Introduction to CUDA Programming

Pangfeng Liu

Department of Computer Science and Information Engineering

National Taiwan University

Outline

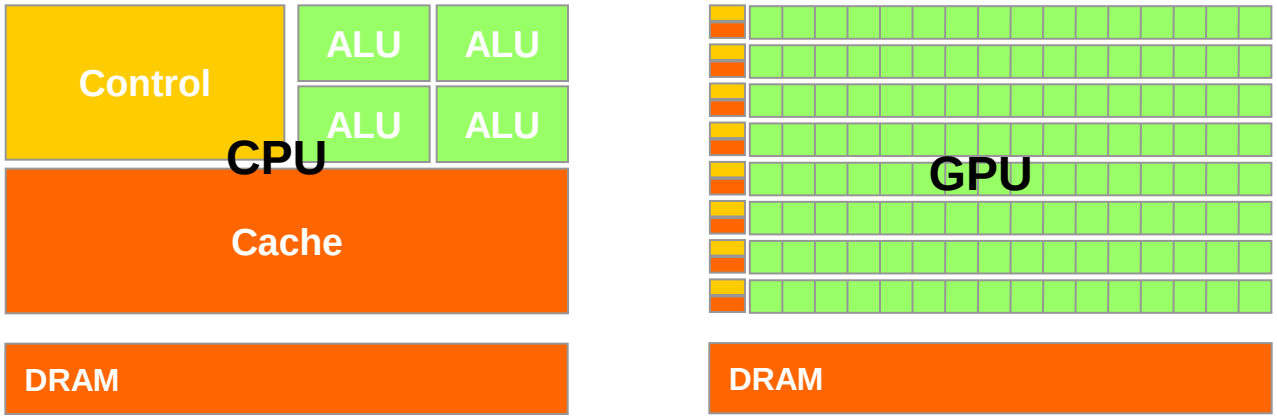
- Introduction
- Private memory
- Performance Tuning

GPU

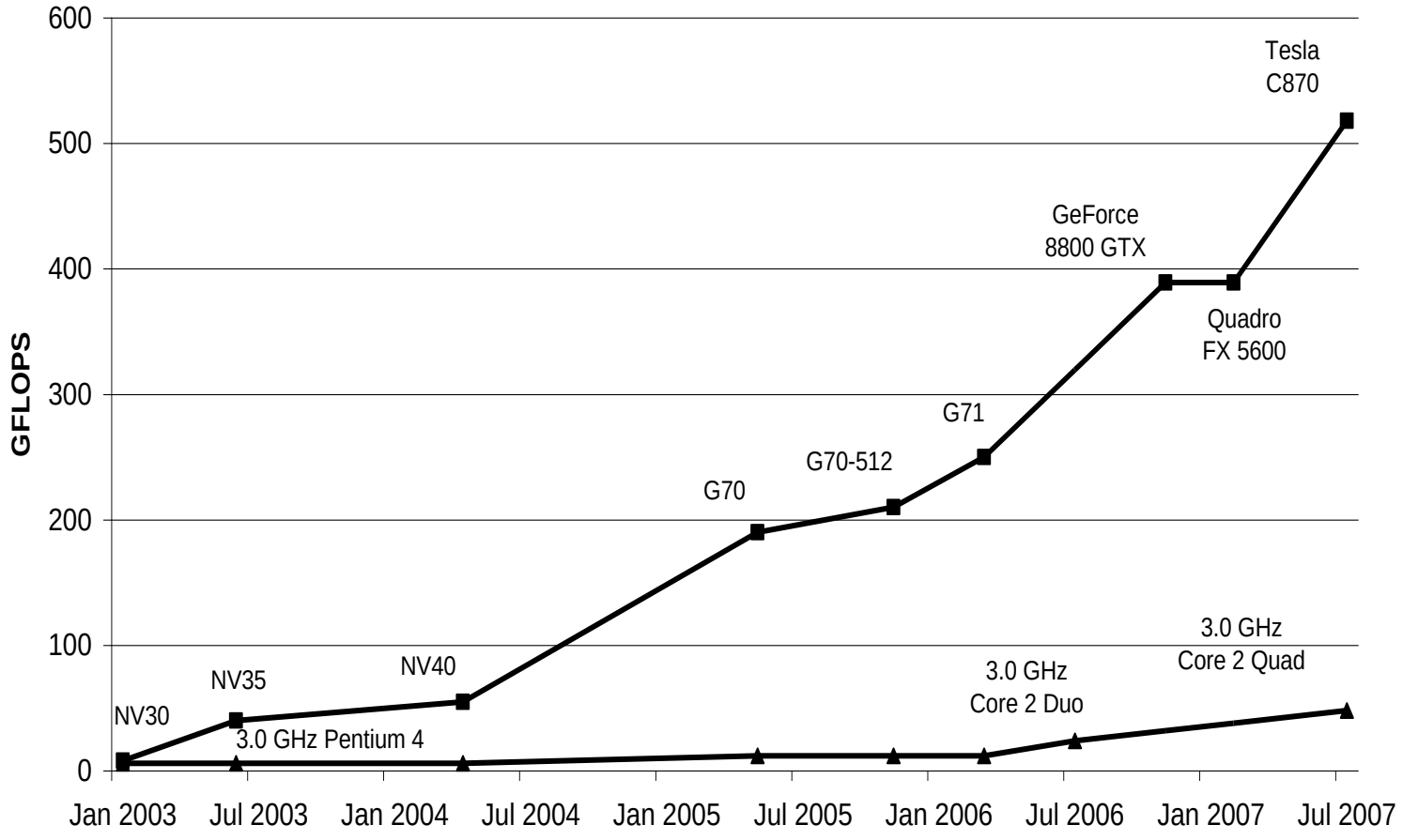
- Graphical Processing Unit
- Used to display 3D graphics on your PC
- Consists of a very large number of transistors.
- Highly parallel computation
  - A large number of pipelines
- Cost-effective
  - The only part that you might consider upgrading in your PC.

CPU vs. GPU

- Less data cache and flow control
- More silicon for computation



Performance in Gflops



GPGPU

- General Purpose computation using GPU — not only for pixels.
- This is not a GPU, but a way to use GPU.

Advantages

- Large data arrays and streaming throughput
- Very large memory bandwidth
- Fine-grain SIMD parallelism
  - A very large number of threads
- Low-latency floating point (FP) computation
  - High power computation capability
- Piggyback on the fast advancing GPU technology

GPGPU Applications

- Tons of applications on GPGPU.org and Nvidia websites.
- http://www.nvidia.com/object/cuda_home.html#
  - Game effects (FX) physics
  - Image processing
  - Physical modeling
  - Computational engineering
  - Matrix algebra
  - Convolution
  - Correlation
  - Sorting

Implication

- Suitable for data-parallel processing
  - The same computation is performed on many data elements in parallel.
  - Low control flow overhead
  - High floating point arithmetic intensity
- Computation intensive threads on large number of data hide large memory latency for each others.
  - No need for large data cache.

Why not before?

- Graphics API only
- Addressing modes is limited by texture size/dimension
- Limited outputs due to shader capabilities
- No instruction on integer and bits
- No interaction between pixels

CUDA in a Long Sentence

- Compute Unified Device Architecture
- A general purpose parallel computing architecture that leverages the parallel compute engine in NVIDIA graphics processing units (GPUs) to solve many complex computational problems in a fraction of the time required on a CPU.

CUDA in a Short Sentence

- Integrates CPU+GPU application C program by running serial parts on CPU and highly parallel parts on GPU.

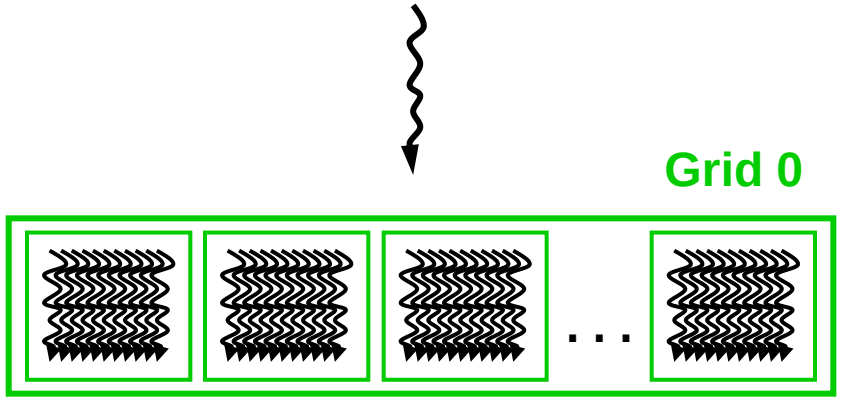
CUDA in You PC

- Can run on your PC with NVIDIA display card.
  - NVIDIA CUDA-enabled products
  - http://www.nvidia.com/object/cuda_learn_products.html
- CUDA download
  - http://www.nvidia.com/object/cuda_get.html

An Illustration

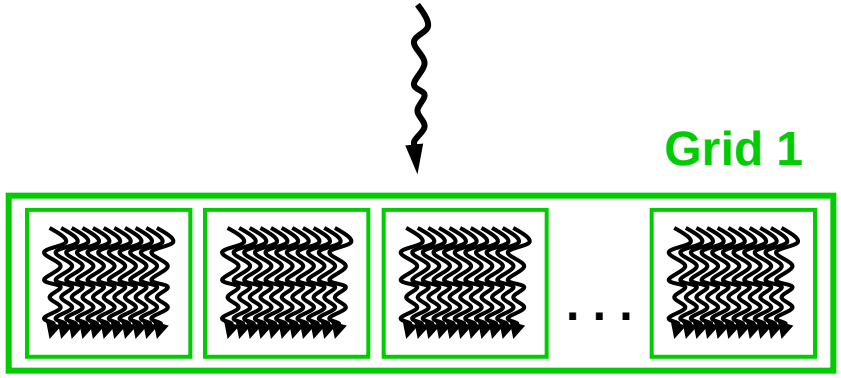
CPU Serial Code

GPU Parallel Kernel
KernelA<<< nBlk, nTid >>>(args);



