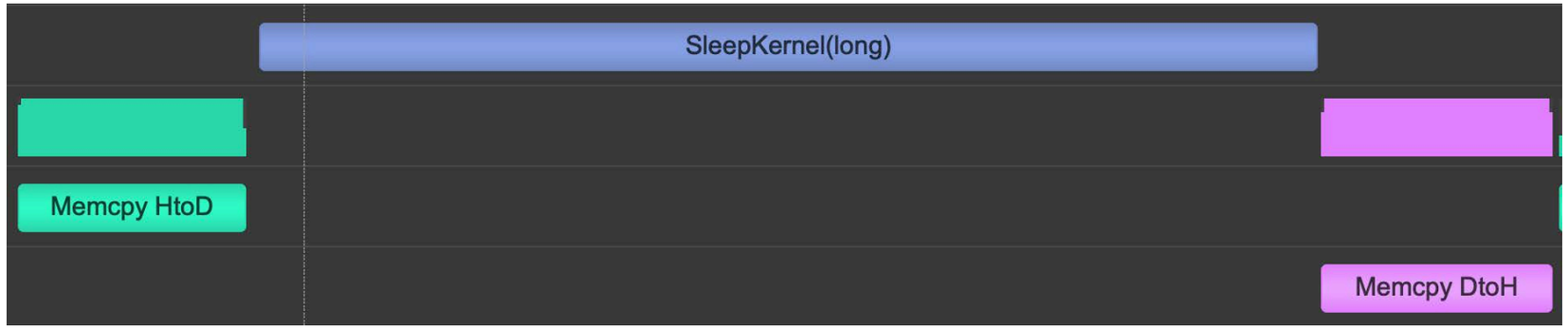
5 steams: 1.33x speedup



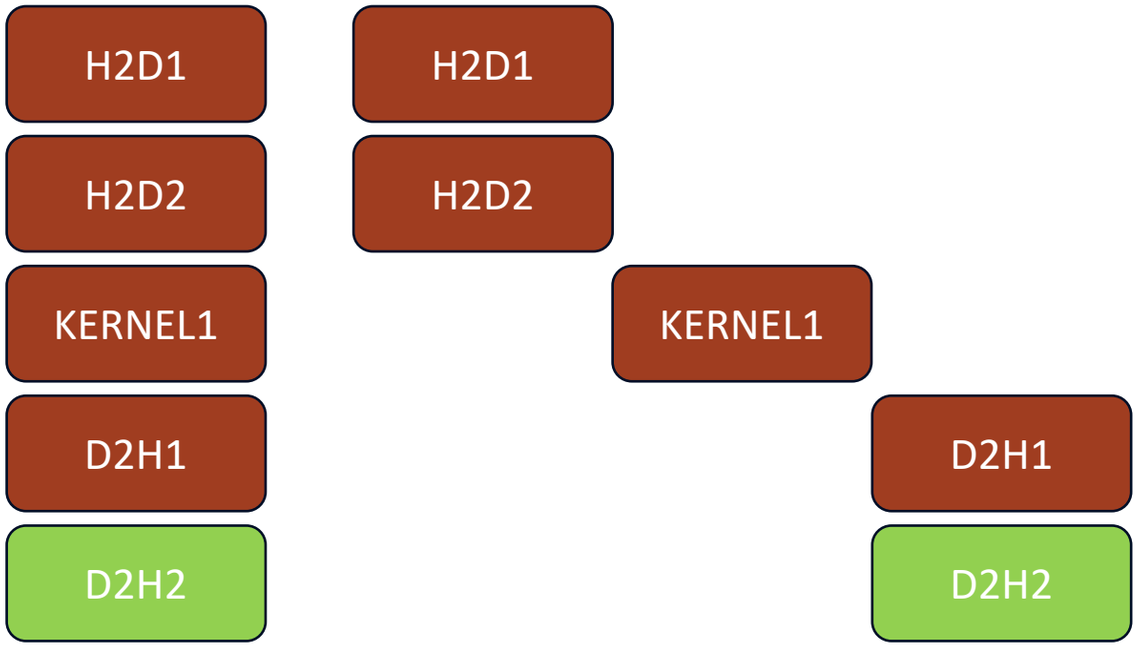
使用nsight进行单流多流的分析(左:单流, 右: 多流)

(当计算资源已经占满的时候，核函数不能够在overlapping了，只能够一个一个的等)

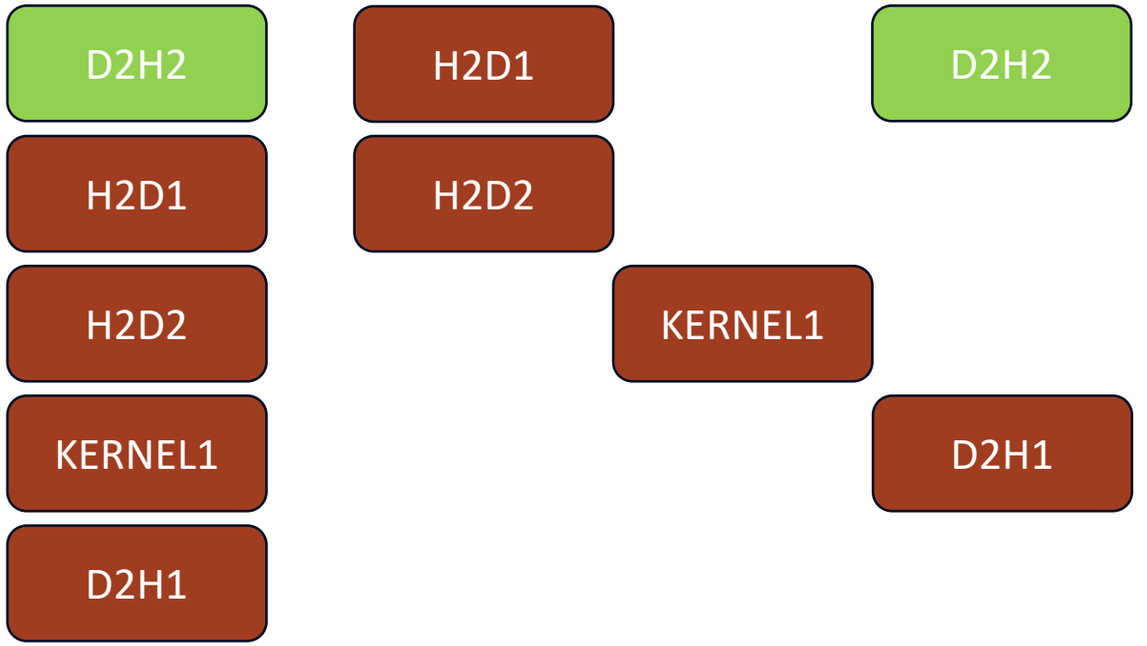
如何隐藏延迟(memory)



