# Optimization Techniques for Parallel Code: 2. CUDA basics

Sylvain Collange
Inria Rennes – Bretagne Atlantique
http://www.irisa.fr/alf/collange/
sylvain.collange@inria.fr

OPT 2018

# From theory to practice

- Last week: programming paper machines
  - PRAM, BSP, multi-BSP
  - Parallel algorithms: reduction, parallel prefix, iterated stencil
- Today: programming actual parallel processors
  - GPUs: Graphics Processing Units
  - Nvidia CUDA programming environment

# Outline

- GPU programming environments
- CUDA host side
- CUDA device side: threads, blocks, grids
- Expressing parallelism
  - Vector add example
- Managing communications
  - Parallel reduction example
- Re-using data
  - Matrix multiplication example

# GPU development environments

For general-purpose programming (not graphics)

- Multiple toolkits
  - NVIDIA CUDA
  - Khronos OpenCL
  - Vulkan Compute
  - Microsoft DirectCompute
  - Google RenderScript
- Mostly syntactical variations
  - Underlying principles are the same
- In this course, focus on NVIDIA CUDA

# Higher-level programming

- Directive-based
  - OpenACC
  - OpenMP 4.x
- Language extensions / libraries
  - Microsoft C++ AMP
  - Intel Cilk+
  - NVIDIA Thrust, CUB
- Languages
  - Intel ISPC

...

- Most companies agree we need common standards...
  - As long as the standard is their own product!

# Outline

- GPU programming environments
- CUDA host side
- Parallel programming models: BSP, multi-BSP
- CUDA device side: threads, blocks, grids
- Expressing parallelism
  - Vector add example
- Managing communications
  - Parallel reduction example
- Re-using data
  - Matrix multiplication example

# Hello World in CUDA

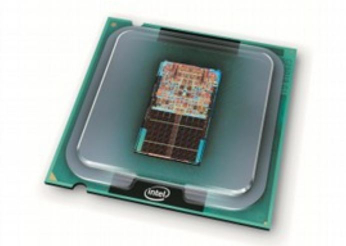
- CPU “host” code + GPU “device” code

hello.cu:

```
__global__ void hello() {
    printf("Hello World!\n");
}

int main() {
    hello<<<1,1>>>();
    return 0;
}
```

Device code

Host code

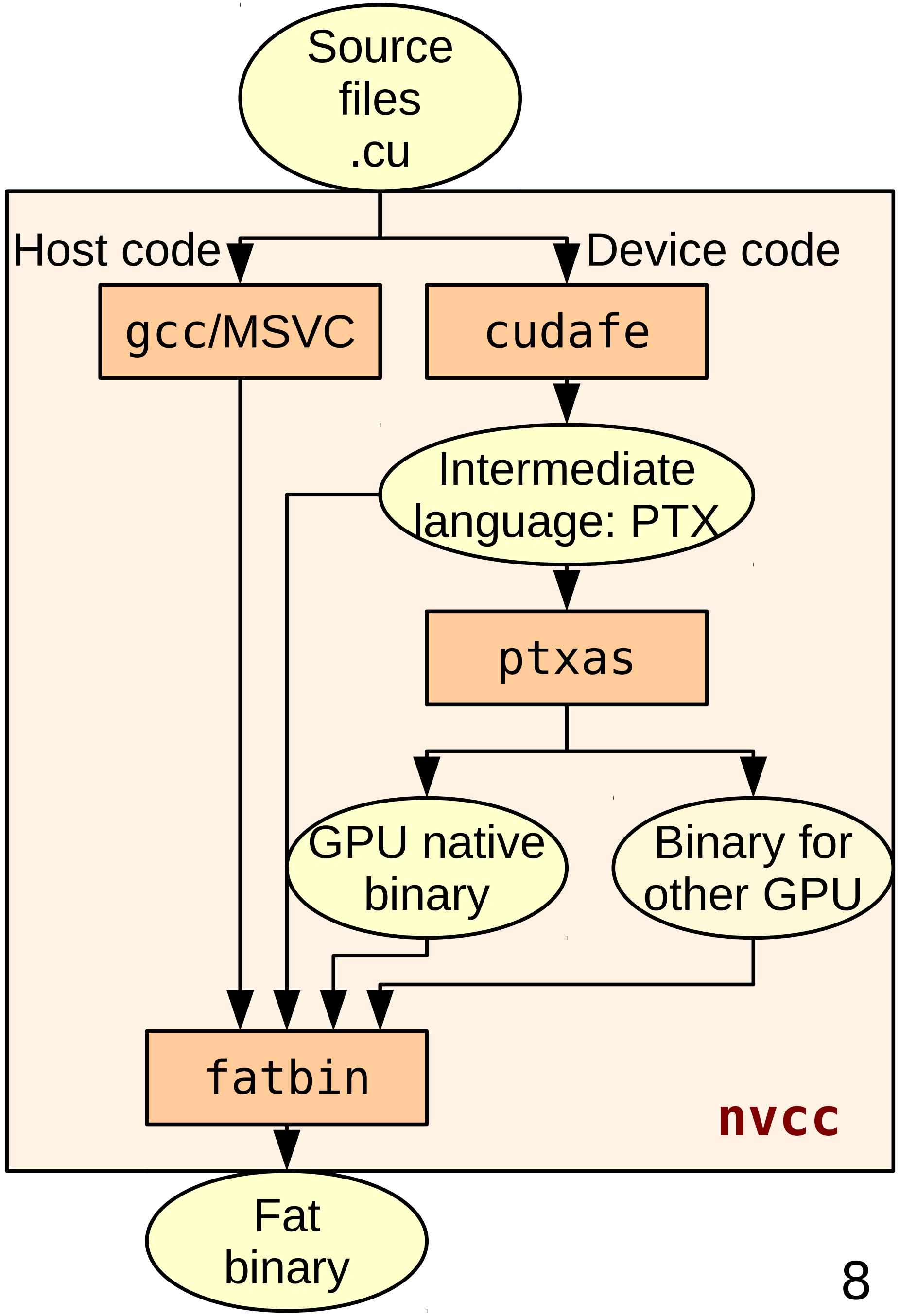


- CUDA C is C++ with a few extra features

# Compiling a CUDA program

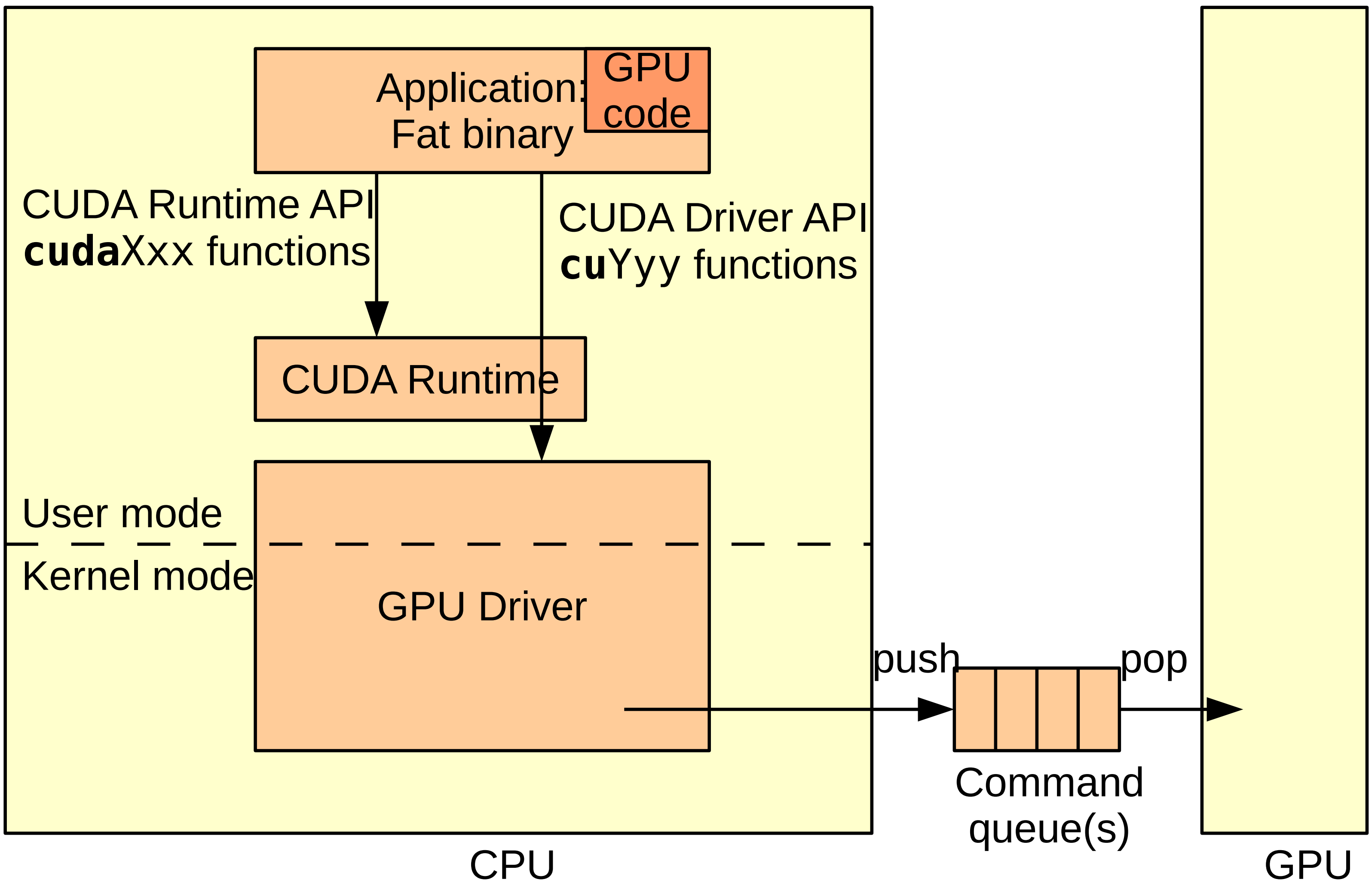
- Executable contains both host and device code
  - Device code in PTX and/or native
  - PTX can be recompiled on the fly
    (e.g. old program on new GPU)
- NVIDIA's compiler driver takes care of the process:

```
nvcc -o hello hello.cu
```