CPU Serial Code

GPU Parallel Kernel
KernelB<<< nBlk, nTid >>>(args);

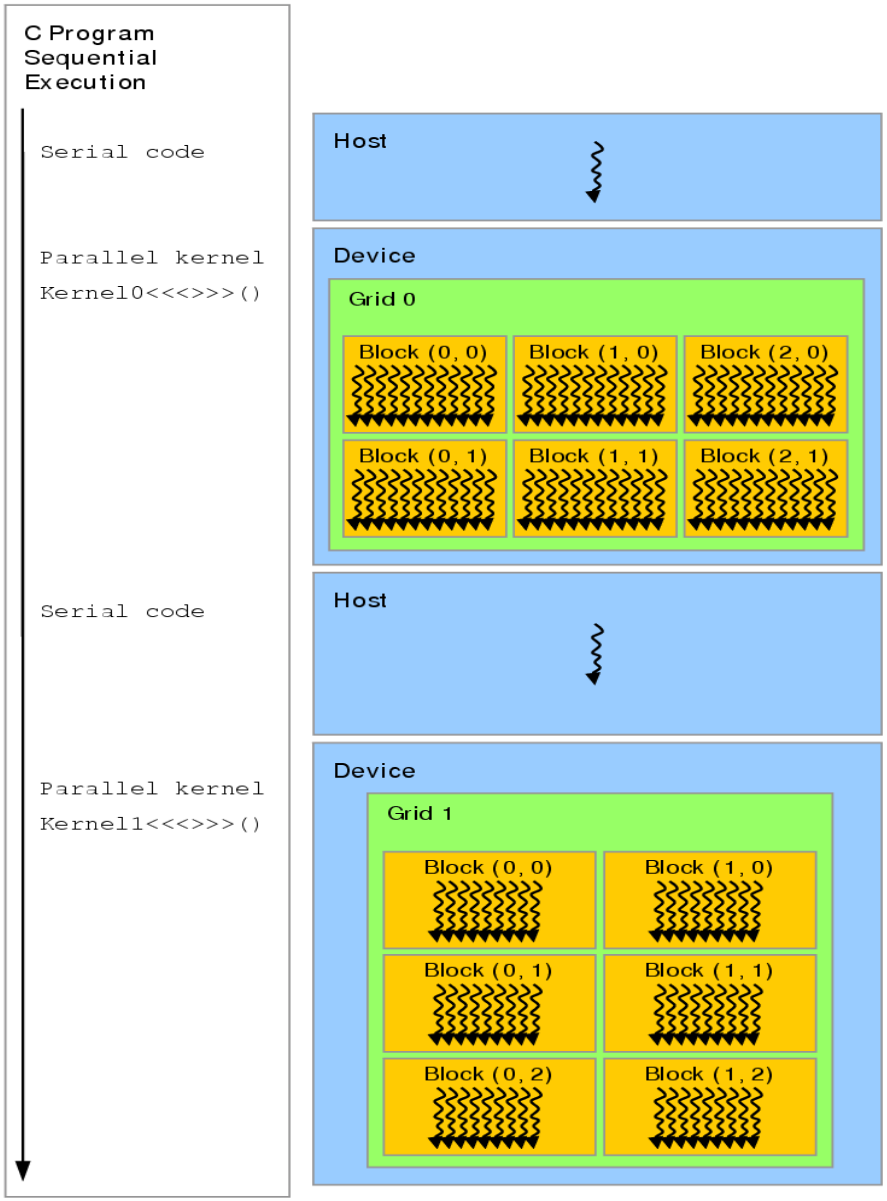


Terminology

- Host
  - CPU
- Device
  - GPU
- Kernel
  - Computation intensive C functions running on devices as threads.
- Grid
  - The view of cores within a device
- Block
  - A basic unit in a device

The Programming Model

- The *host* runs the *sequential* part of the application.
- The *parallel* data intensive parts are written as *kernel* functions, and sent to *devices* for execution (as *threads*).
- The *blocks* are the unit for computation and they are organized as a *grid*.



Serial code executes on the host while parallel code executes on the device.

Figure 2-3. Heterogeneous Programming

GPU and CPU Thread

- GPU threads are very lightweight and have very little creation overhead, but CPU thread creation overhead is very high.
- The implication is that GPU can use a huge number of threads to achieve high performance, but CPU can only use a relatively small number of threads.

Thread id

- Each thread has an id so that it will be able to determine which part of data it is supposed to handle.
  - The thread id is stored in a variable threadIdx
  - The contents of threadIdx depends on the dimensionality of the grid.

Thread Blocks

- A block is an N-dimensional array of threads.
  - N could be 1, 2, or 3.
- The coordinates of a thread within a block are stored within the variable threadIdx as a vector of three elements, as x, y, and z.
- When the host calls the kernel function to run on the devices, it can specify the thread block layout between the name of the kernel function and the parameter list.
  - initialize <<<1, 5>>> (array);
  - One block with a linear layout of 5 threads.

The ThreadIdx

- The kernel function can retrieve the thread index by referencing threadIdx variable.
  - a[threadIdx.x] = threadIdx.x;
  - Note that this is for a one-dimensional layout, so we use the x element only.

Global Memory

- The devices can share global memory.
- This special global memory must be allocated using special allocation routine.
- The prototypes of cudaMalloc is different from malloc.
  - cudaMalloc(&pointer, size);
    - The first argument is the address of the pointer.
  - cudaFree(pointer);

Host Memory

- The host can allocate and free its own memory using malloc and free.
- This memory cannot be used by the devices.

CUDA Computation

- The host allocates and initializes its memory.
- The host allocates the shared memory.
- The host copies the host memory into shared memory.
- The hosts calls kernel function to performs computation on the shared memory.
- The host copies the device memory back to the host memory.
- The host outputs the results.

Memory Transfer

- Host to device
  - cudaMemcpy(device_memory, host_memory, cudaMemcpyHostToDevice);
- Device to host
  - cudaMemcpy(host_memory, device_memory, size, cudaMemcpyDeviceToHost);
- Note that it is always from the second argument to the first argument.

Thread Layout

- The layout of threads in a block is specified by the second argument with the <<< >>>.
- If it is a simple integer, then the threads are organized as a one-dimensional array.
  - You can get the thread index by threadIdx.x.
  - For example, a[threadIdx.x] = threadIdx.x;

More Thread Layout

- For two dimensional thread layout, we need a “block layout information”.
  - dim3 blocks (N, N);
  - This means that the threads in this block are organized as an N by N matrix.
- threadIdx.x and threadIdx.y give you the x and y coordinate in the matrix.
  - a[threadIdx.x][threadIdx.y] *= (threadIdx.x + threadIdx.y);
  - The memory address changes in row major.

Grid Layout

- A grid can have many blocks, just like a block can have many threads.
- You can specify the layout of a grid as the first argument within <<< >>>.
- The format of dim3 is the same for the block layout.

More Grid Layout

- Now consider the following three dimensional layout.
  - A grid has N blocks as a one-dimension array.
  - A block has N^2 threads as a two-dimension array.
- We need to call the kernel function as follows.
  - dim3 blocks (N,N);
  - initialize <<<N, blocks>>> (a);

Block and Thread Indices

- The coordinate information of a block within a grid is stored in blockIdx.
- We can use the following to initialize the sequence number for the previous example.
  - a[blockIdx.x][threadIdx.x][threadIdx.y] = blockIdx.x * N * N + threadIdx.x * N + threadIdx.y;

Execution

- Kernel function is run in grid.
  - <<<grid-dim, block-dim>>>
- A grid has many blocks.
  - The layout of the grid is specified by grid-dim.
  - The layout of the block is specified by block-dim.

Function Qualifiers

- __global__
  - Host code invokes, runs on device.
- __device__
  - Device code invokes, runs on device.
- __host__
  - Host code invokes, runs on host.