Memcpy HtoD队列
Memcpy DtoH队列

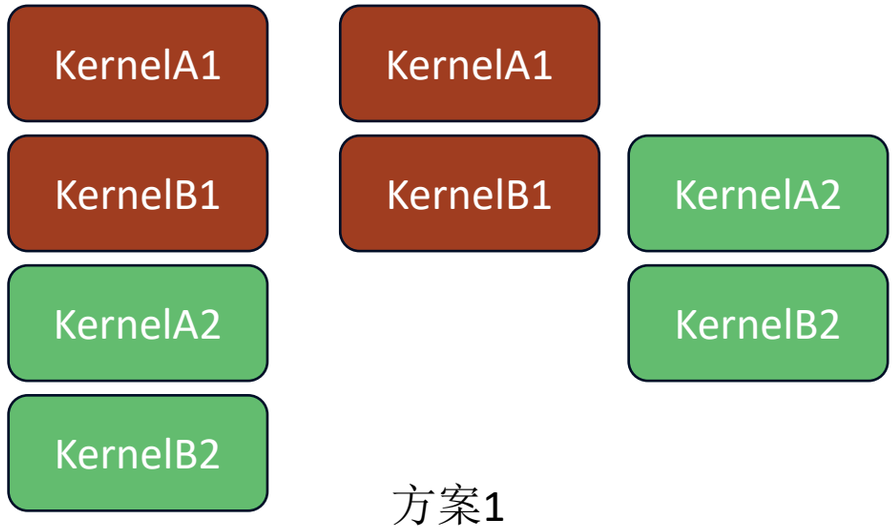


方案1
(按照正常的方式去调度)

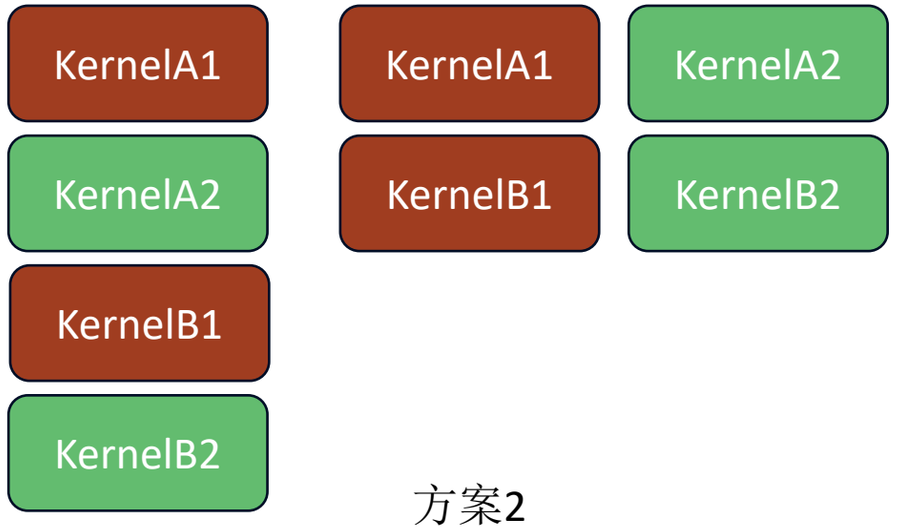


方案1
(把D2H2放在最一开始issue，可以减少
D2H Queue的等待时间)

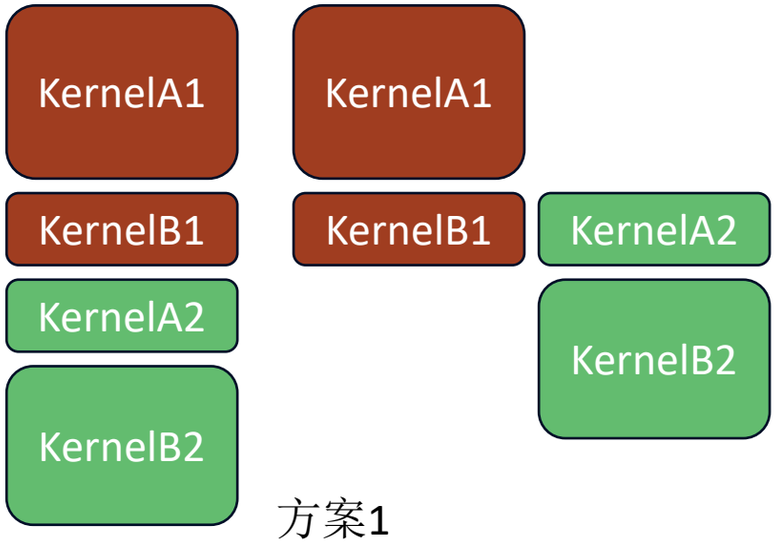
如何隐藏延迟(kernel)