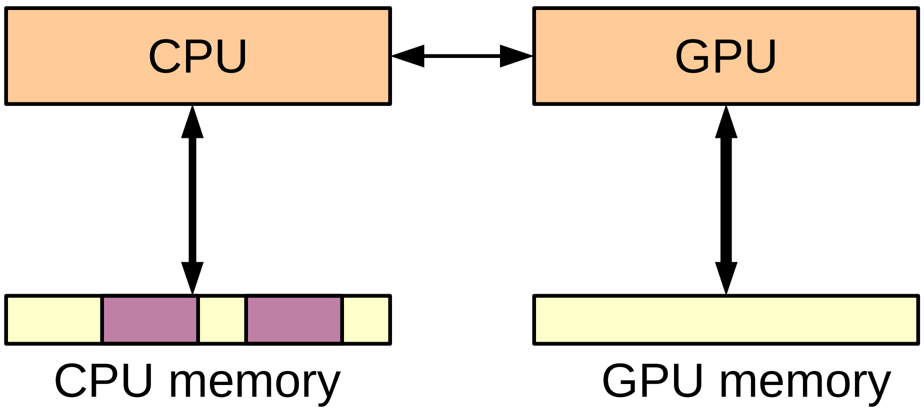
# Control flow

- Program running on CPUs
- Submit work to the GPU through the GPU driver
- Commands execute asynchronously



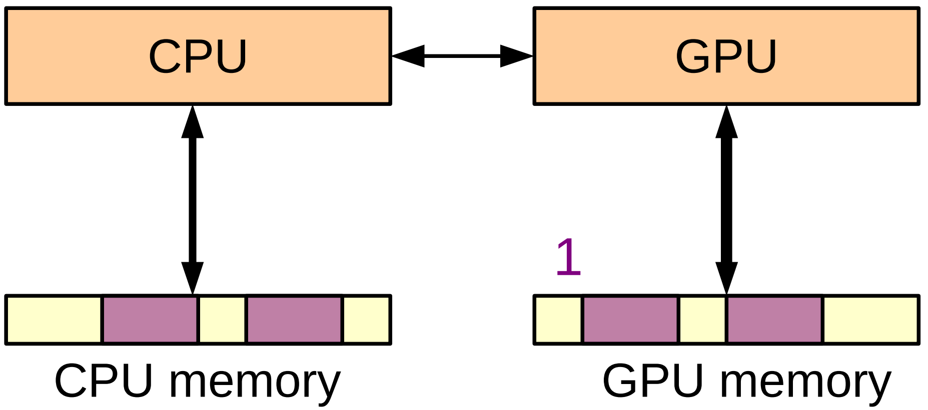
# Data flow

- Main program runs on the host
  - Manages memory transfers
  - Initiate work on GPU
- Typical flow



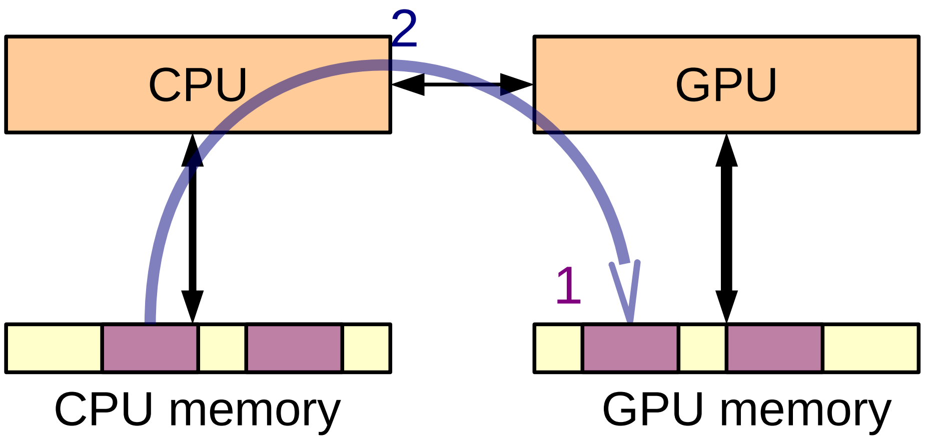
# Data flow

- Main program runs on the host
  - Manages memory transfers
  - Initiate work on GPU
- Typical flow
  - **1.** Allocate GPU memory



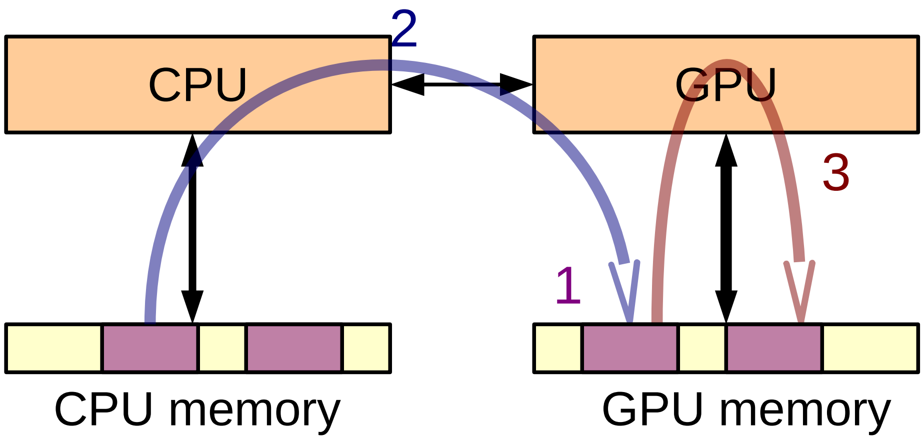
# Data flow

- Main program runs on the host
  - Manages memory transfers
  - Initiate work on GPU
- Typical flow
  - **1.** Allocate GPU memory
  - **2.** Copy inputs from CPU mem to GPU memory



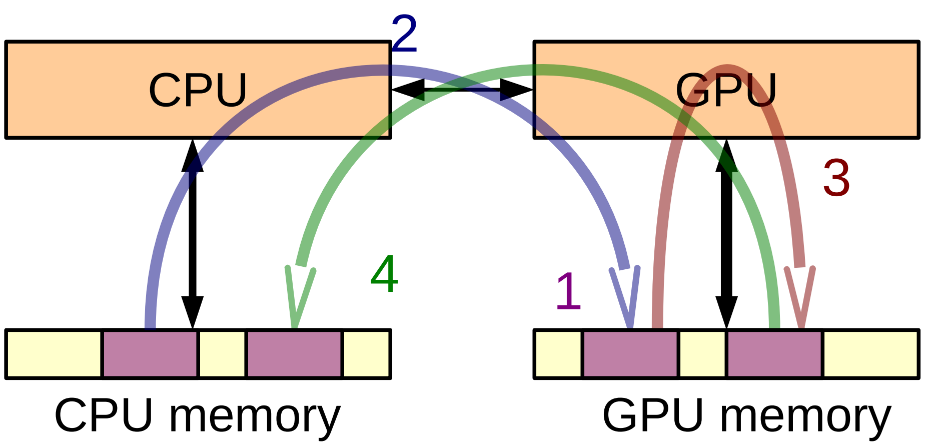
# Data flow

- Main program runs on the host
  - Manages memory transfers
  - Initiate work on GPU
- Typical flow
  - **1.** Allocate GPU memory
  - **2.** Copy inputs from CPU mem to GPU memory
  - **3.** Run computation on GPU



# Data flow

- Main program runs on the host
  - Manages memory transfers
  - Initiate work on GPU
- Typical flow
  - **1.** Allocate GPU memory
  - **2.** Copy inputs from CPU mem to GPU memory
  - **3.** Run computation on GPU
  - **4.** Copy back results to CPU memory



# Example: a + b

- Our Hello World example did not involve the GPU
- Let's add up 2 numbers on the GPU
- Start from host code

```
int main()
{
    float ab[2] = {1515, 149};   // Inputs

    float c[1]; // Output
    // c[0] = ab[0] + ab[1];
    printf("c = %f\n", c[0]);
}
```

vectorAdd example: cuda/samples/0_Simple/vectorAdd

# Step 1: allocate GPU memory

```
int main()
{
    float ab[2] = {1515, 149}, c[1];  // Inputs, in host mem

    // Allocate GPU memory
    float *d_AB, *d_C;
    cudaMalloc((void **)&d_AB, 2*sizeof(float));
    cudaMalloc((void **)&d_C, sizeof(float));
```

Passing a pointer to the pointer to be overwritten

```
    // Free GPU memory
    cudaFree(d_AB);
    cudaFree(d_C);
}
```

- Allocate space for a, b and c in GPU memory
- At the end, free memory

# Step 2, 4: copy data to/from GPU memory

```
int main()
{
    float ab[2] = {1515, 149}, c[1];    // Inputs/outputs, CPU mem

    // Allocate GPU memory
    float *d_AB, *d_C;
    cudaMalloc((void **)&d_AB, 2*sizeof(float));
    cudaMalloc((void **)&d_C, sizeof(float));

    // Copy from CPU mem to GPU mem
    cudaMemcpy(d_AB, ab, 2*sizeof(float), cudaMemcpyHostToDevice);

    // Copy results back to CPU mem
    cudaMemcpy(c, d_C, sizeof(float), cudaMemcpyDeviceToHost);
    printf("c = %f\n", c[0]);

    // Free GPU memory
    cudaFree(d_AB);
    cudaFree(d_C);
}
```

# Step 3: launch kernel

```
__global__ void addOnGPU(float * ab, float * c)
{
    c[0] = ab[0] + ab[1];
}
```

```
int main()
{
    float ab[] = {1515, 159};     // Inputs, CPU mem

    // Allocate GPU memory
    float *d_AB, *d_C;
    cudaMalloc((void **)&d_AB, 2*sizeof(float));
    cudaMalloc((void **)&d_C, sizeof(float));
    // Copy from CPU mem to GPU mem
    cudaMemcpy(d_AB, ab, 2*sizeof(float), cudaMemcpyHostToDevice);


    // Launch computation on GPU
    addOnGPU<<<1, 1>>>(d_AB, d_C);

    float c[1];     // Result on CPU

    // Copy results back to CPU mem
    cudaMemcpy(c, d_C, sizeof(float), cudaMemcpyDeviceToHost);
    printf("c = %f\n", c[0]);
    // Free GPU memory
    cudaFree(d_AB);
    cudaFree(d_C);
}
```