An Example

```
#include <stdio.h>
#include <cuda.h>

#define N 8

__global__ void hello(int int_array[N])
{
    int_array[threadIdx.x] *= threadIdx.x;
}
```

An Example

```
int main(void)
{
    int *device_int_array;
    int *host_int_array;
    int i;
    int size = sizeof(int) * N;

    cudaMalloc((void **)&device_int_array, size);
    host_int_array = (int *)malloc(size);

    for (i = 0; i < N; i++)
        host_int_array[i] = i;
    cudaMemcpy(device_int_array, host_int_array, size,
            cudaMemcpyHostToDevice);

    hello <<< 1, N >>> (device_int_array);
    cudaMemcpy(host_int_array, device_int_array, size,
            cudaMemcpyDeviceToHost);

    for (i = 0; i < N; i++)
        printf("host_int_array[%d] = %d\n", i, host_int_array[i]);

    cudaFree(device_int_array);
    free(host_int_array);
}
```

Output

```
pangfeng@cuda01:~/examples> ./a.out
host_int_array[0] = 0
host_int_array[1] = 1
host_int_array[2] = 4
host_int_array[3] = 9
host_int_array[4] = 16
host_int_array[5] = 25
host_int_array[6] = 36
host_int_array[7] = 49
```

Another Example

```
#include <stdio.h>
#include <cuda.h>

#define N 4

__global__ void hello(int int_array[N][N])
{
    int_array[threadIdx.x][threadIdx.y]
                    *= (threadIdx.x + threadIdx.y);
}
```

```
int main(void)
{
    int *device_int_array;
    int i, j;
    int size = sizeof(int) * N * N;
    int host_int_array[N][N];
    dim3 blocks (N,N);
    cudaMalloc((void **)&device_int_array, size);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            host_int_array[i][j] = i + j;
    cudaMemcpy(device_int_array, host_int_array, size,
               cudaMemcpyHostToDevice);
    hello <<< 1, blocks >>> ((int (*)[N])device_int_array);
    cudaMemcpy(host_int_array, device_int_array, size,
               cudaMemcpyDeviceToHost);
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            printf("host_int_array[%d][%d] = %d\n", i, j,
                   host_int_array[i][j]);

    cudaFree(device_int_array);
}
```

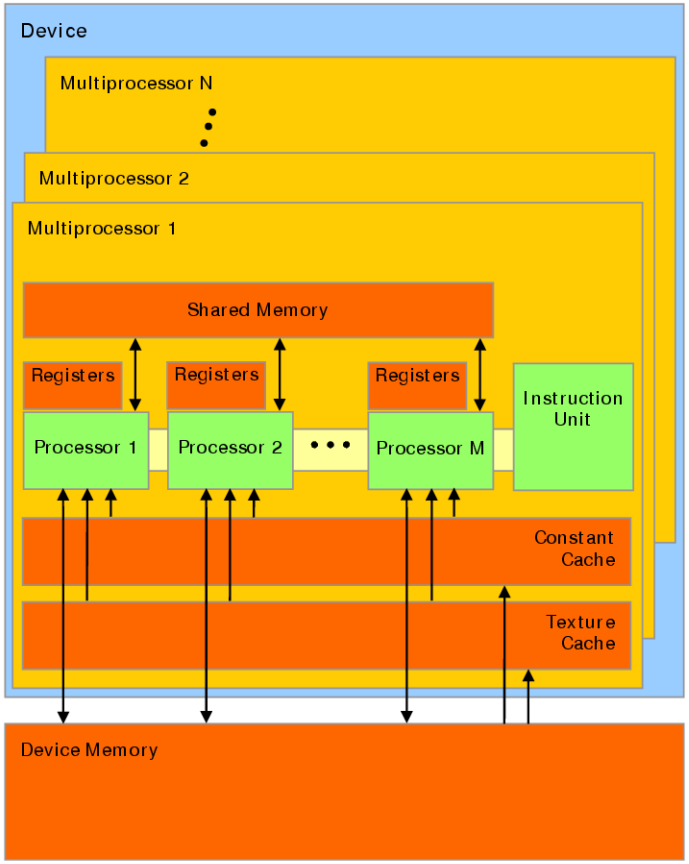
Output

```
pangfeng@cuda01:~/examples> ./a.out
host_int_array[0][0] = 0
host_int_array[0][1] = 1
host_int_array[0][2] = 4
host_int_array[0][3] = 9
host_int_array[1][0] = 1
host_int_array[1][1] = 4
host_int_array[1][2] = 9
host_int_array[1][3] = 16
host_int_array[2][0] = 4
host_int_array[2][1] = 9
host_int_array[2][2] = 16
host_int_array[2][3] = 25
host_int_array[3][0] = 9
host_int_array[3][1] = 16
host_int_array[3][2] = 25
host_int_array[3][3] = 36
```

Hardware

- A GPU has many multiprocessors.
- A multiprocessor has
  - Core for computation
  - Register for computation
  - Memory for storage

An Illustration of GPU



A set of SIMT multiprocessors with on-chip shared memory.

Figure 3-1. Hardware Model

Block and Multiprocessor

- A thread block runs on a multiprocessor as a unit.
  - No “partial” blocks
  - No migration
- A multiprocessor may run multiple blocks.
  - A multiprocessor has the resources to run many threads simultaneously.
  - The number of threads is limited by resource. If one wishes to have more threads, he should get more blocks, not more threads per block.

Synchronization

- Threads with the same block can synchronize with __syncthreads().
- Kernel waits for all previous CUDA calls returns, but the control returns to host immediately.
- cudaMemcpy() starts after all previous CUDA calls returns, and return only when memory operation finishes.

Function Qualifiers

- __global__
  - Host code invokes, runs on device.
- __device__
  - Device code invokes, runs on device.
- __host__
  - Host code invokes, runs on host.

__global__ Restrictions

- Must return void.
- No recursion
- No static variables
- No variable number of arguments

Combination

- __host__ and __device__ can be combined.
  - The same code could be run on both device and host for performance and convenience.
  - Same source but different binary codes for CPU and GPU.

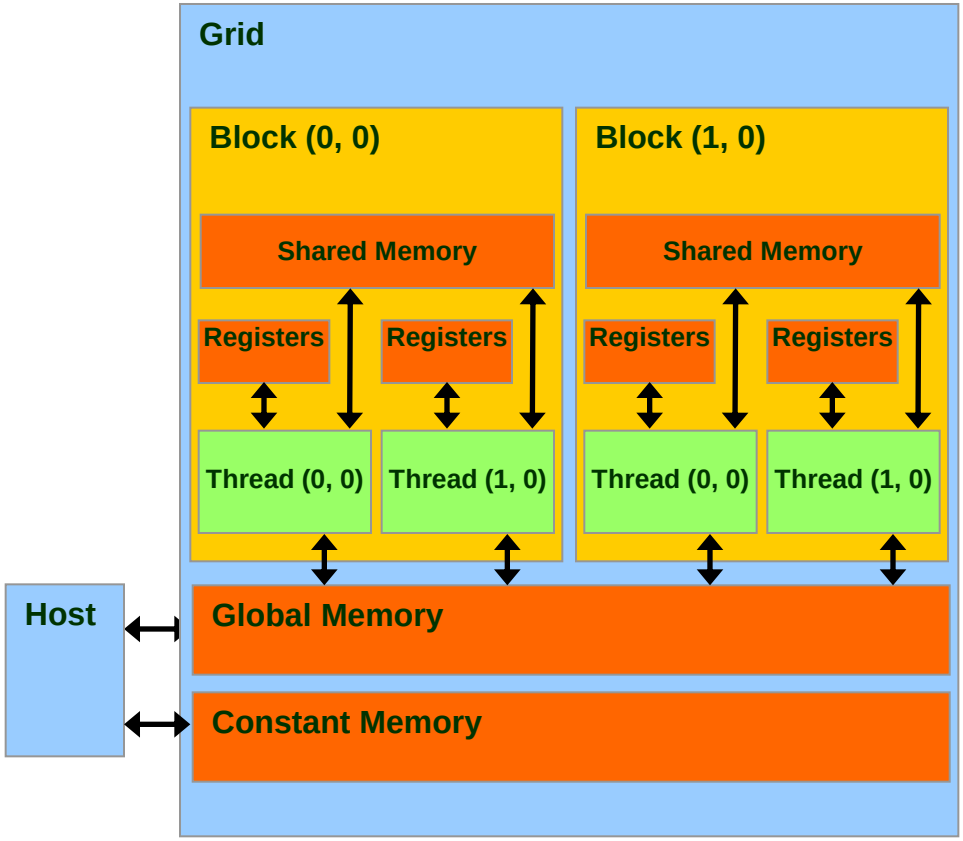
Built-in Device Variables

- All of type dims
  - With constructor, e.g. (N, N).
  - Has component x, y, and z.
- Block id and grid dimension
  - gridDim
  - blockIdx
- Thread id and block dimension
  - threadIdx
  - blockDim

Memory Hierarchy

- Global and constant memory
  - Large
  - Shared by all threads and host
  - Off-chip
- Shared memory
  - Small
  - Shared by all threads in the same block
  - On-chip

An Illustration



Implications

- A large number of threads hide instruction and memory latency for each other.
- Threads of the same block can share data via fast shared memory.
- All threads can share data via global memory.

GPU Variable Types

- Classification
  - __device__
  - __shared__
  - __constant__
  - __local__

GPU Variable Types

- Where the variables will be allocated
- The lifetime of the variable

__device__ Variables

- So-called global memory
- Large and slow
- Allocated with cudaMalloc
- Accessible by all threads and host
- Valid throughout the entire execution

__shared__ Variables

- So called shared memory
- Small and fast
- Allocated by declaration or kernel invocation
- Accessible by threads in the same block, but values assigned are guaranteed to be visible by other threads only after __syncthreads().
- Valid only during kernel execution

__shared__ by Declaration

```
__global__ void kernel(…)
{
    __shared__ float sData[256];
    …
}

int main(void)
{
    kernel<<<griddim, blockdim>>>(…);
}
```

__shared__ by Kernel Invocation

```
__global__ void kernel(…)
{
    extern shared float sData[]; /* we will have 256 floats here. */
    …
}

int main(void)
{
    kernel<<<griddim, blockdim, 256 * sizeof(float)>>>(…);
}
```

Kernel Invocation

- <<<griddim, blockdim, sharedmemsize>>>
- sharedmemsize is optional, and the default value is 0.
- If not 0 then it specifies the number of bytes of shared memory allocated for each block.
  - The allocate shared memory can be accessed as sData.

__constant__ Variables

- Stored in constant memory.
- Read-only on device, but can be set on host.
- Accessible by all threads and host.
- Valid throughout the entire execution.

