# SeSE course: GPU programming for Machine Learning and Data Processing

## GPU programming with CUDA, OpenMP Target, Thrust

# Important computational kernels in deep learning algorithms

- Feedforward model consists of matrix-vector product at each layer, combined with some activation function
- Similarly, the backpropagation for a single value $y$ against measurement $\hat{y}$ passes of a through the various layers one by one, involving matrix-vector products
- Typical models are evaluated (feedforward) or trained (backpropagation) for many sample sets
  - Backpropagation trains for many inputs/output pairs $\left(x^{(1)}, \hat{y}^{(1)}\right)$, $\left(x^{(2)}, \hat{y}^{(2)}\right)$, $\left(x^{(3)}, \hat{y}^{(3)}\right)$, ...
- Concatenate many vectors into a matrix $\rightarrow$ much improved data locality, matrix-vector products transformed in **matrix-matrix multiplications**
- Due to cacheability of matrix-matrix product ($n^2$ data, $n^3$ operations), much better performance possible

# Matrix-matrix multiplication performance-critical

- Central operation of most neural networks in both inference (feedforward) and training (backpropagation) is matrix-matrix multiplication (weight matrix: dense or sparse)
- How to make matrix-matrix multiplication fast?
- Long history in high-performance computing
  - Use of vectorization (SIMD) on CPUs, data re-use in registers most crucial (plus cache blocking for larger sizes)
  - Apply SIMT concept from GPUs
- Observation from practice
  - High precision in weights $w_{ij}$ not needed, result of optimization algorithm with number of layers/sparsity more important than many digits
  - Stochastic gradient descent insensitive to roundoff effects
  - Need only rough information (some digits)
- Idea: Reduce precision from `FP64` (53 bits mantissa) to `FP32` (24 bit mantissa), `FP16` (11 bit mantissa), `BF16` (binary float, exponent range of FP32 but only 8 bit mantissa), or even `INT16` (-32767...32767) or `INT8` (-255...255) in mat-mat

# Specializing hardware for matrix-matrix multiplications

- Given importance of matrix-matrix multiplication in neural network algorithms, want to make it fast
- Possibilities by reducing precision
  - Reduced precision means less data transfer, allowing to transfer bigger "vectors" across a bus of fixed width
  - Multiplication scales quadratically with the number of bits, halving number of bits means we can fit $\sim 4\times$ the multiplications in same transistor budget
- Exploiting theoretical $4\times$ improvements of low-precision multiplications bumps into limits of data transfer between registers and execution units $\rightarrow$ traditional vector style execution only improves by $\sim 2\times$
- Solution: Must perform matrix-matrix multiplication inside **execution unit** to avoid involving storage in registers
  - Instruction for $4 \times 4$, $4 \times 8$, $8 \times 8$ ... matrix multiplications
  - Example $A, B, C \in \mathbb{R}^{4\times 4}$ for `C += A * B`: 48 read accesses, 16 write access versus 128 arithmetic operations; traditional vector-style involves $4\times$ the transfer (from registers), 192 reads and 64 writes

# Hardware for matrix-matrix multiplications

- NVIDIA GPUs: **tensor cores** $\rightarrow$ see Nvidia Ampere whitepaper
  - $2\times$ higher throughput for double precision, $8-16\times$ higher throughput for FP32/FP16, even more for INT8
  - The wider the data, the smaller the size of matrices
- Next-generation CPUs scheduled to include matrix extensions
  - AMX on x86, `https://en.wikichip.org/wiki/x86/amx` to work on tiles, coming in 2022 (?)
  - Scalable matrix extensions for ARM (2022 or later)
  - Apple chips also have "AMX2" unit, not directly accessible, only via frameworks like neural network or BLAS libraries
- Google's own **tensor processing unit**, TPU (8 bit), specific interconnects
- They all target lower-precision computations with AI as main goal
- NVIDIA tensor cores allow sparsity in matrices, given $2\times$ speedup if at most $0.5\times$ matrix is populated, to reflect sparse weights in convolutional networks