- Kernel is a function prefixed by `__global__`
  - Runs on GPU
- Invoked from CPU code with **<<<>>>** syntax

Note: we could have passed
a and b directly
as kernel parameters

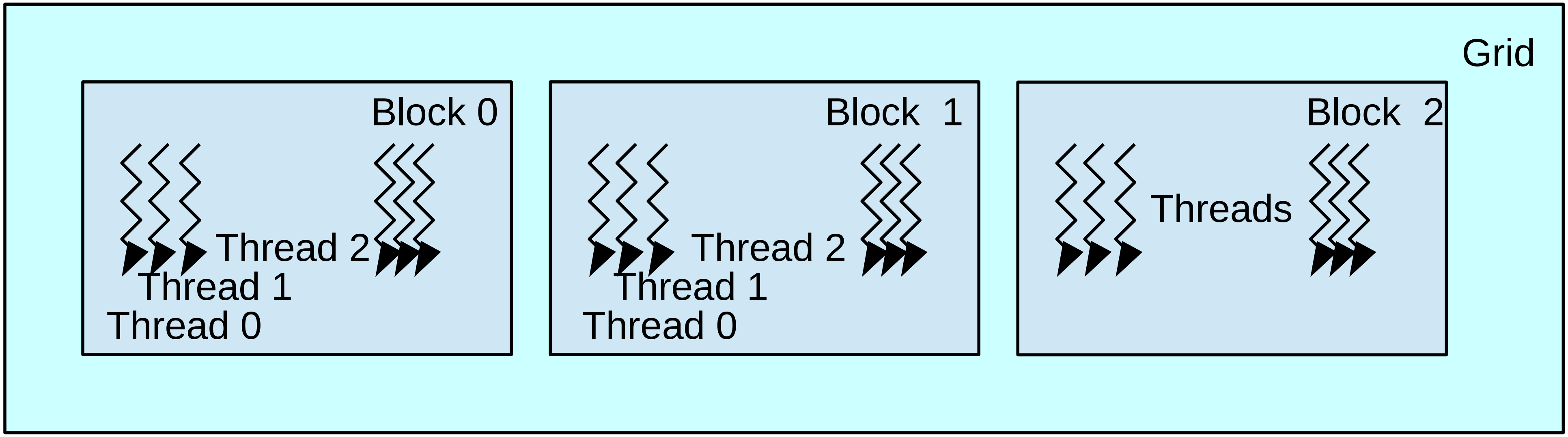
What is inside the <<<>>>?

# Outline

- GPU programming environments
- CUDA host side
- CUDA device side: threads, blocks, grids
- Expressing parallelism
  - Vector add example
- Managing communications
  - Parallel reduction example
- Re-using data
  - Matrix multiplication example

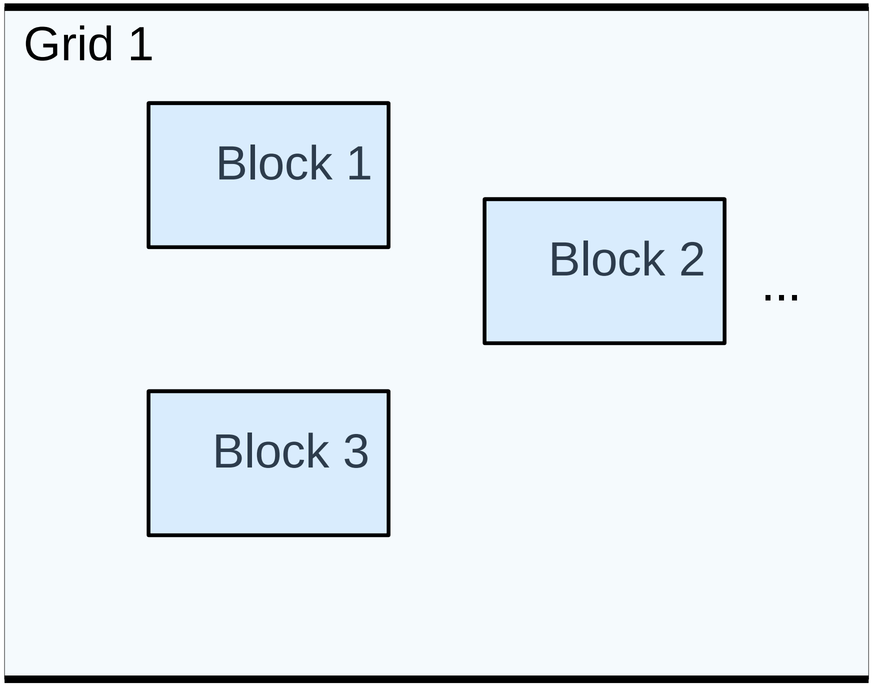
# Workload: logical organization

- A kernel is launch on a grid: `my_kernel<<<blocks, threads>>>(...)`
- Two nested levels
  - Blocks
  - Threads



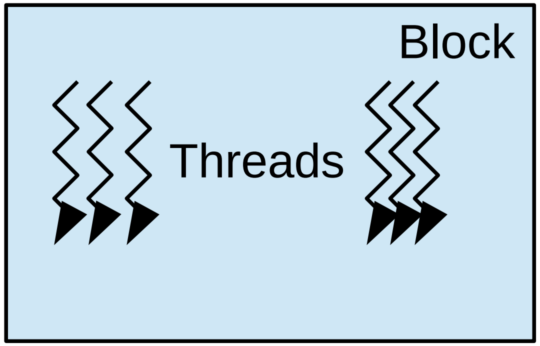
# Outer level: grid of blocks

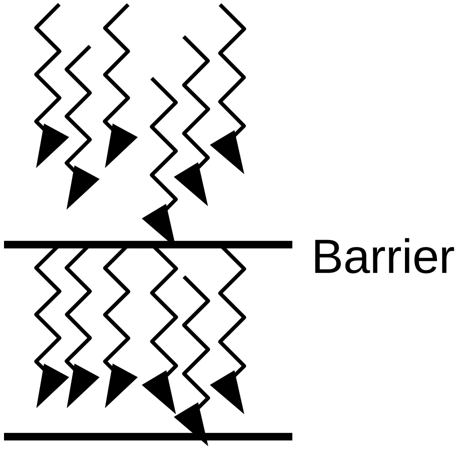
- Blocks also named Concurrent Thread Arrays (CTAs)
- **No communication** between blocks of the same grid
- Practically **unlimited number** of blocks / grid



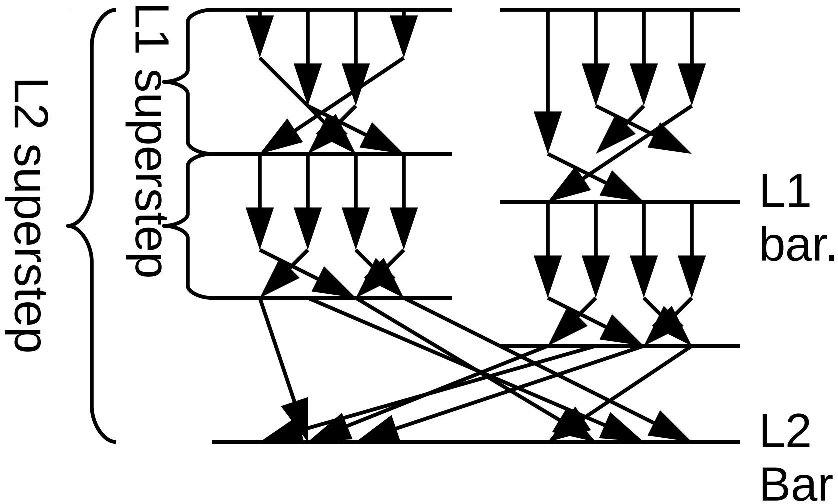
# Inner level: threads

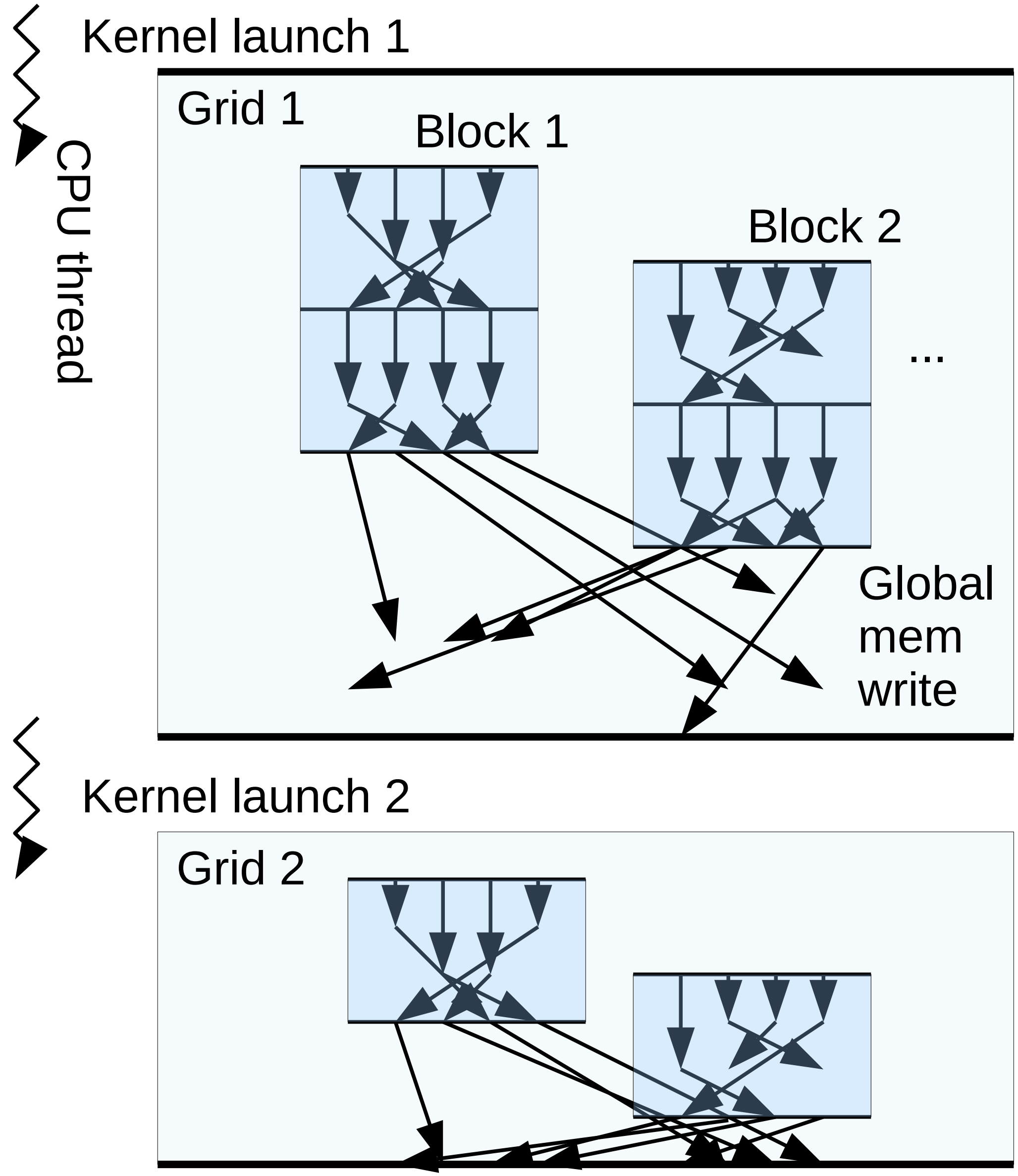
- Blocks contain threads
- All threads in a block
  - Run on the same SM:
    they can **communicate**
  - Run in parallel:
    they can **synchronize**
- Constraints on
  **number of threads / block**
  - Maximum:
    512 to 1024 depending on arch
  - Recommended:
    at least 64 threads for good performance
  - Recommended:
    multiple of the warp size (32)