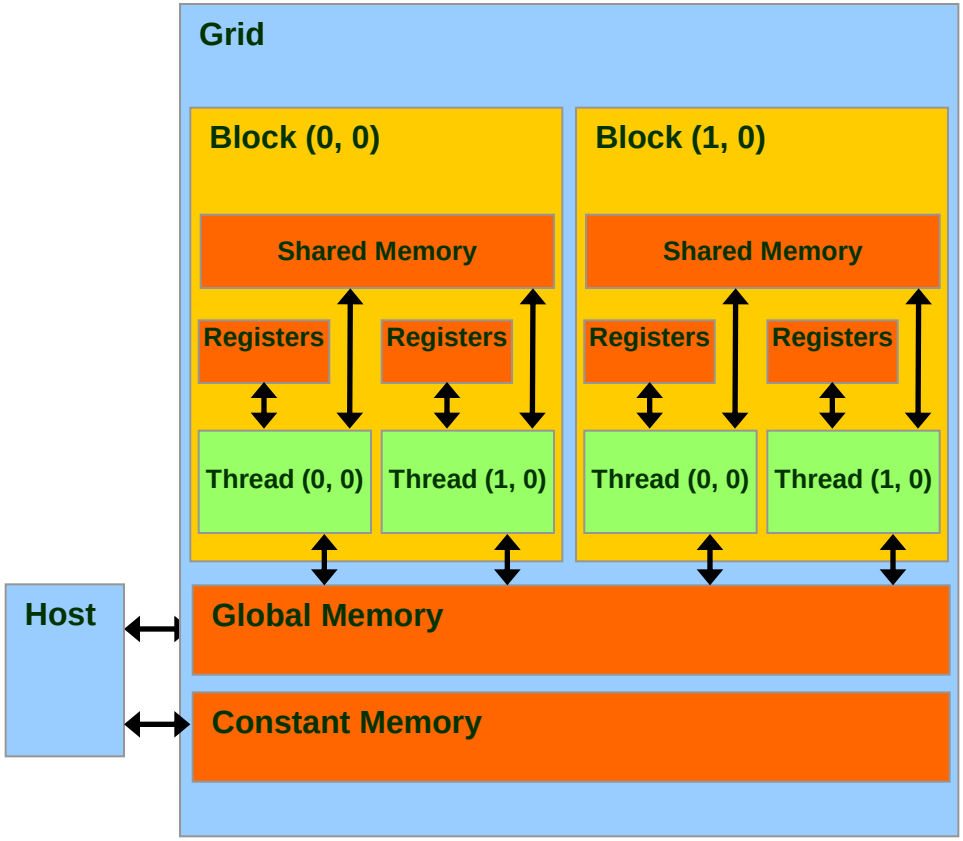
__local__ Variables

- Stored in registers if not array.
- Allocated by declaration.
- Accessible by the thread that declares it.
- Valid when the thread is active.

Access

- Host may access
  - __global__
  - __constant__
- Kernel only
  - __local__
  - __shared__

An Illustration



Strategy

- Shared memory is much faster than global memory
  - Host moves data into global memory.
  - Thread moves data from global memory into private memory for processing.
  - Thread computes.
  - Copy results from private to global memory.

Timing

- The clock() routine returns the wall clock time in clock ticks as a clock_t.
- The number of clock ticks per second is in the constant CLOCKS_PER_SEC.
- Before calling clock make sure all CUDA routines finish by calling cudaThreadSynchronize() at host.

Thread Synchronization

- __syncthreads()
  - Calls from device code
  - Synchronizes all threads in the same block
  - Enforces consistent view on shared memory
- cudaThreadSynchronize()
  - Calls from host code.
  - Synchronizes all threads
  - Enforces correct timing on kernel

Matrix Multiplication

- Two versions to compute C = A x B
- Both A and B are in global memory
  - Version 1 retrieves data from global memory directly.
  - Version 2 retrieves data from global memory and stores in shared memory.

Using Global Memory

```
#include <stdio.h>
#include <cuda.h>
#include <time.h>

#define N 1000

__global__ void multiply(int A[N][N], int B[N][N], int C[N][N])
{
    int k, sum = 0;
    for (k = 0; k < N; k++)
        sum += A[blockIdx.x][k] * B[k][threadIdx.x];

    C[blockIdx.x][threadIdx.x] = sum;
}
```

Main Program

```
int main(void)
{
    int *device_A, *device_B, *device_C;
    int *host_A, *host_B, *host_C;
    int i, j, k;
    int size = sizeof(int) * N * N;
    int *aptr, *bptr, *cptr;

    dim3 blocks(N, N);
    clock_t start, elapsed;

    cudaMalloc((void **)&device_A, size);
    cudaMalloc((void **)&device_B, size);
    cudaMalloc((void **)&device_C, size);
    host_A = (int *)malloc(size);
    host_B = (int *)malloc(size);
    host_C = (int *)malloc(size);
```

Initialize and Copying

```
aptr = host_A;
bptr = host_B;
cptr = host_C;
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++) {
        *aptr++ = 1;
        *bptr++ = 1;
        *cptr++ = 0;
    }

cudaThreadSynchronize();
start = clock();

cudaMemcpy(device_A, host_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(device_B, host_B, size, cudaMemcpyHostToDevice);
```

Kernel Function

```
multiply <<< N, N >>> ((int (*)[N])device_A, (int (*)[N])device_B,
                       (int (*)[N])device_C);
cudaMemcpy(host_C, device_C, size, cudaMemcpyDeviceToHost);
cudaThreadSynchronize();
elapsed = clock() - start;
printf("the multiplcaition takes %f seconds\n",
      (double)(elapsed) / CLOCKS_PER_SEC);

k = 0;
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        printf("host_C[%d][%d] = %d\n", i, j, host_C[k++]);

cudaFree(device_A);
cudaFree(device_B);
cudaFree(device_C);
free(host_A);
free(host_B);
free(host_C);
}
```

Using Shared Memory

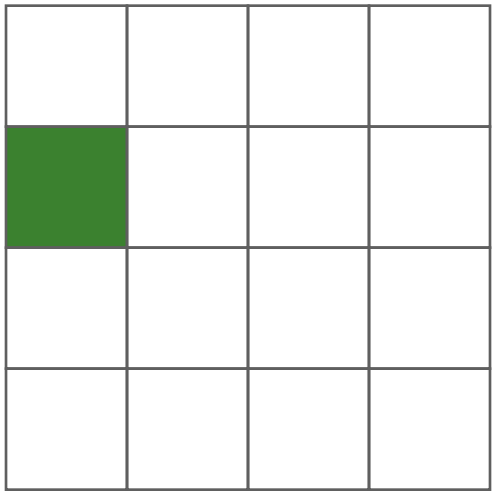
- Organize the grid as a N/b by N/b grids, and each block has b by b threads.
- Each thread block will declare two b by b shared arrays in shared memory (for A and B).
- Each thread in a block will be responsible for loading an element from A, and an element from B.

Matrix Loading

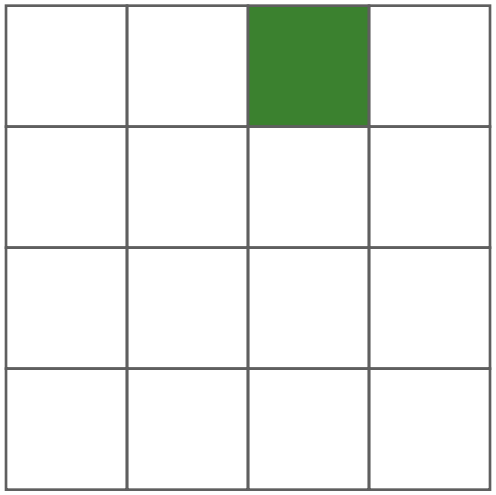
- The matrix A and B will be divided into N/b * N/b sub-matrices, each of size b by b.
- If a block is in the i-th row and j-th column of the grid, then the threads in the block will load the first sub-matrix (of size b by b) from the i-th sub-matrix row of A, and the first sub-matrix (of size b by b) from the j-th sub-matrix column of B, then the threads will load from the second sub-matrix, and so on.

An Illustration

- First step



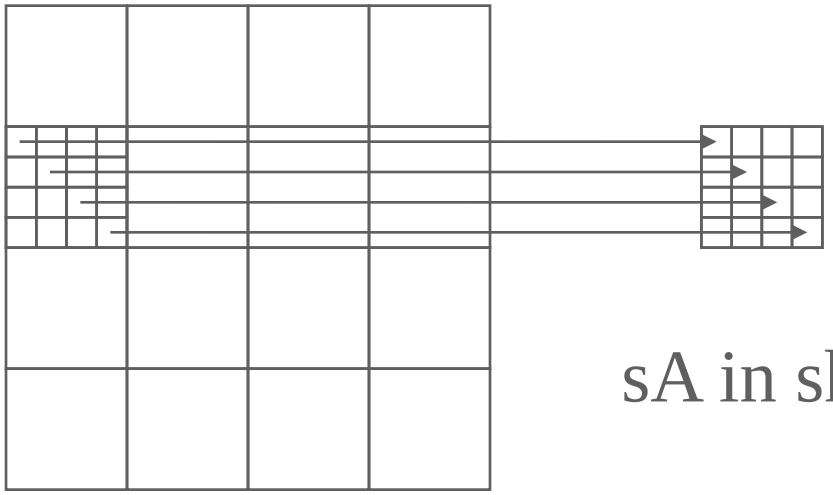
A



B

An Illustration

- Each thread moves an element in A.