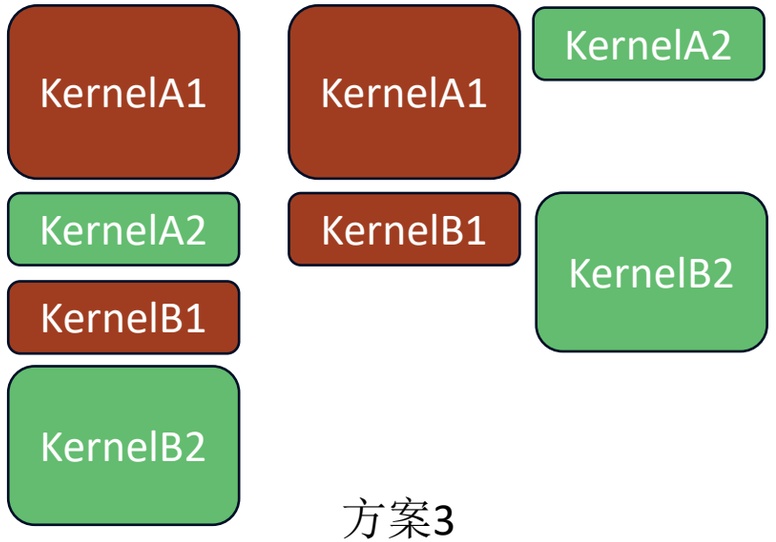
方案1
(按照正常的方式去调度)



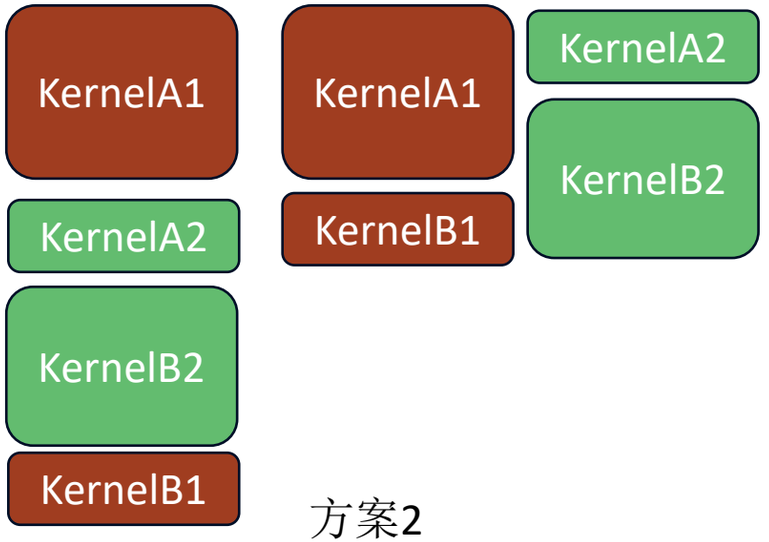
方案2
(在KernelA1执行等待时launchKernelA2)



方案1
(按照正常的方式去调度)



方案3
(在KernelA1执行等待的时候启动A2)

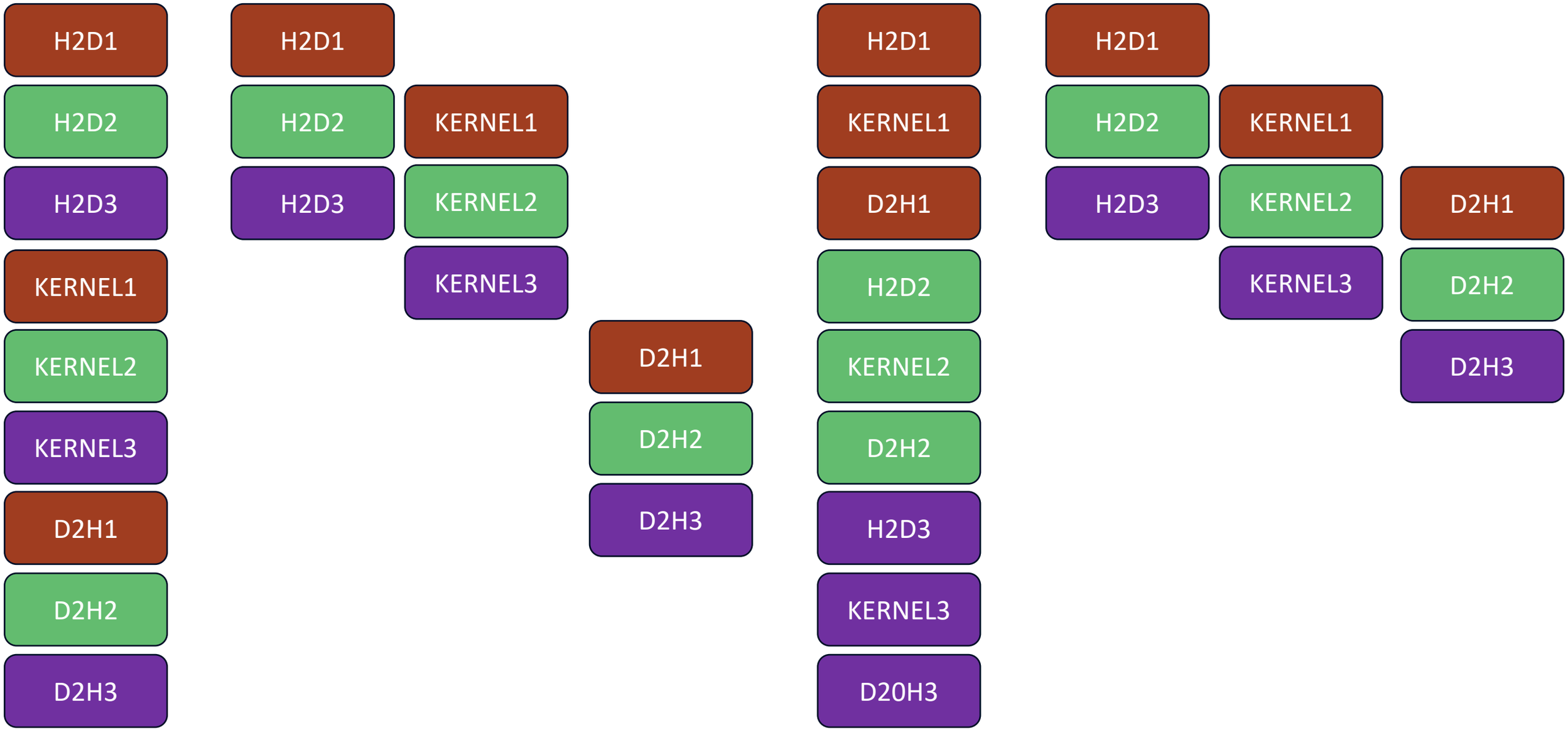


方案2
(A2先执行完发现可以启动B2)