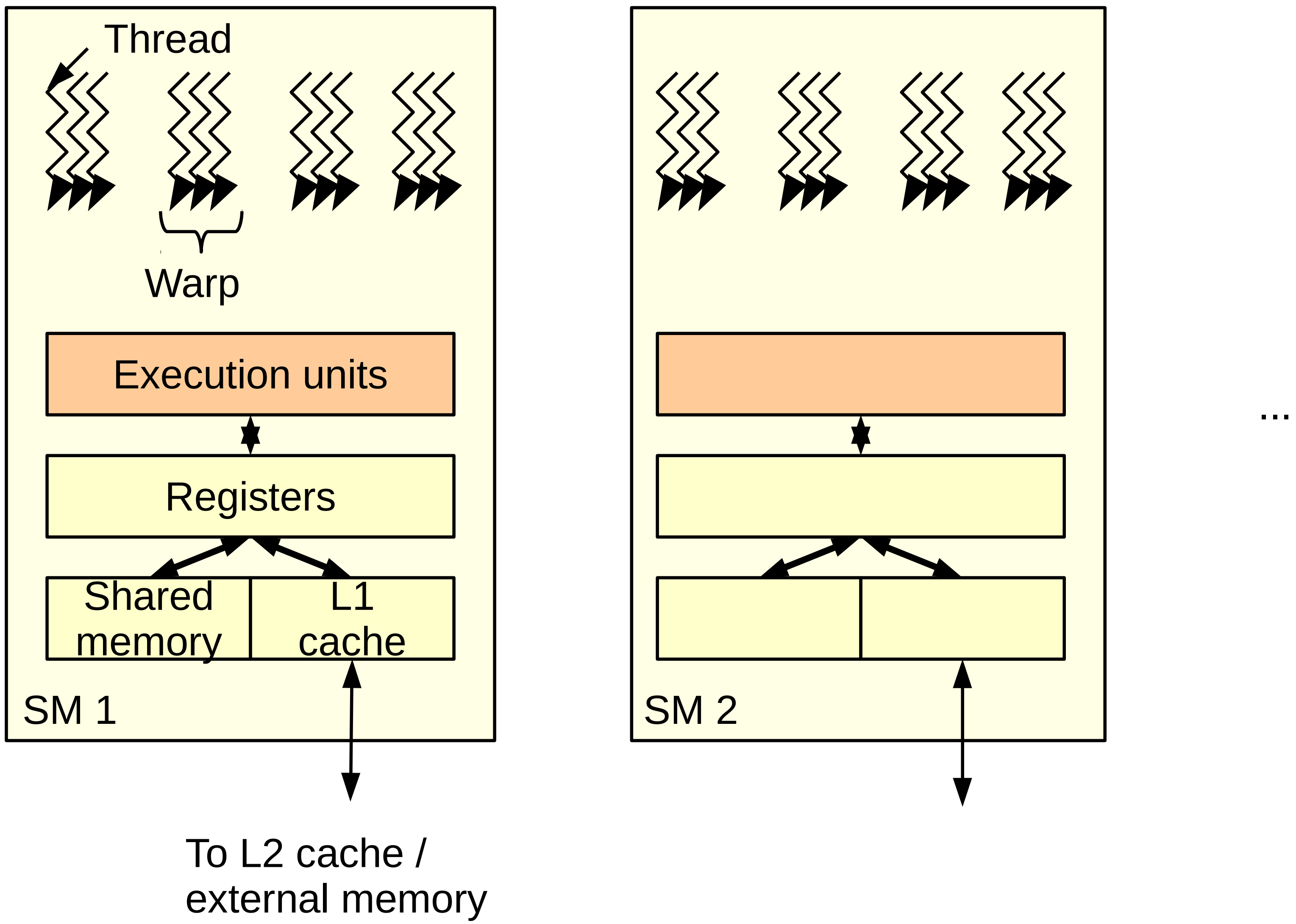
# Multi-BSP and CUDA



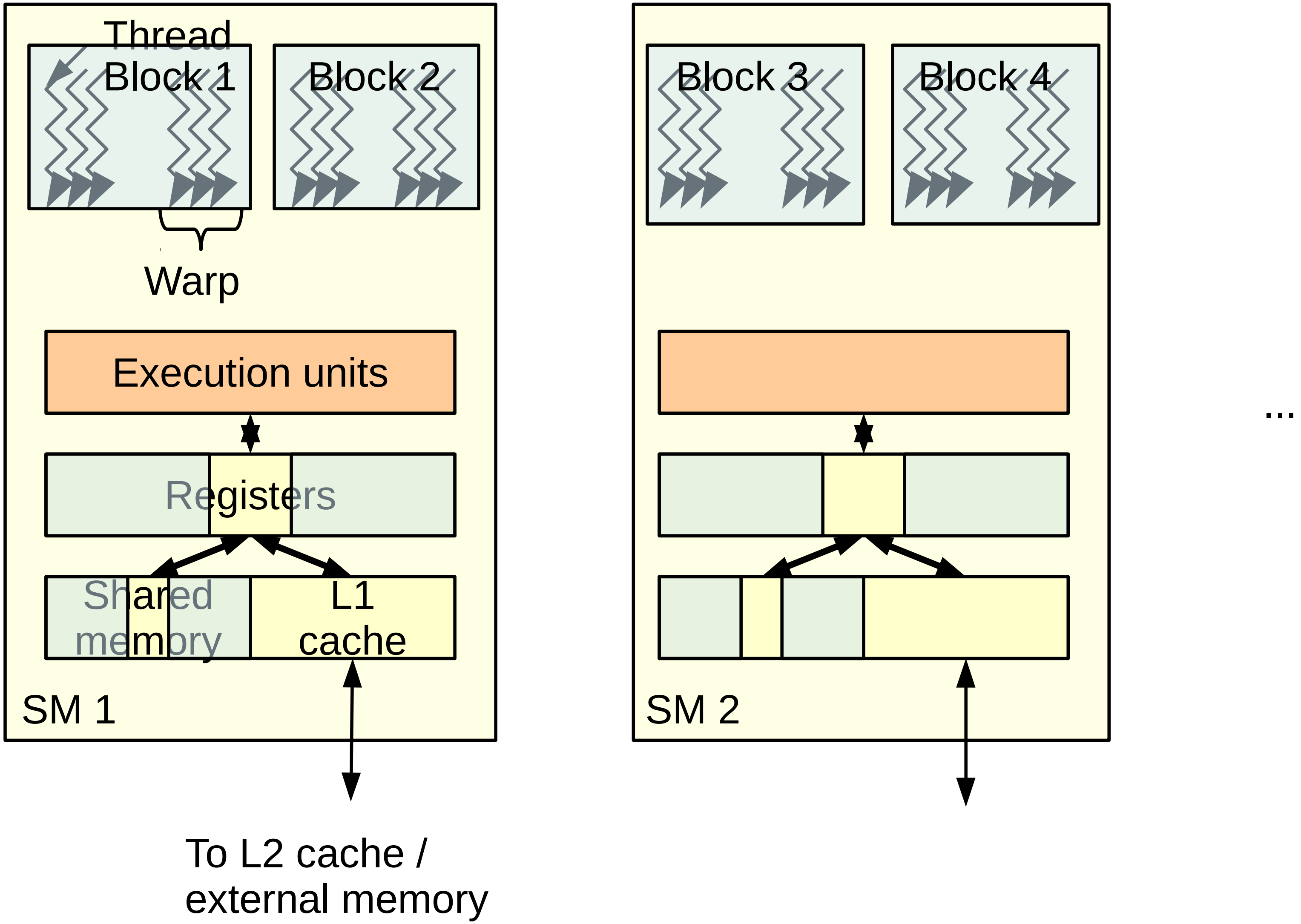




Minor difference: BSP is based on message passing, CUDA on shared memory

# GPU physical organization



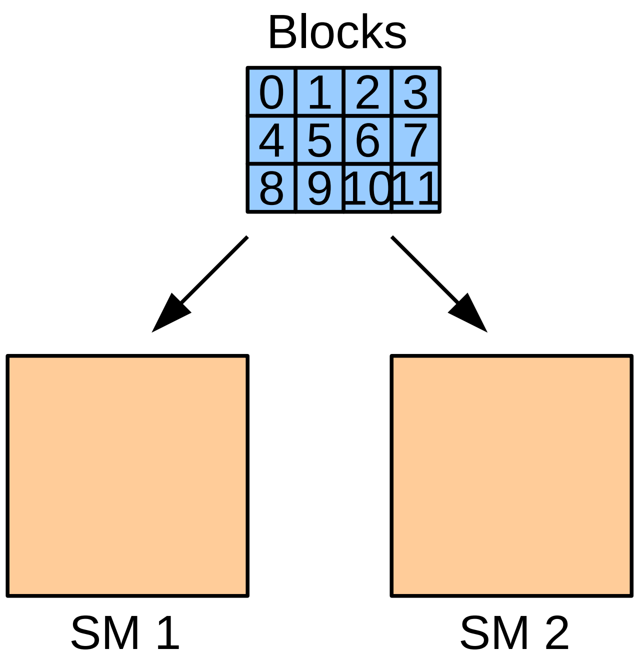
# Mapping blocks to hardware resources




- SM resources are partitioned across blocks

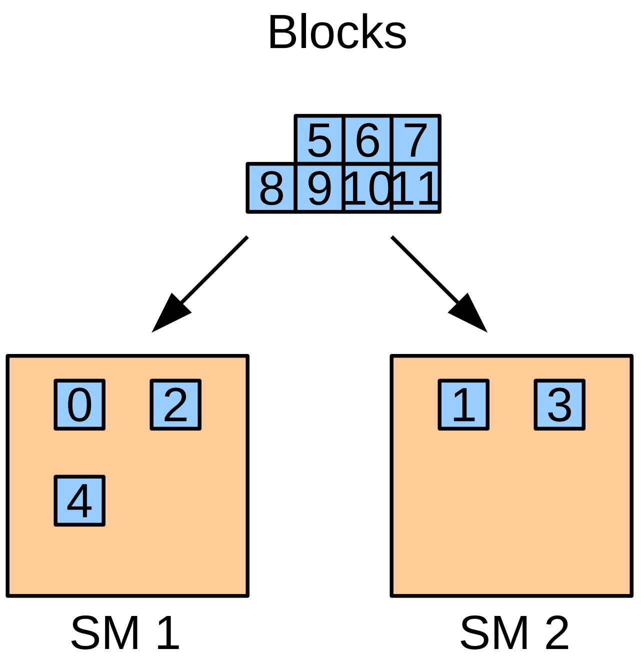
# Block scheduling

- Blocks may
  - Run serially or in parallel
  - Run on the same or different SM
  - Run in order or out of order
- Should not assume anything on execution order of blocks



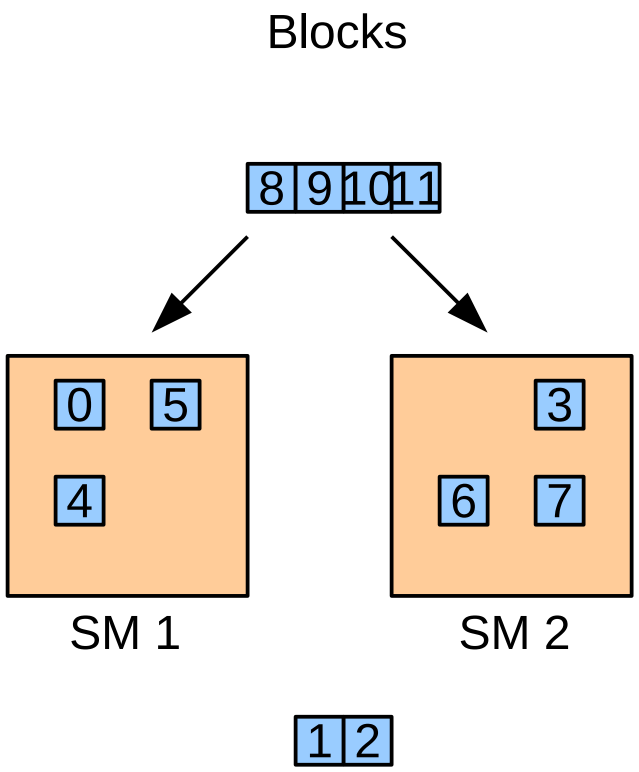
# Block scheduling

- Blocks may
  - Run serially or in parallel
  - Run on the same or different SM
  - Run in order or out of order
- Should not assume anything on execution order of blocks



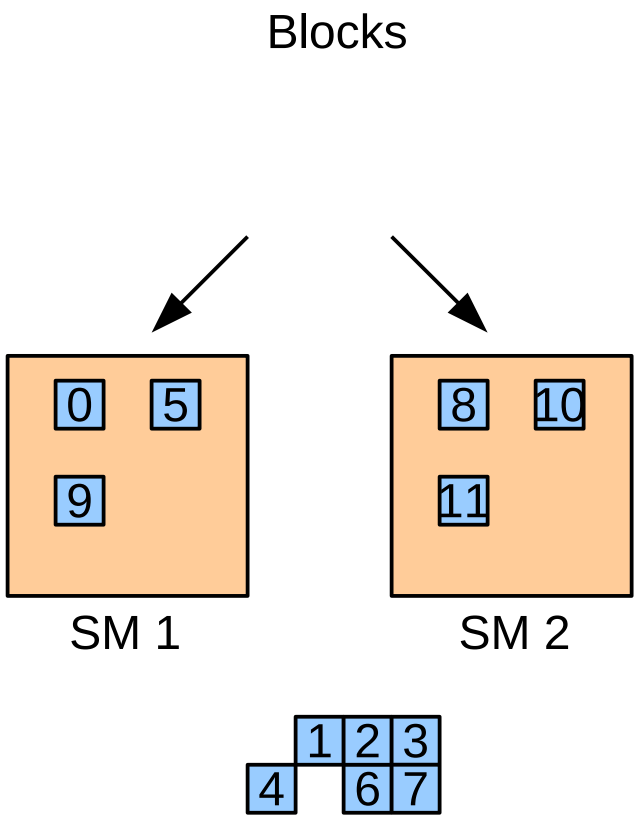
# Block scheduling

- Blocks may
  - Run serially or in parallel
  - Run on the same or different SM
  - Run in order or out of order
- Should not assume anything on execution order of blocks



# Block scheduling

- Blocks may
  - Run serially or in parallel
  - Run on the same or different SM
  - Run in order or out of order
- Should not assume anything on execution order of blocks



# Outline

- GPU programming environments
- CUDA host side
- CUDA device side: threads, blocks, grids
- Expressing parallelism
  - Vector add example
- Managing communications
  - Parallel reduction example
- Re-using data
  - Matrix multiplication example

# Example: vector addition

- Addition example: only 1 thread
    - Now let's run a parallel computation
- Start with multiple blocks, 1 thread/block
    - Independent computations in each block
- No communication/synchronization needed

# Host code: initialization

- A and B are now arrays: just change allocation size

```
int main()
{
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);
    Initialize(h_A, h_B);

    // Allocate device memory
    float *d_A, *d_B, *d_C;
    cudaMalloc((void **)&d_A, size);
    cudaMalloc((void **)&d_B, size);
    cudaMalloc((void **)&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