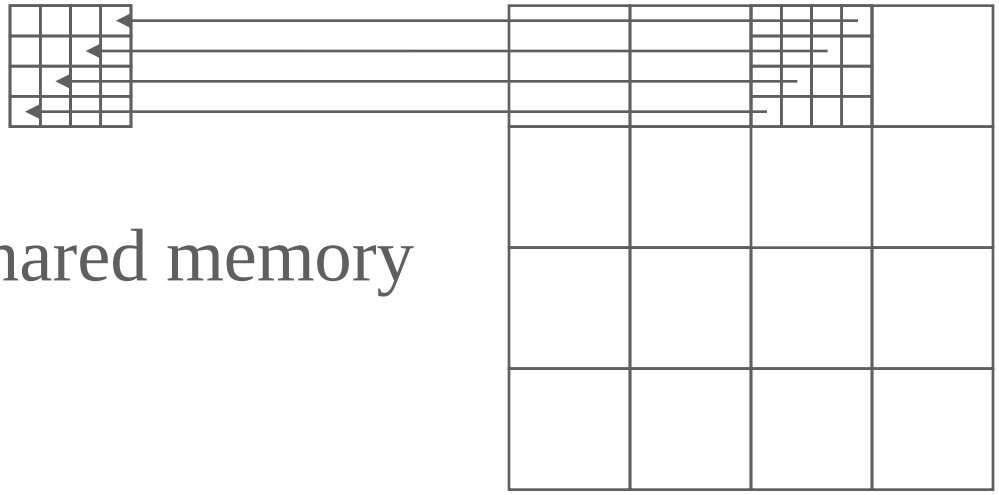


sA in shared memory

A in global memory

An Illustration

- Each thread moves an element in B

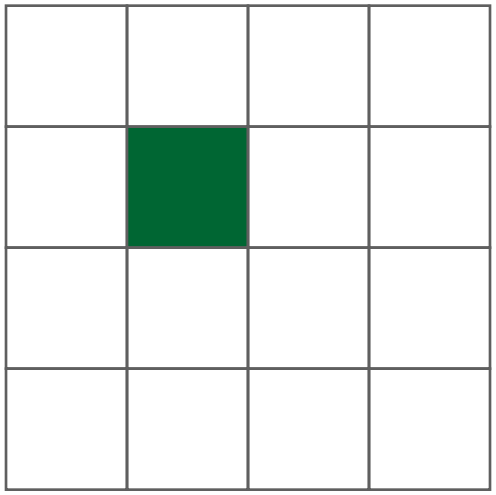


sB in shared memory

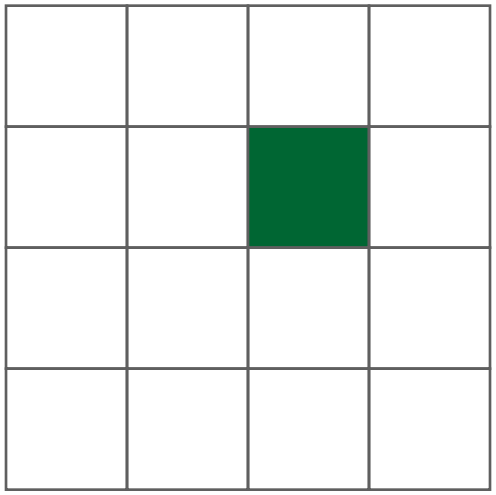
B in global memory

An Illustration

- Second step



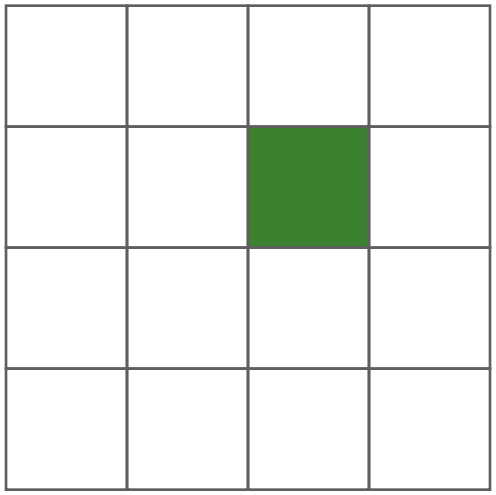
A



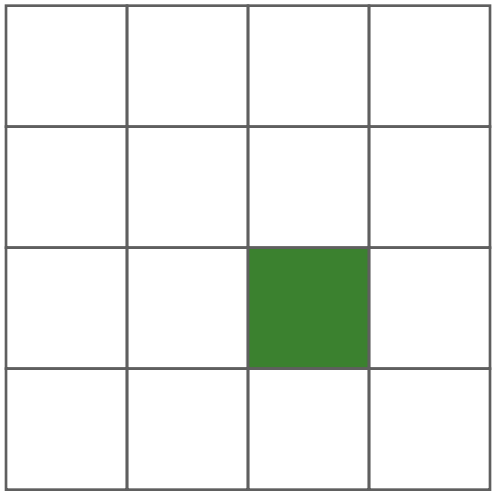
B

An Illustration

- Third step



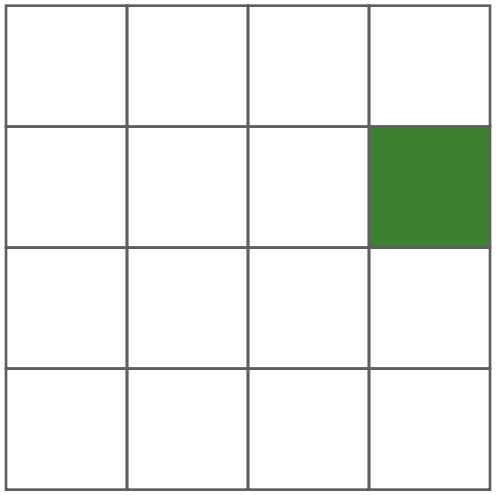
A



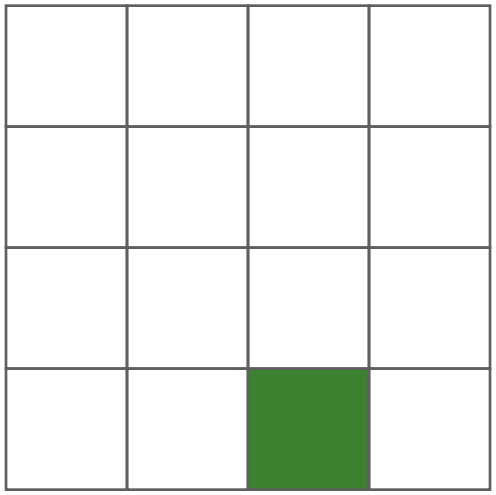
B

An Illustration

- Final step



A



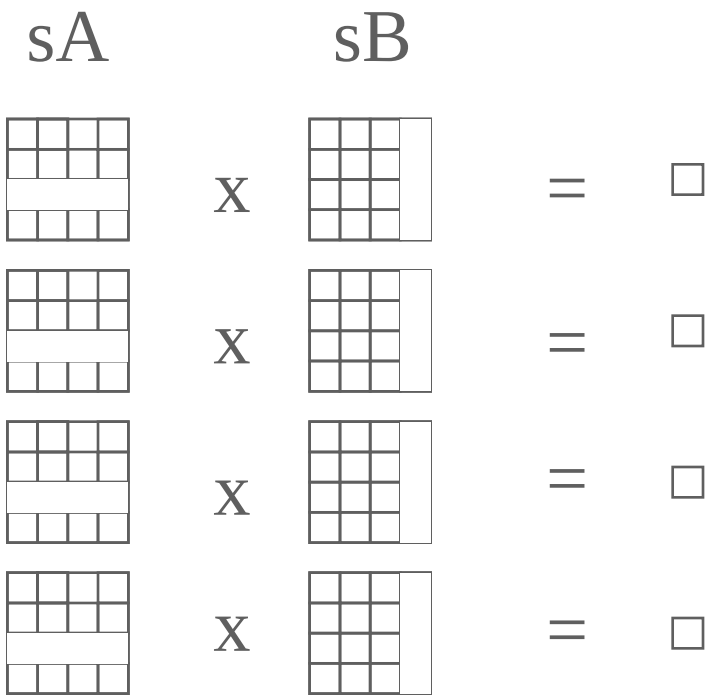
B

Inner Product

- Every time the threads load two sub-matrix (A and B), it performs an inner product it is responsible for.
  - The thread at the i-th row and j-th column of a thread block will compute the inner product of the i-th row of sub-matrix A and j-th column of sub-matrix B.
- The sum of these inner products will be the final C_{ij}

An Illustration

- Each thread computes an element in C



Using Shared Memory

```
__global__ void multiply(int A[N][N], int B[N][N], int C[N][N])
{
    int k, sum = 0;
    int row = blockIdx.x * b + threadIdx.x;
    int column = blockIdx.y * b + threadIdx.y;
    int r;
    __shared__ int sA[b][b];
    __shared__ int sB[b][b];

    for (r = 0; r < b; r++) {
        sA[threadIdx.x][threadIdx.y] = A[row][r * b + threadIdx.y];
        sB[threadIdx.x][threadIdx.y] = B[r * b + threadIdx.x][column];
        __syncthreads();
        for (k = 0; k < b; k++)
            sum += sA[threadIdx.x][k] * sB[k][threadIdx.y];
        __syncthreads();
    }
    C[row][column] = sum;
}
```

Performance Tuning

- Machine configuration
- Accurate timing
- Block size
- Control flow
- Memory access
- Unroll
- Double buffering

Machine Configuration

- Use cudaGetDeviceCount(&device_count) to determine the number of devices in the system.
- Use cudaGetDeviceProperties(cudaDeviceProp *, int deviceid) to determine the important parameters in performance tuning.
  - Warp size
  - Maximum threads per block

cudaDeviceProp

- char name[256];
- size_t totalGlobalMem;
- size_t sharedMemPerBlock;
- int regsPerBlock;
- int warpSize;
- size_t memPitch;
- int maxThreadsPerBlock;
- int maxThreadsDim[3];
- int maxGridSize[3];

cudaDeviceProp

- size_t totalConstMem;
- int major;
- int minor;
- int clockRate;
- size_t textureAlignment;
- int deviceOverlap;
- int multiProcessorCount;

Acurate Timing

- Use cuda events for accurate timing, since an event has time information.
- Place an start event into the event stream 0.
- Place the code for timing.
- Place an end event into the event stream 0.
- Wait for the end event to finish.
- Compute the elapse time between the start and the end events.