如何隐藏延迟(kernel + memory)

哪个好？

如何隐藏延迟(kernel + memory)



05

bilinear interpolation 的cuda实现

Goal: 理解如何使用cuda进行opencv的图像处理的加速，
理解双线性插值进行图像大小调整的算法流程

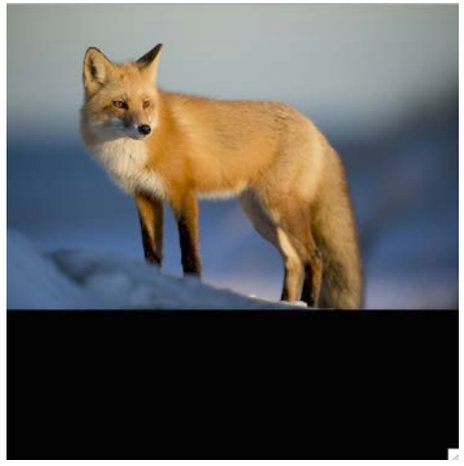
执行一下我们的第十个CUDA程序

```
2.10-bilinear-interpolation
├── compile_commands.json
├── config
│   └── Makefile.config
├── data
│   ├── cat.png
│   ├── deer.png
│   ├── eagle.png
│   ├── fox.png
│   ├── tiny-cat.png
│   ├── unknown.png
│   └── wolf.png
├── include
│   ├── preprocess.hpp
│   ├── timer.hpp
│   └── utils.hpp
├── Makefile
└── src
    ├── main.cpp
    ├── preprocess.cpp
    ├── preprocess.cu
    ├── timer.cpp
    └── utils.cpp
```

2.10-bilinear-interpolation

fox.png

fox_resized_blinear_gpu.png



fox_resized_blinear_letterbox_gpu.png

fox_resized_blinear_letterbox_center_gpu.png

```
Resize(bilinear) in cpu takes:                          uses 1.185867 ms
Resize(nearest) in gpu takes:                           uses 0.021600 ms
Resize(bilinear) in gpu takes:                          uses 0.037280 ms
Resize(bilinear-letterbox) in gpu takes:                uses 0.018336 ms
Resize(bilinear-letterbox-center) in gpu takes:         uses 0.018624 ms
```

make之后执行trt-cuda的一部分结果
(这里显示了使用shared memory中没有bank conflict和有bank conflict时的速度差异)

执行一下我们的第八个CUDA程序

```cpp
// 根据比例进行缩放 (CPU版本)
cv::Mat preprocess_cpu(cv::Mat &src, const int &tar_h, const int &tar_w, Timer timer, i
    cv::Mat tar;

    int height  = src.rows;
    int width   = src.cols;
    float dim   = std::max(height, width);
    int resizeH = ((height / dim) * tar_h);
    int resizeW = ((width / dim) * tar_w);

    int xOffSet = (tar_w - resizeW) / 2;
    int yOffSet = (tar_h - resizeH) / 2;

    resizeW    = tar_w;
    resizeH    = tar_h;

    timer.start_cpu();

    /*BGR2RGB*/
    cv::cvtColor(src, src, cv::COLOR_BGR2RGB);

    /*Resize*/
    cv::resize(src, tar, cv::Size(resizeW, resizeH), 0, 0, cv::INTER_LINEAR);



    timer.stop_cpu();
    timer.duration_cpu<Timer::ms>("Resize(bilinear) in cpu takes:");

    return tar;
}
```

preprocess.cpp
(bilinear resize的opencv实现)

```cpp
// 根据比例进行缩放 (GPU版本)
cv::Mat preprocess_gpu(
    cv::Mat &h_src, const int& tar_h, const int& tar_w, Timer timer, int tactis)
{
    uint8_t* d_tar = nullptr;
    uint8_t* d_src = nullptr;

    cv::Mat h_tar(cv::Size(tar_w, tar_h), CV_8UC3);

    int height   = h_src.rows;
    int width    = h_src.cols;
    int chan     = 3;

    int src_size  = height * width * chan;
    int tar_size  = tar_h * tar_w * chan;

    // 分配device上的src和tar的内存
    CUDA_CHECK(cudaMalloc(&d_src, src_size));
    CUDA_CHECK(cudaMalloc(&d_tar, tar_size));

    // 将数据拷贝到device上
    CUDA_CHECK(cudaMemcpy(d_src, h_src.data, src_size, cudaMemcpyHostToDevice));

    timer.start_gpu();

    // device上处理resize, BGR2RGB的核函数
    resize_bilinear_gpu(d_tar, d_src, tar_w, tar_h, width, height, tactis);

    // host和device进行同步处理
    CUDA_CHECK(cudaDeviceSynchronize());

    timer.stop_gpu();
    switch (tactis) {
        case 0: timer.duration_gpu("Resize(nearest) in gpu takes:"); break;
        case 1: timer.duration_gpu("Resize(bilinear) in gpu takes:"); break;
        case 2: timer.duration_gpu("Resize(bilinear-letterbox) in gpu takes:"); break
        case 3: timer.duration_gpu("Resize(bilinear-letterbox-center) in gpu takes:")
        default: break;
    }

    // 将结果返回给host上
    CUDA_CHECK(cudaMemcpy(h_tar.data, d_tar, tar_size, cudaMemcpyDeviceToHost));


    CUDA_CHECK(cudaFree(d_src));
    CUDA_CHECK(cudaFree(d_tar));

    return h_tar;
}
```

preprocess.cpp
(bilinear resize的cuda实现(接口部分))