   ...
}
```

# Host code: kernel and kernel launch

```
__global__ void vectorAdd2(float *A, float *B, float *C)
{
    int i = blockIdx.x;

    C[i] = A[i] + B[i];
}
```

- Launch n blocks of 1 thread each (for now)

```
int blocks = numElements;
vectorAdd2<<<blocks, 1>>>(d_A, d_B, d_C);
```

# Device code

```
__global__ void vectorAdd2(float *A, float *B, float *C)
{
    int i = blockIdx.x;

    C[i] = A[i] + B[i];
}
```

Built-in CUDA variable:
in device code only

- Block number *i* processes element *i*
- Grid of blocks may have up to 3 dimensions (`blockIdx.x`, `blockIdx.y`, `blockIdx.z`)
  - For programmer convenience: no effect on scheduling

# Multiple blocks, multiple threads/block

Fixed number of threads / block: here 64

- Host code

Not necessarily multiple of block size!

```
int threads = 64;
int blocks = (numElements + threads - 1) / threads; // Round up

vectorAdd3<<<blocks, threads>>>(d_A, d_B, d_C, numElements);
```

- Device code

```
__global__ void vectorAdd3(const float *A, const float *B, float *C,
    int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if(i < n) {
        C[i] = A[i] + B[i];
    }
}
```

Global index

Last block may have less work to do

Thread block may also have up to 3 dimensions: `threadIdx.{x,y,z}`

# Outline

- GPU programming environments
- CUDA host side
- CUDA device side: threads, blocks, grids
- Expressing parallelism
  - Vector add example
- Managing communications
  - Parallel reduction example
- Re-using data
  - Matrix multiplication example

# Barriers

- Threads can synchronize inside one block
  - Wait until all threads in the block have reached the barrier
- In C for CUDA:

```
__syncthreads();
```

- Needs to be called at the same place
  for all threads of the block

```
if(tid < 5) {
    ...
}
else {
    ...
}
__syncthreads();
```

**Correct**

```
if(a[0] == 17) {
    __syncthreads();
}
else {
    __syncthreads();
}
```

Same condition
for all threads in the block

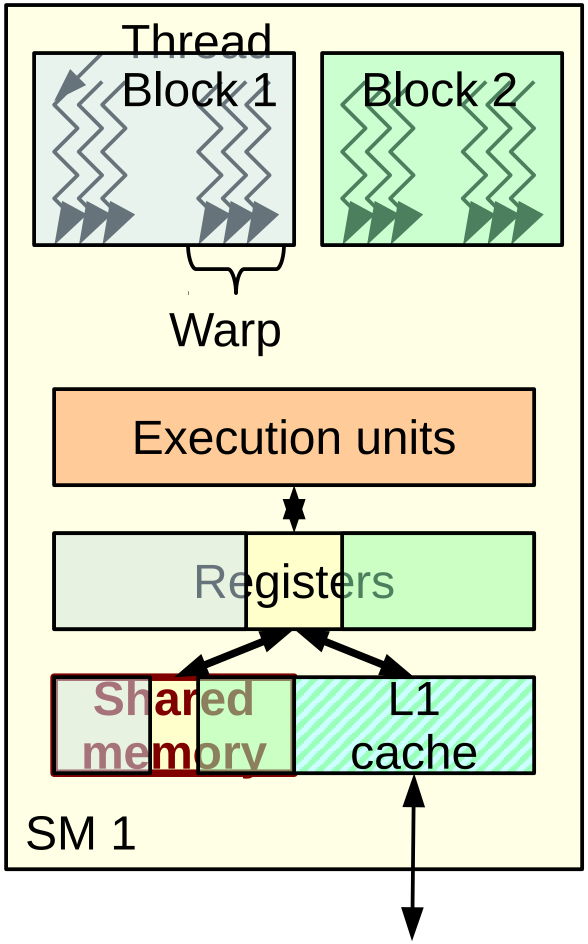
**Correct**

```
if(tid < 5) {
    __syncthreads();
}
else {
    __syncthreads();
}
```