Event Routines

- Events are declared as of type cudaEvent_t.
- Events are initialized with cudaEventCrteate(*cudaEvent_t).
- Record events into the operation stream by cudaEventRecord(cudaEvent_t, int stream).
  - Use stream 0, which is for all events.
- To wait for, or synchronize with, the end event with cudaEventSynchronize(cudaEvent_t).
- Compute the elapsed time between the start end end events with cudaEventElapsedTime(float *elapsed_time, cudaEvent_t start, cudaEvent_t end).
- Destroy events with cudaEnentDestroy(cudaEvent_t).

An Example

```
float time;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
```

An Example

```
cudaEventRecord(start, 0);
multiply <<< grids, blocks >>> ((int (*)[N])device_A, (int
    (*)[N])device_B, (int (*)[N])device_C);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);
printf("the multiplcaition takes %f seconds\n", time);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Block Size

- How many threads should go into a block?
  - Limitation on the resource of multiprocessors

Thread Block

- Up to 512 threads per block
- Up to three dimensions
- Has shared memory
- Can synchronize with __syncthread()

Multiprocessor

- So called Streaming Multiprocessors (SM)
- An SM can run a limited number of blocks and threads.
  - 8 blocks and 768 threads for G80
- An SM will maintain thread and block ids for all thread blocks running on it, and schedule these threads for execution.

Warps

- Thread are scheduled in unit of 32 (called Warp).
- CUDA programmer cannot see this, but should be aware of this.
- Only one warp (from all blocks assigned to a SM) will execute on a SM at any given time.

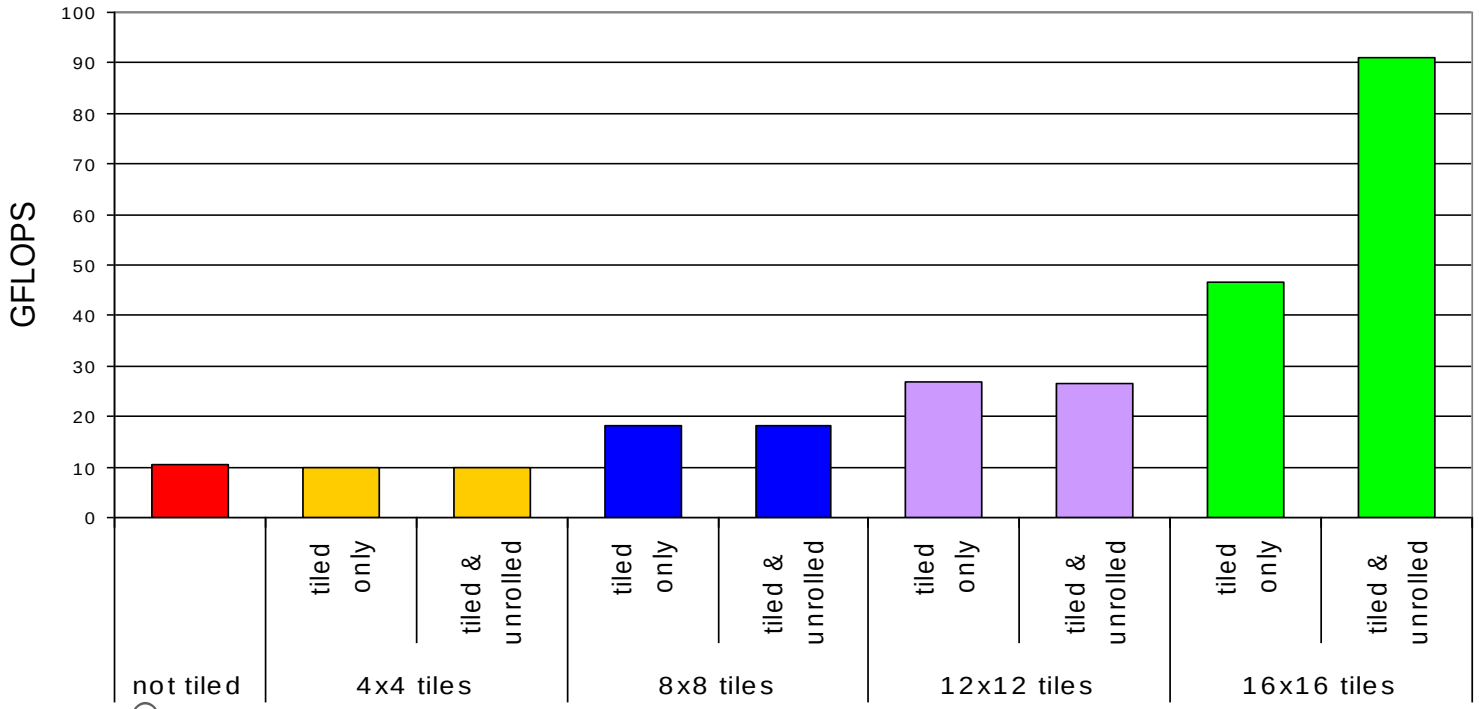
Warp Scheduling

- Operand-ready warps are eligible for execution.
- It takes 4 cycles to dispatch the instruction to a warp.
- It takes 200 cycles to fetch from global memory.
- How many warps does it take to hide memory latency?