Bilinear interpolation

中文叫做”双线性插值”，是一种对图像进行缩放/放大的一种计算方法。opencv默认的resize方式

src_width

src
_height

1000 x 600

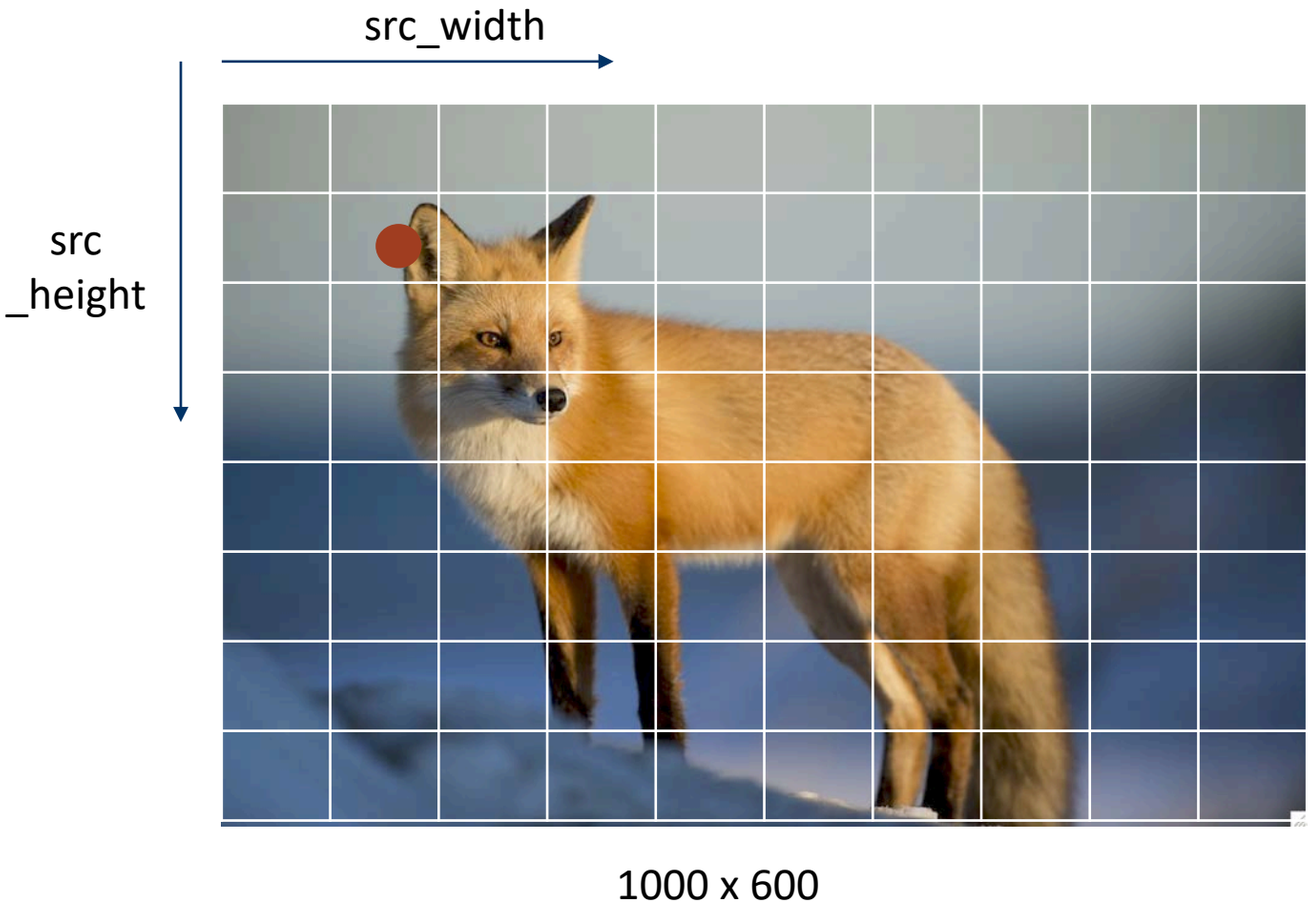
tar_width