**Wrong**

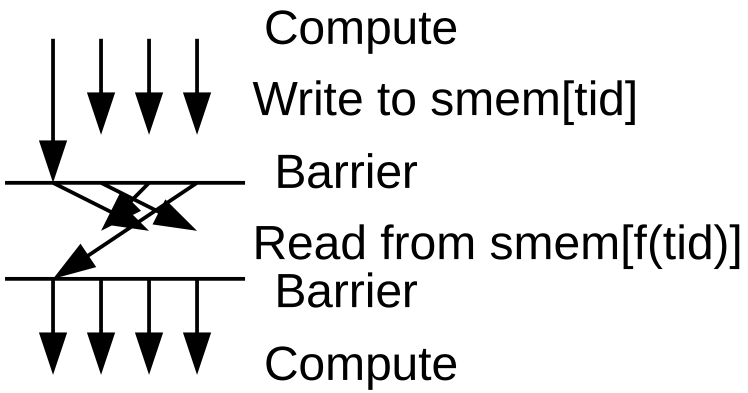
# Shared memory

- Fast, software-managed memory
  - Faster than global memory
- Valid only inside one block
  - Each block sees its own copy
- Used to exchange data between threads
- Concurrent writes:
  one thread wins, but we do not know which one



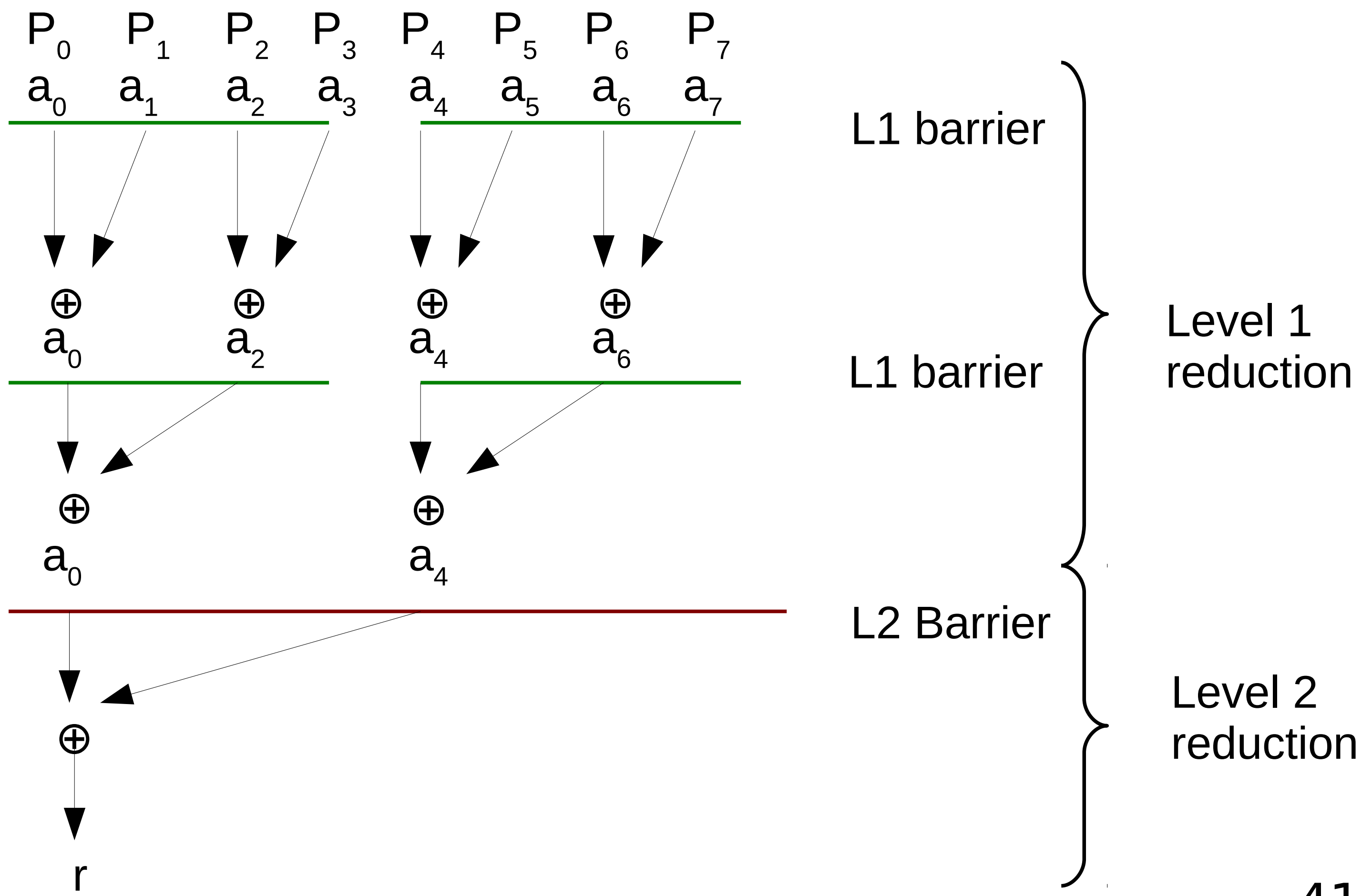
# Thread communication: common pattern

- Each thread writes to its own location
  - No write conflict
- Barrier
  - Wait until all threads have written
- Read data from other threads



# Example: parallel reduction

- Algorithm for 2-level multi-BSP model



# Reduction in CUDA: level 1

```
__global__ void reduce1(float *g_idata, float *g_odata, unsigned int n)
{
    extern __shared__ float sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Load from global to shared mem
    sdata[tid] = (i < n) ? g_idata[i] : 0;
    __syncthreads();

    for(unsigned int s = 1; s < blockDim.x; s *= 2) {
        int index = 2 * s * tid;

        if(index < blockDim.x) {
            sdata[index] += sdata[index + s];
        }
        __syncthreads();
    }

    // Write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Dynamic shared memory allocation: will specify size later

cuda/samples/6_Advanced/reduction

# Quick sanity check to remember

- Each thread block has its own shared memory space
  - If **blockIdx** appears in the calculation of a **shared** memory index, you are probably doing something wrong!

```
__global__ void reduce1(float *g_idata, float *g_odata, unsigned int n)
{
    extern __shared__ float sdata[];

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Load from global to shared mem
    sdata[tid] = (i < n) ? g_idata[i] : 0;
    __syncthreads();

    for(unsigned int s = 1; s < blockDim.x; s *= 2) {
        int index = 2 * s * tid;

        if(index < blockDim.x) {
            sdata[index] += sdata[index + s];
        }
        __syncthreads();
    }

    // Write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

**Global** memory index may depend on **blockIdx** and **threadIdx**

Shared memory index may depend on **threadIdx**, but never on **blockIdx**

# Reduction: host code

```
int smemSize = threads * sizeof(float);
reduce1<<<blocks, threads, smemSize>>>(d_idata, d_odata, size);
```

Optional parameter:
Size of dynamic shared memory per block

- Level 2: run reduction kernel again,
  until we have 1 block left
- By the way, is our reduction operator associative?

# A word on floating-point

- Parallel reduction requires the operator to be **associative**
- Is addition associative?

# A word on floating-point

- Parallel reduction requires the operator to be **associative**
- Is addition associative?
  - On reals: yes, $(a + b) + c = a + (b + c)$
  - On floating-point numbers: **no**
    Example with 4 decimal digits:
    (1.234 + 123.4) - 123.4 = 124.6 - 123.4 = **1.200**
    1.234 + (123.4 - 123.4) = 1.234 + 0 = **1.234**

1. 2 3 4
\+ 1 2 3. 4
1 2 4. 6 | ~~3 4~~

1 2 4. 6
\- 1 2 3. 4
0 0 1. 2 0 0

1 2 3. 4
\- 1 2 3. 4
0 0 0. 0

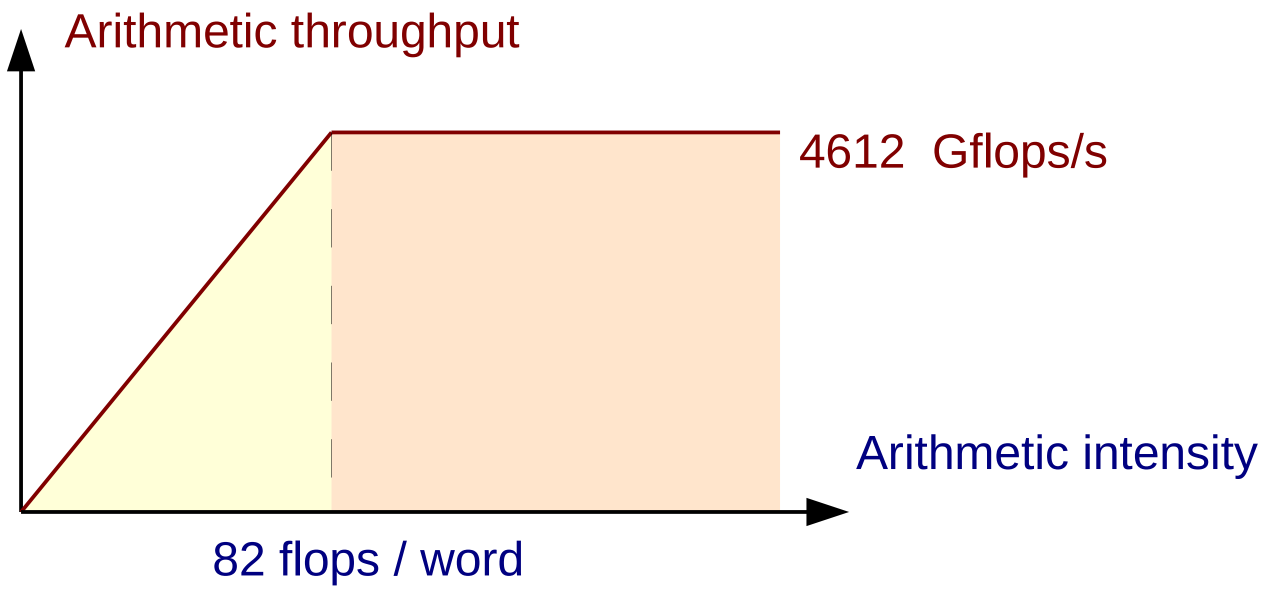
0 0 0. 0
\+ 1. 2 3 4
0 0 1. 2 3 4

- Consequence: different result depending on thread count

# Outline

- GPU programming environments
- CUDA host side
- Parallel programming models: BSP, multi-BSP
- CUDA device side: threads, blocks, grids
- Expressing parallelism
  - Vector add example
- Managing communications
  - Parallel reduction example
- Re-using data
  - Matrix multiplication example