Block Size

- 8 by 8 block size
  - 768 / 64 = 12 > 8 blocks
  - Only 8 blocks = 512 threads are used
- 16 by 16 blocks
  - 768 / 256 = 3 blocks
  - All 3 blocks = 768 threads are used
- 32 by 32
  - 1024 threads cannot go into a SM

Tiling Size Effects



Control Flow

- SPMD program
  - Threads do things according to thread id.
- A warp should follow the same control path as offten as possible.

Granularity of Control Flow

- If threads with a warp take different control paths, they will be serialized.
  - Remember that the same instruction will be issued to all threads in the same block.
  - That means we will not have good speedup.
  - Therefore we should get all threads in a warp to go along the same control flow as much as possible.

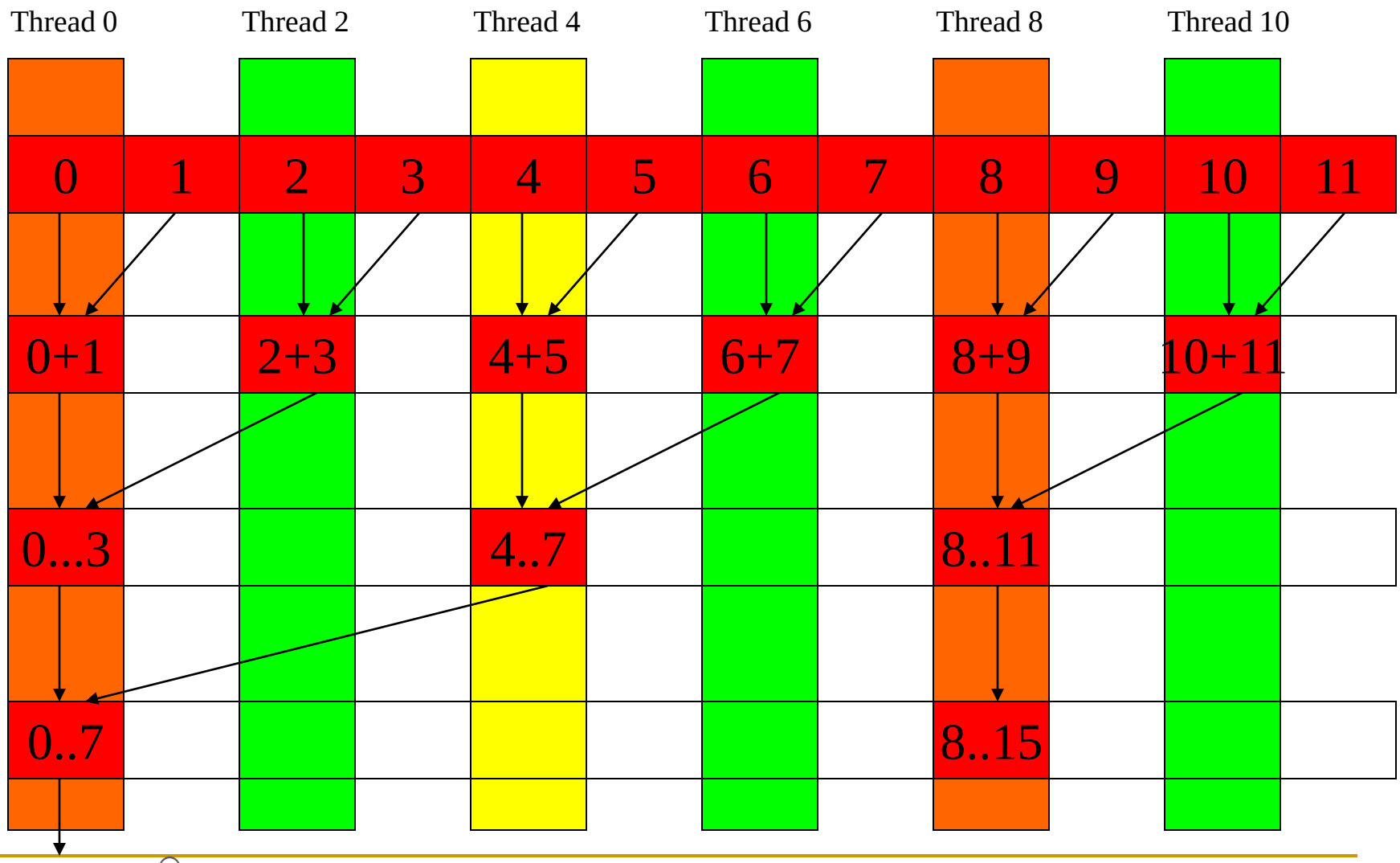
Examples

- if (threadIdx.x > 2)
  - Thread 0 and 1 will take else.
  - 30 other threads take then.
  - Two groups will be serialized.
- If (threadIdx.x / groupsize > 2)
  - If groupsize is a multiple of 32, all threads in a warp go along the same control path.

Sum of N Numbers

- To add the numbers in a global array of N elements.
- Use a block of N threads to compute the sum.
- The computation is in log N phases.
  - In the first phase all even number threads with index i add the element in the odd number threads that have indices i + 1 to them.
  - The next phase all threads of indices multiple of 4 add the elements in the threads that have indices i + 2 to them.
  - We repeat this process until the first element has the sum of all elements.

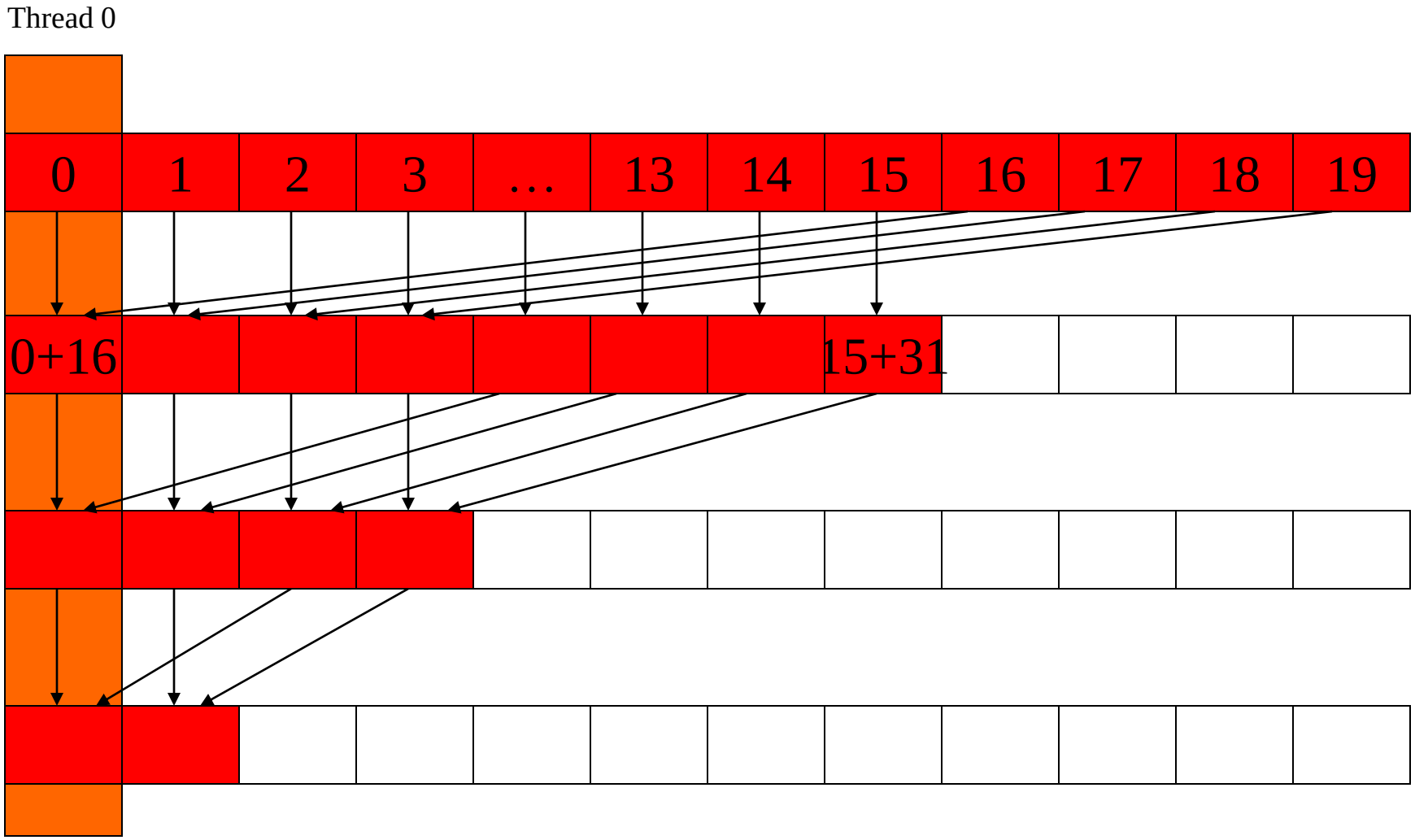
An Illustration



Reverse Order

- Add the elements in decreasing strides.
- In the first iteration, those threads with indices smaller than N/2 will add the elements that have indices N/2 greater than them to the element they have.
- Then the stride reduces by half and we do it again.
- We repeat the process until the first element has the sum of all numbers.