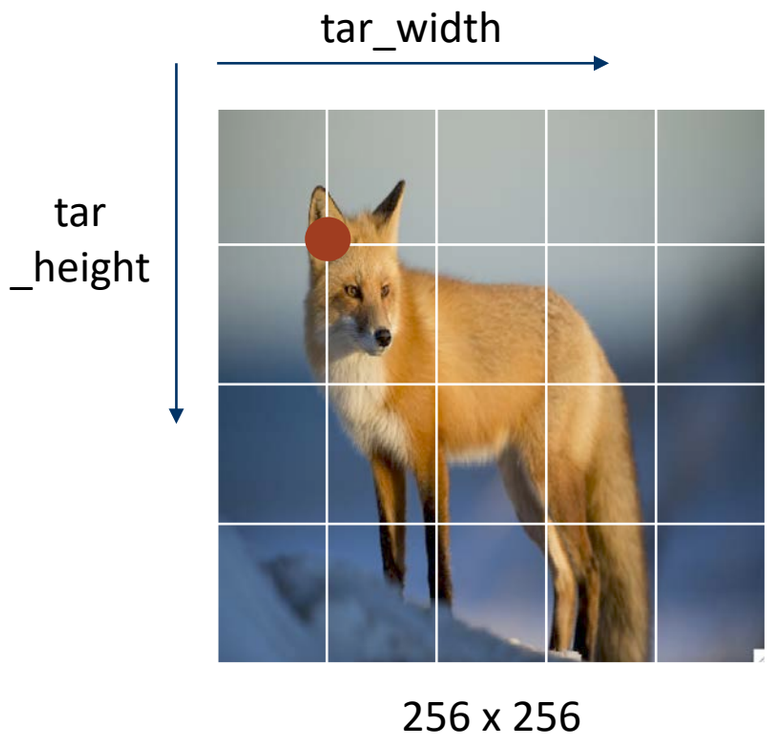
tar
_height

256 x 256

Bilinear interpolation

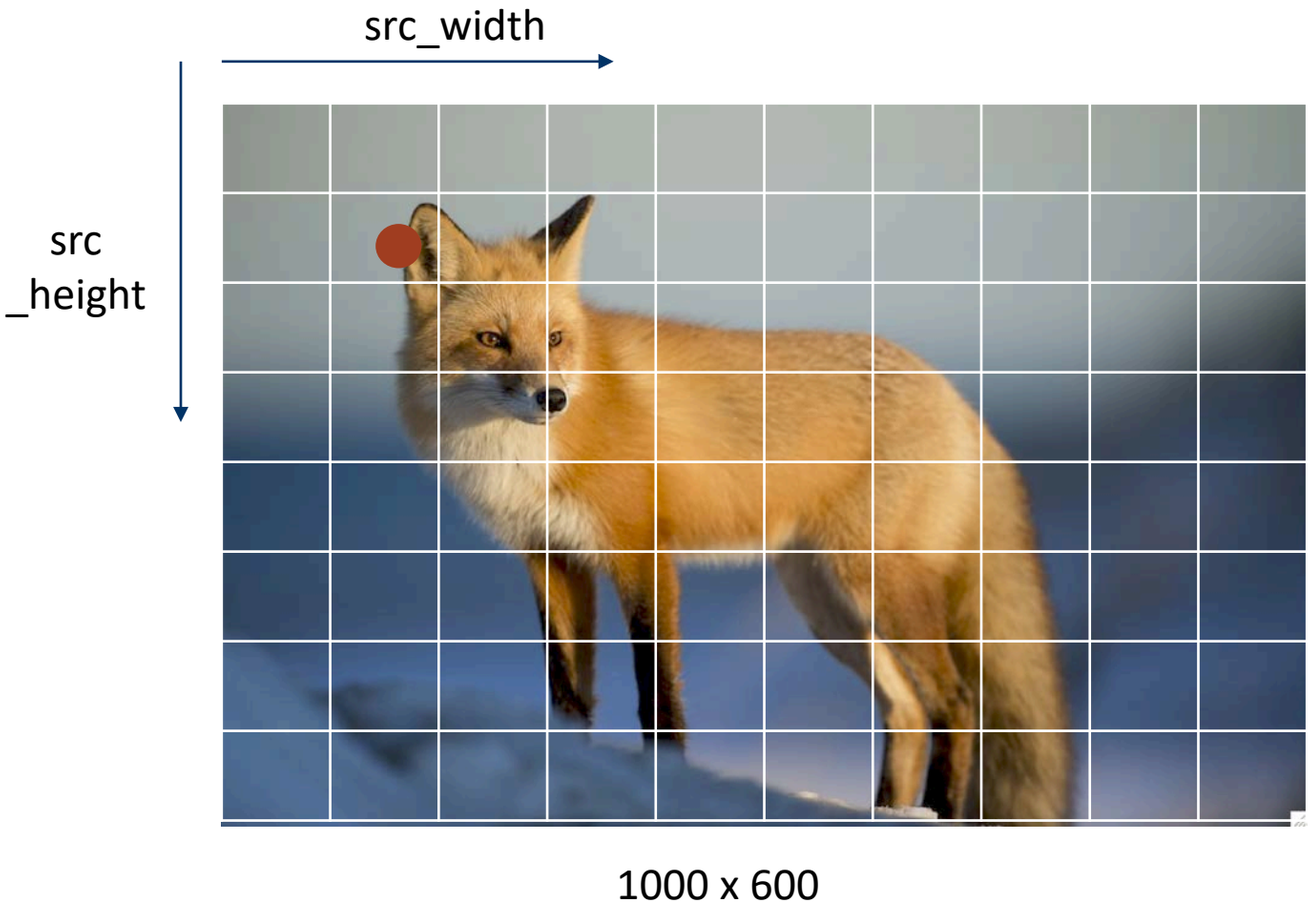
中文叫做”双线性插值”，是一种对图像进行缩放/放大的一种计算方法。opencv默认的resize方式