# Arithmetic intensity

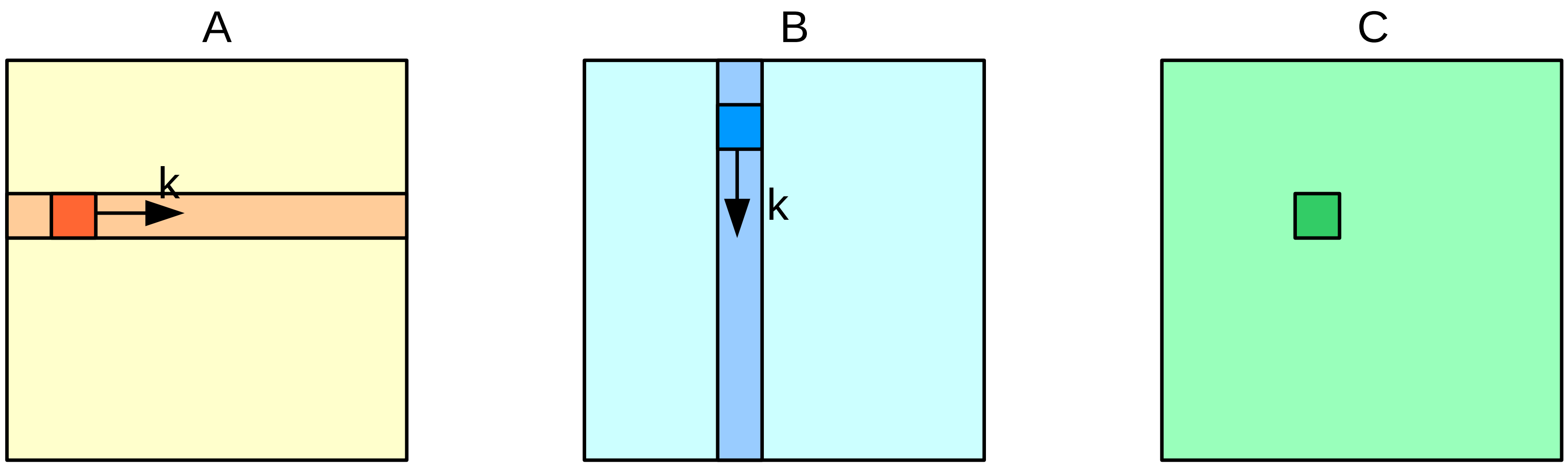



- Example: our GPU
  - NVIDIA GTX 980 needs ≥82 floating-point operations per word from memory to reach peak performance
- How to reach enough arithmetic intensity?
  - Need to **reuse** values loaded from memory

# Classic example: matrix multiplication

- Naive algorithm

```
for i = 0 to n-1
    for j = 0 to n-1
        for k = 0 to n-1
            C[i,j]+=A[i,k]*B[k,j]
```

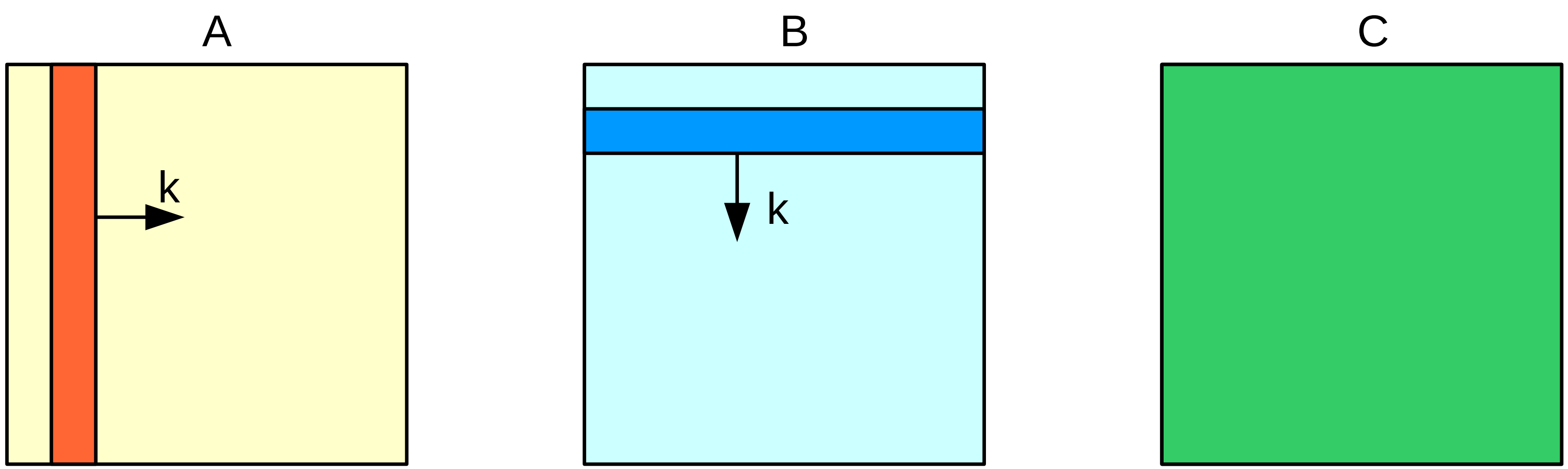



- Arithmetic intensity: 1:1   :(

# Reusing inputs

- Move loop on *k* up

```
for k = 0 to n-1
    for i = 0 to n-1
        for j = 0 to n-1
            C[i,j]+=A[i,k]*B[k,j]
```



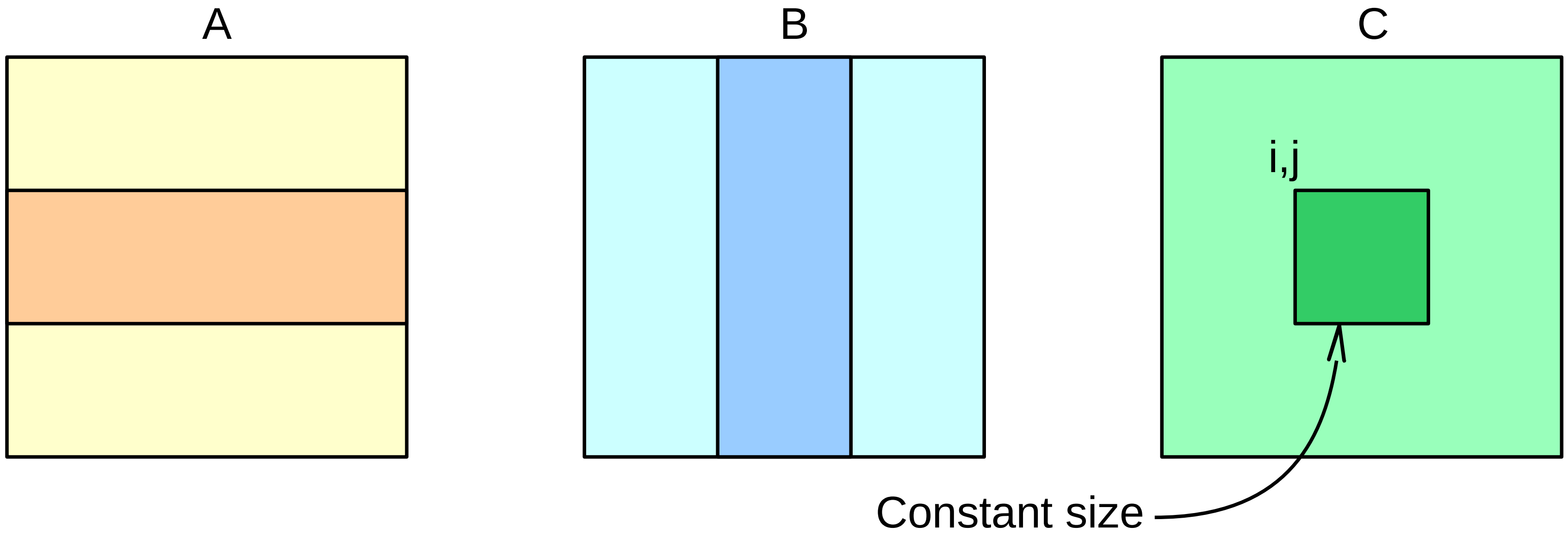

- Enable data reuse on inputs A and B
- But no more reuse on matrix C!