An Illustration



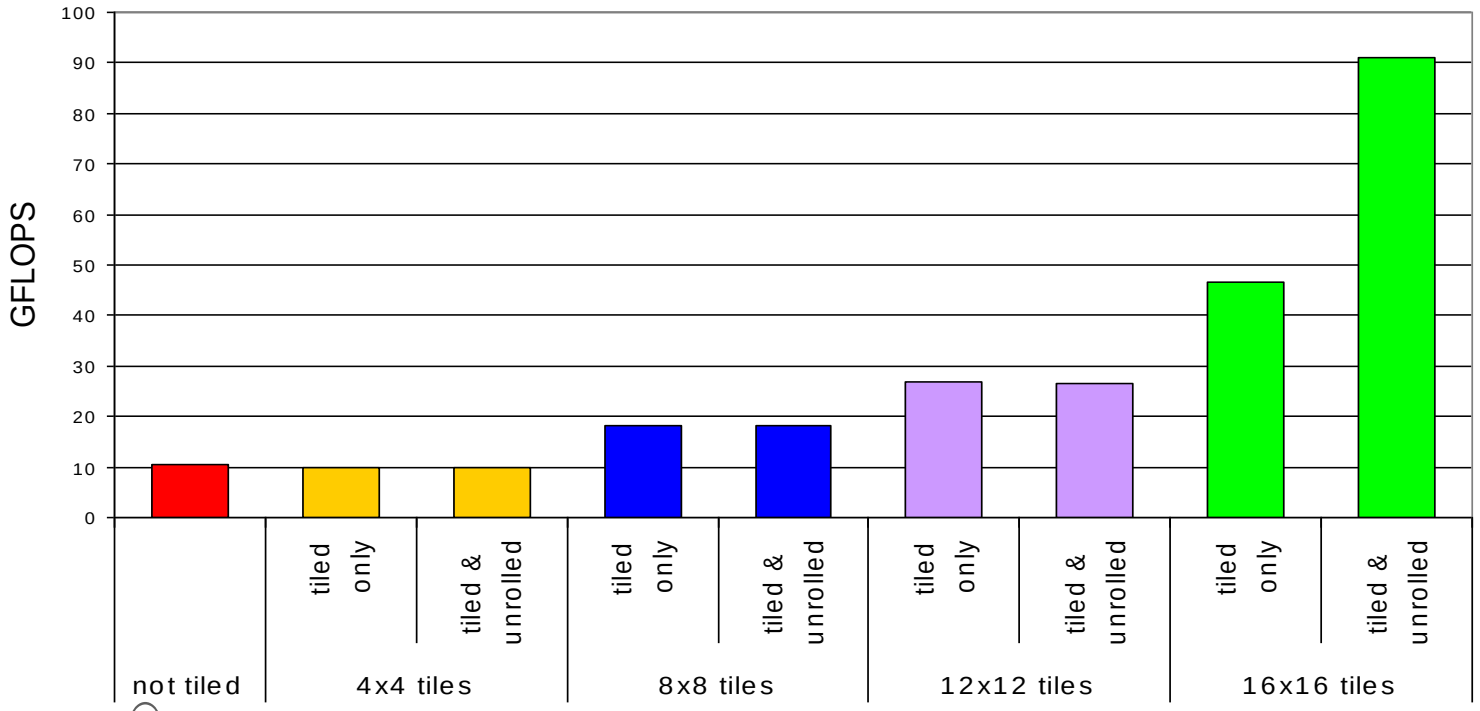
Memory Access

- A warp should access memory in horizontal (contiguous) order .
- If horizontal access is not possible, one should reduce the stride in rows.

Tiled Matrix Multiplication

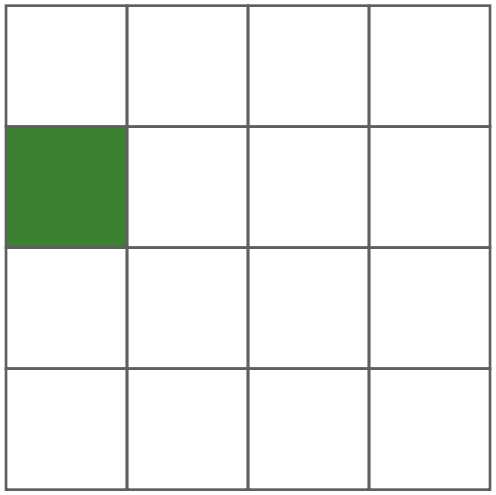
- In version 1 (using global memory only), the warps reference data horizontally (N) and vertically (N).
- In version 2 (using global and shared memory), the warps reference data horizontally (b) and vertically (b).

Tiling Size Effects

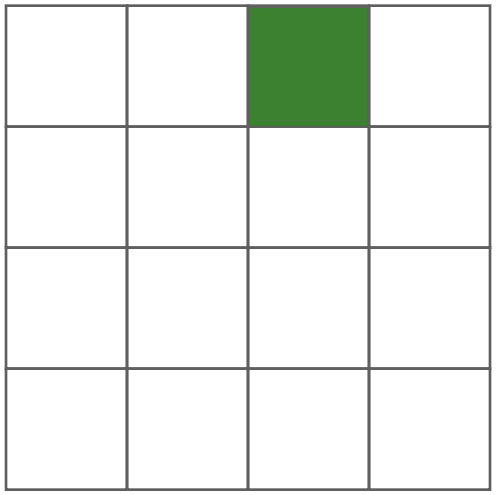


An Illustration

- First step



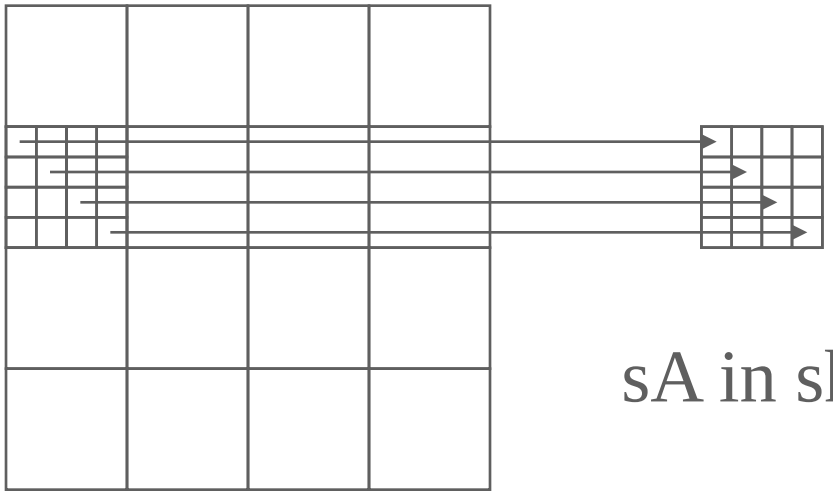
A



B

An Illustration

- Each thread moves an element in A.

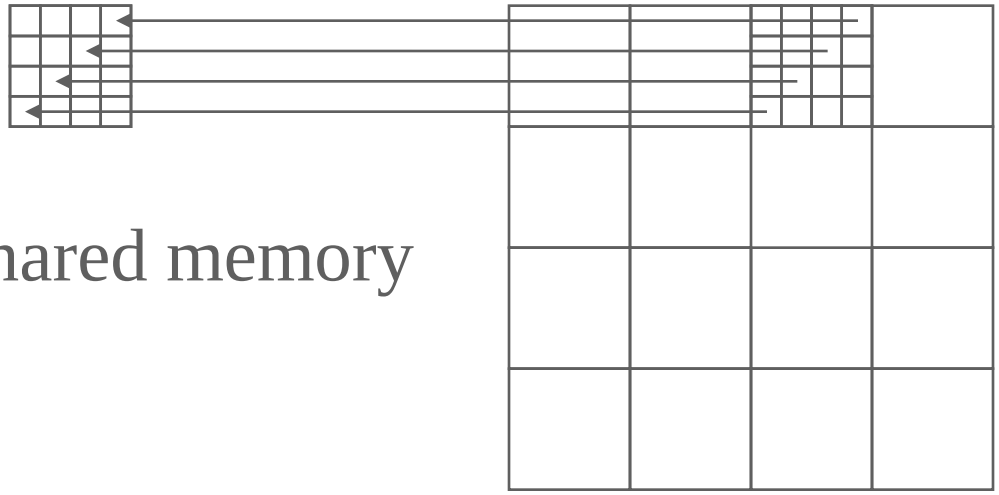


sA in shared memory

A in global memory

An Illustration

- Each thread moves an element in B

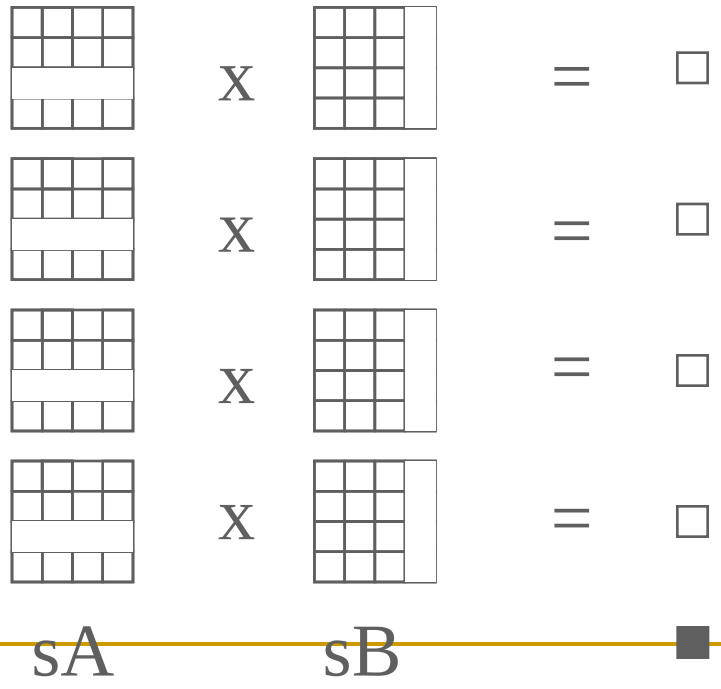


sB in shared memory

B in global memory

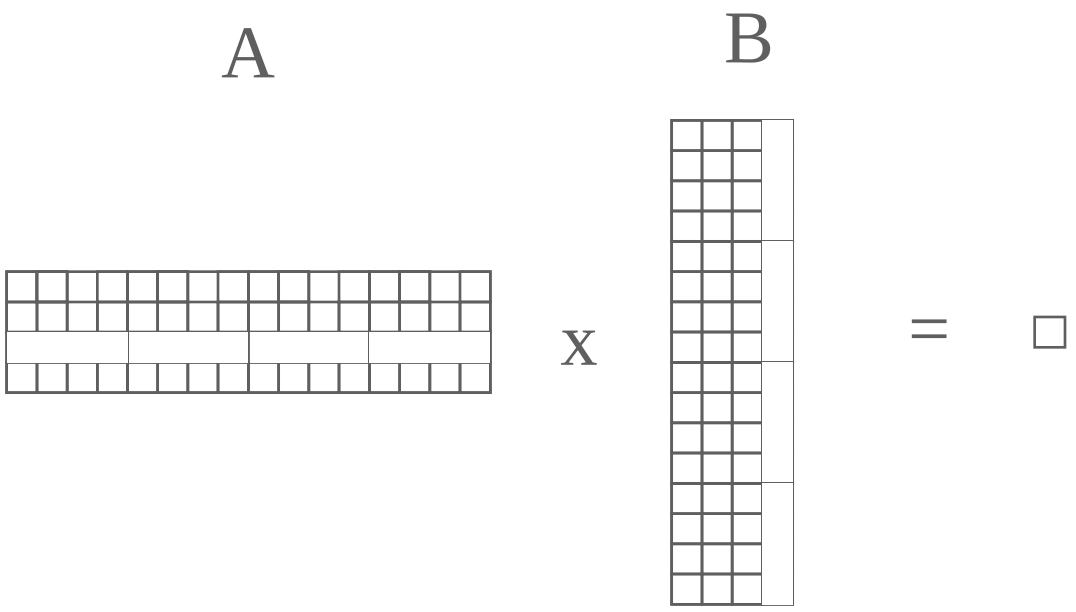
An Illustration

- Each thread computes an element in C using shared memory.



An Illustration

- Each thread computes an element in C using global memory.



Unroll

- Inline the most time consuming loop.
- for (k = 0; k < b; k++)
  - sum += sA[threadIdx.x][k] * sB[k][threadIdx.y];
- Change it to the following.
  - sum += sA[threadIdx.x][0] * sB[0][threadIdx.y] + sA[threadIdx.x][1] * sB[1][threadIdx.y] + … + sA[threadIdx.x][7] * sB[7][threadIdx.y];

Advantage

- Without loop overhead
  - All indices are determined at compile time.
- Hopefully better function unit utilization.
- Larger amount of computation between jumps.

Double Buffering

- Use two buffers
  - One for ongoing memory fetch.
  - One for current computation.

An Illustration

- Load A and B into “current” shared memory
- Repeat
  - Load A and B into “next” shared memory.
  - Compute on “current” shared memory.
  - Switch the current and next memory.