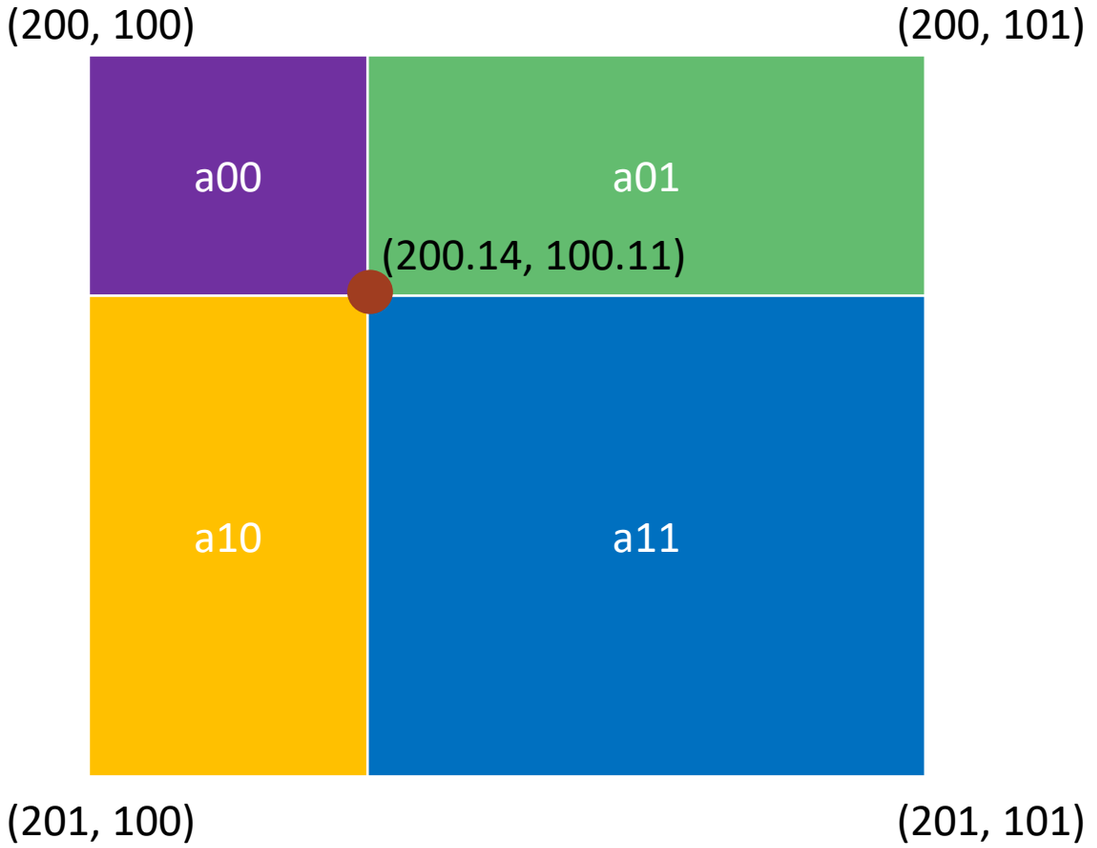
Bilinear interpolation

中文叫做”双线性插值”，是一种对图像进行缩放/放大的一种计算方法。opencv默认的resize方式



Bilinear interpolation

中文叫做”双线性插值”，是一种对图像进行缩放/放大的一种计算方法。
opencv默认的resize方式




```
// bilinear interpolation -- 实现bilinear interpolation
tar[tarIdx + 0] = round(
                  a1_1 * src[srcIdx1_1 + 0] +
                  a1_2 * src[srcIdx1_2 + 0] +
                  a2_1 * src[srcIdx2_1 + 0] +
                  a2_2 * src[srcIdx2_2 + 0]);

tar[tarIdx + 1] = round(
                  a1_1 * src[srcIdx1_1 + 1] +
                  a1_2 * src[srcIdx1_2 + 1] +
                  a2_1 * src[srcIdx2_1 + 1] +
                  a2_2 * src[srcIdx2_2 + 1]);

tar[tarIdx + 2] = round(
                  a1_1 * src[srcIdx1_1 + 2] +
                  a1_2 * src[srcIdx1_2 + 2] +
                  a2_1 * src[srcIdx2_1 + 2] +
                  a2_2 * src[srcIdx2_2 + 2]);
```

(62, 70)

$$
\begin{aligned}
&tar[70][63] \\
&= \\
&src[100][200] * a11 + \\
&src[101][200] * a10 + \\
&src[100][201] * a01 + \\
&src[101][201] * a00
\end{aligned}
$$

letter box

$$\max\left(\frac{src\_width}{tar\_width}, \frac{src\_height}{tar\_height}\right)$$

如果我们想要缩放后依然保持比例的话，我们对宽和高的缩放比保持一致就可以了

$$\min\left(\frac{tar\_width}{src\_width}, \frac{tar\_height}{src\_height}\right)$$

```
//scaled resize
float scaled_h = (float)srcH / tarH;
float scaled_w = (float)srcW / tarW;
float scale = (scaled_h > scaled_w ? scaled_h : scaled_w);
```

src_width

src
_height

1000 x 600

tar_width

tar
_height

256 x 256

letter box(center)

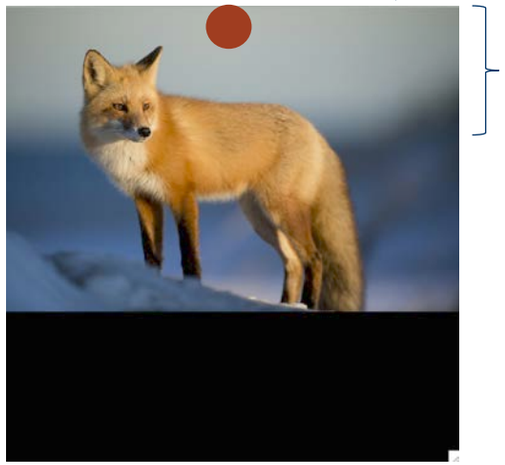
一般来说，在进行缩放以后我们希望让图像居中，所以需要让图像的中心坐标shift一定的像素

```
// bilinear interpolation -- 计算原图在目标图中的x, y方向上的偏移量
y = y - int(srcH / (scaled_h * 2)) + int(tarH / 2);
x = x - int(srcW / (scaled_w * 2)) + int(tarW / 2);
```