# With tiling

- Block loops on i and j

```
for i = 0 to n-1 step 16
   for j = 0 to n-1 step 16
      for k = 0 to n-1
         for i2 = i to i+15
            for j2 = j to j+15
               C[i2,j2]+=A[i2,k]*B[k,j2]
```

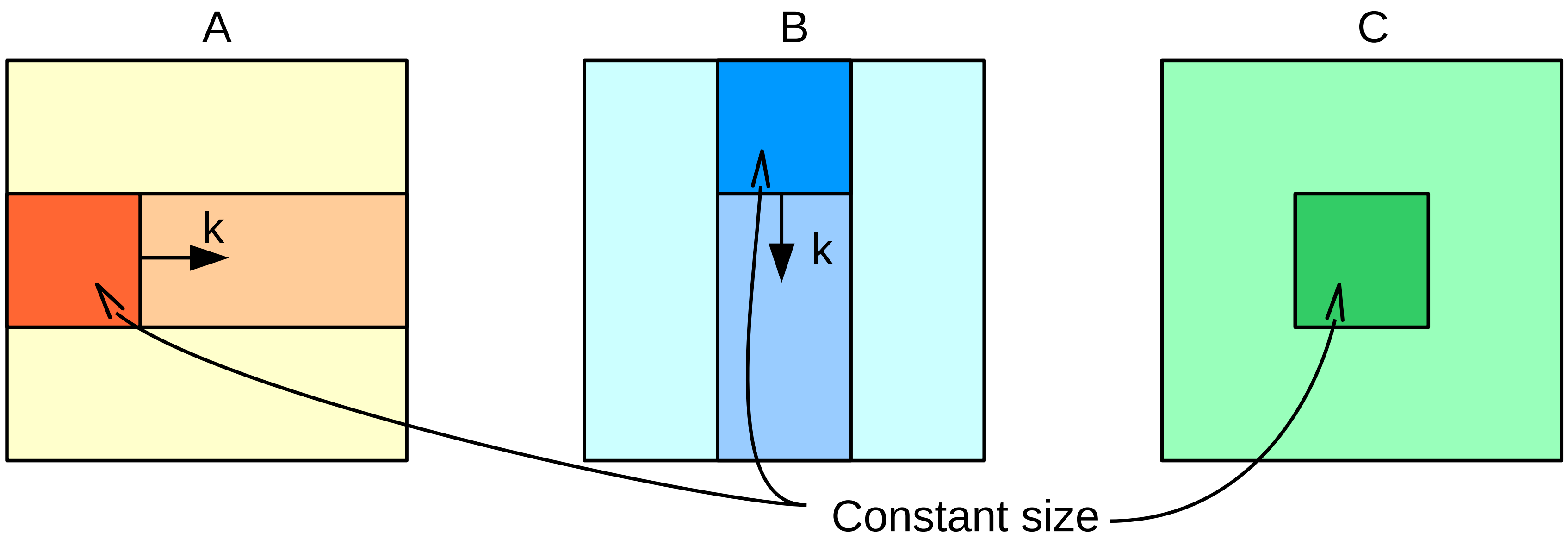
- For one block: product between horizontal panel of A and vertical panel of B



# With more tiling

- Block loop on k

```
for i = 0 to n-1 step 16
    for j = 0 to n-1 step 16
        for k = 0 to n-1 step 16
            for k2 = k to k+15
                for i2 = i to i+15
                    for j2 = j to j+15
                        C[i2,j2]+=A[i2,k]*B[k,j2]
```



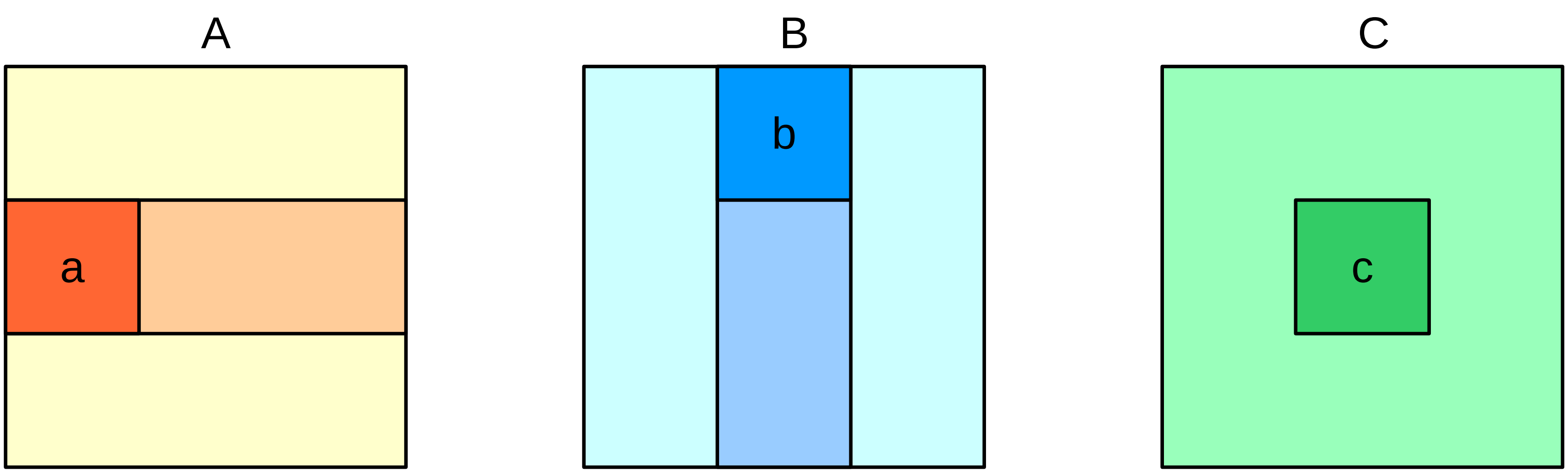
# Pre-loading data

```
for i = 0 to n-1 step 16
    for j = 0 to n-1 step 16
        c = {0}
        for k = 0 to n-1 step 16
            a = A[i..i+15,k..k+15]
            b = B[k..k+15,j..j+15]

            for k2 = 0 to 15
                for i2 = 0 to 15
                    for j2 = 0 to 15
                        c[i2,j2]+=a[i2,k2]*b[k2,j2]

        C[i..i+15,j..j..j+15] = c
```

Load submatrices a and b

Multiply submatrices
c = a × b

Store submatrix c




Arithmetic intensity?

# Breaking into two levels

- Run loops on i, j, i2, j2 in parallel

```
for // i = 0 to n-1 step 16
    for // j = 0 to n-1 step 16
        c = {0}
        for k = 0 to n-1 step 16
            a = A[i..i+15,k..k+15]
            b = B[k..k+15,j..j+15]

            for k2 = 0 to 15
                for // i2 = 0 to 15
                    for // j2 = 0 to 15
                        c[i2,j2]+=a[i2,k2]*b[k2,j2]

        C[i..i+15,j..j+15] = c
```

Level 2:
Blocks

Level 1:
Threads

- Let's focus on threads

# Level 1: SIMD (PRAM-style) version

- Each processor has ID (x,y)
  - Loops on i2, j2 are implicit

```
c[x,y] = 0
for k = 0 to n-1 step 16
    a[x,y] = A[i+x,k+y]
    b[x,y] = B[k+x,j+y]

    for k2 = 0 to 15
        c[x,y]+=a[x,k2]*b[k2,y]

C[i+x,j+y] = c[x,y]
```

Load submatrices a and b

Multiply submatrices
c = a × b

Store submatrix c

Private writes: no conflict

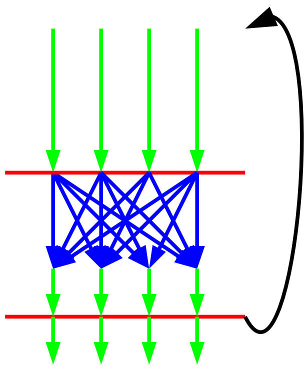
Read from other processors

- How to translate to SPMD (BSP-style) ?

# SPMD version

- Place synchronization barriers

```
c[x,y] = 0
for k = 0 to n-1 step 16
    a[x,y] = A[i+x,k+y]
    b[x,y] = B[k+x,j+y]
    Barrier
    for k2 = 0 to 15
        c[x,y]+=a[x,k2]*b[k2,y]
    Barrier
C[i+x,j+y] = c[x,y]
```



- Why do we need the second barrier ?

# Data allocation

- 3 memory spaces: Global, Shared, Local
  - Where should we put: `A, B, C, a, b, c` ?

```
c[x,y] = 0
for k = 0 to n-1 step 16
    a[x,y] = A[i+x,k+y]
    b[x,y] = B[k+x,j+y]
    Barrier
    for k2 = 0 to 15
        c[x,y] += a[x,k2]*b[k2,y]
    Barrier
C[i+x,j+y] = c[x,y]
```

# Data allocation

- Memory spaces: Global, Shared, Local
  - As local as possible

```
c = 0
for k = 0 to n-1 step 16
    a[x,y] = A[i+x,k+y]
    b[x,y] = B[k+x,j+y]
    Barrier
    for k2 = 0 to 15
        c += a[x,k2]*b[k2,y]
    Barrier
C[i+x,j+y] = c
```

**Local**: private to each thread (indices are implicit)

**Global**: shared between blocks / inputs and outputs

**Shared**: shared between threads, private to block

# CUDA version

- Straightforward translation

```
float Csub = 0;

for(int a = aBegin, b = bBegin;
     a <= aEnd;
     a += aStep, b += bStep) {
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];

    __syncthreads();
    for(int k = 0; k < BLOCK_SIZE; ++k)
    {
        Csub += As[ty][k] * Bs[k][tx];
    }
    __syncthreads();
}

int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
```

Precomputed base addresses

Declare shared memory

Linearized arrays

matrixMul example: cuda/samples/0_Simple/matrixMul

# Local memory

- Registers are fast but
  - Limited in size
  - Not addressable
- Local memory used for
  - Local variables that do not fit in registers (*register spilling*)
  - Local arrays accessed with indirection

```
int a[17];
b = a[i];
```

- **Warning**: local is a misnomer!
  - Physically, local memory usually goes off-chip

# Device functions

- Kernel can call functions
- Need to be marked for GPU compilation

```
__device__ int foo(int i) {
}
```

- A function can be compiled for both host and device

```
__host__ __device__ int bar(int i) {
}
```

- Device functions can call device functions
  - Older GPUs do not support recursion

# Recap

- Memory management: two spaces
  - Host code and memory
  - Device code and memory
- GPU Kernel
  - One function, many threads
- Dimensions of parallelism
  - Grids, Blocks, Threads
- Memory spaces
  - Global, Local, Shared memory
- Next time:
  - GPU architecture
  - Optimizing memory access
  - Advanced features and optimization techniques