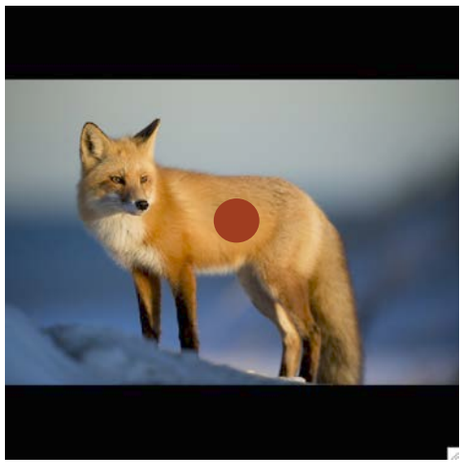
tar_width

tar
_height

256 x 256



256 x 256

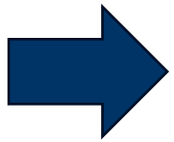


256 x 256

BGR2RGB

opencv读取完图片以后，默认的格式是BGR的。如果要将读取的图片传给DNN进行推理的话，我们需要将channel的方向改变一下。在CUDA中可以这么更改:

```
// bilinear interpolation -- 实现bilinear inter
tar[tarIdx + 0] = round(
                  a1_1 * src[srcIdx1_1 + 0] +
                  a1_2 * src[srcIdx1_2 + 0] +
                  a2_1 * src[srcIdx2_1 + 0] +
                  a2_2 * src[srcIdx2_2 + 0]);

tar[tarIdx + 1] = round(
                  a1_1 * src[srcIdx1_1 + 1] +
                  a1_2 * src[srcIdx1_2 + 1] +
                  a2_1 * src[srcIdx2_1 + 1] +
                  a2_2 * src[srcIdx2_2 + 1]);

tar[tarIdx + 2] = round(
                  a1_1 * src[srcIdx1_1 + 2] +
                  a1_2 * src[srcIdx1_2 + 2] +
                  a2_1 * src[srcIdx2_1 + 2] +
                  a2_2 * src[srcIdx2_2 + 2]);
```



```
// bilinear interpolation -- 实现bilinear interp
tar[tarIdx + 0] = round(
                  a1_1 * src[srcIdx1_1 + 2] +
                  a1_2 * src[srcIdx1_2 + 2] +
                  a2_1 * src[srcIdx2_1 + 2] +
                  a2_2 * src[srcIdx2_2 + 2]);

tar[tarIdx + 1] = round(
                  a1_1 * src[srcIdx1_1 + 1] +
                  a1_2 * src[srcIdx1_2 + 1] +
                  a2_1 * src[srcIdx2_1 + 1] +
                  a2_2 * src[srcIdx2_2 + 1]);

tar[tarIdx + 2] = round(
                  a1_1 * src[srcIdx1_1 + 0] +
                  a1_2 * src[srcIdx1_2 + 0] +
                  a2_1 * src[srcIdx2_1 + 0] +
                  a2_2 * src[srcIdx2_2 + 0]);
```

其实，如果需要的话，我们也可以在BGR2RGB的同时，实现BHWC2BCHW的转变。类似于PyTorch中的transpose

最终的实现效果

```cpp
// resized之后的图tar上的坐标
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;

// bilinear interpolation -- 计算x,y映射到原图时最近的4个坐标
int src_y1 = round((float)y * scaled_h);
int src_x1 = round((float)x * scaled_w);
int src_y2 = src_y1 + 1;
int src_x2 = src_x1 + 1;

if (src_y1 < 0 || src_x1 < 0 || src_y1 > srcH || src_x1 > srcW) {
    // bilinear interpolation -- 对于越界的坐标不进行计算
} else {
    // bilinear interpolation -- 计算原图上的坐标(浮点类型)在0~1之间的值
    float th    = (float)y * scaled_h - src_y1;
    float tw    = (float)x * scaled_w - src_x1;

    // bilinear interpolation -- 计算面积(这里建议自己手画一张图来理解一下)
    float a1_1 = (1.0 - tw) * (1.0 - th);
    float a1_2 = (1.0 - tw) * th;
    float a2_1 = tw * (1.0 - th);
    float a2_2 = tw * th;

    // bilinear interpolation -- 计算4个坐标所对应的索引
    int srcIdx1_1 = (src_y1 * srcW + src_x1) * 3;
    int srcIdx1_2 = (src_y1 * srcW + src_x2) * 3;
    int srcIdx2_1 = (src_y2 * srcW + src_x1) * 3;
    int srcIdx2_2 = (src_y2 * srcW + src_x2) * 3;

    // bilinear interpolation -- 计算原图在目标图中的x, y方向上的偏移量
    y = y - int(srcH / (scaled_h * 2)) + int(tarH / 2);
    x = x - int(srcW / (scaled_w * 2)) + int(tarW / 2);

    // bilinear interpolation -- 计算resized之后的图的索引
    int tarIdx    = (y * tarW  + x) * 3;

    // bilinear interpolation -- 实现bilinear interpolation + BGR2RGB
    tar[tarIdx + 0] = round(
                      a1_1 * src[srcIdx1_1 + 2] +
                      a1_2 * src[srcIdx1_2 + 2] +
                      a2_1 * src[srcIdx2_1 + 2] +
                      a2_2 * src[srcIdx2_2 + 2]);

    tar[tarIdx + 1] = round(
                      a1_1 * src[srcIdx1_1 + 1] +
                      a1_2 * src[srcIdx1_2 + 1] +
                      a2_1 * src[srcIdx2_1 + 1] +
                      a2_2 * src[srcIdx2_2 + 1]);

    tar[tarIdx + 2] = round(
                      a1_1 * src[srcIdx1_1 + 0] +
                      a1_2 * src[srcIdx1_2 + 0] +
                      a2_1 * src[srcIdx2_1 + 0] +
                      a2_2 * src[srcIdx2_2 + 0]);
}
```

可以看到，不同于opencv，在CUDA编程中我们可以把很多操作放在一起写。这样可以避免多次调用kernel，同时可以实现内存复用，这样可以提高效率。

比如说这一核函数实现了:
bilinear interpolation
+ BGR2RGB
+ shift

这里其实也将可以将这一系列操作写成一个affine transformation(仿射变换)，这个在之后的案例中会讲到

```
Resize(bilinear) in cpu takes:                          uses 1.185867 ms
Resize(nearest) in gpu takes:                           uses 0.021600 ms
Resize(bilinear) in gpu takes:                          uses 0.037280 ms
Resize(bilinear-letterbox) in gpu takes:                uses 0.018336 ms
Resize(bilinear-letterbox-center) in gpu takes:         uses 0.018624 ms
```

通过cuda可以达到65倍的加速