# Lecture goal

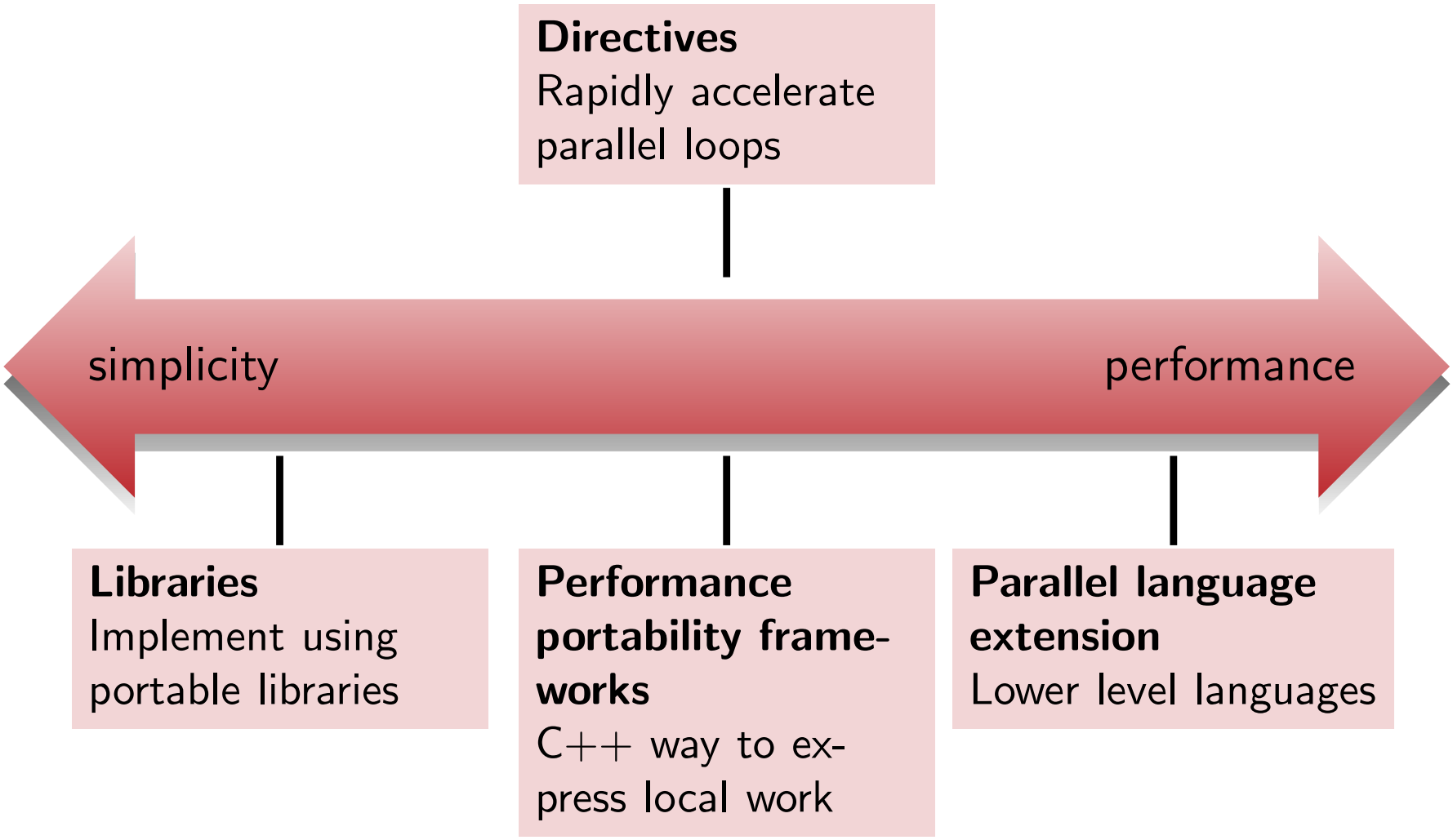


- GPU hardware makes algorithms run fast
- Learn more of the underlying principles
- Not directly tied to AI

# The GPU software-hardware interface

# Using a GPU

- Specific hardware architecture of GPUs necessitates problem-adapted codes
- Tradeoff between programming effort and achievable performance



# Writing applications for a GPU

- **Libraries**
  - Small code changes – call an accelerated function for big chunk of work
  - High performance
  - Limited availability, no fine-grained control
  - Example: TensorFlow, MXNet, ...
- **Directives**
  - Annotate loops in existing languages
  - Simple to adapt, but need to touch all relevant "loops"
  - Less performance control
  - Example: OpenMP Target, OpenCL, OpenACC
- **Languages**
  - Maximal performance
  - Low-level interface close to the hardware
  - Time-consuming to develop and maintain
  - Example: CUDA

# GPU parallelism

- Fine-grained data parallelism
- **SIMT** (Single Instruction **Multiple Thread**) execution
  - Many threads execute concurrently
  - Different data elements correspond to different threads
  - Hardware automatically handles thread divergence
- Not the same as SIMD because of multiple register sets, addresses, and flow paths[1]
- Hardware multithreading
  - High number of threads to hide latency
  - Context switching is essentially free

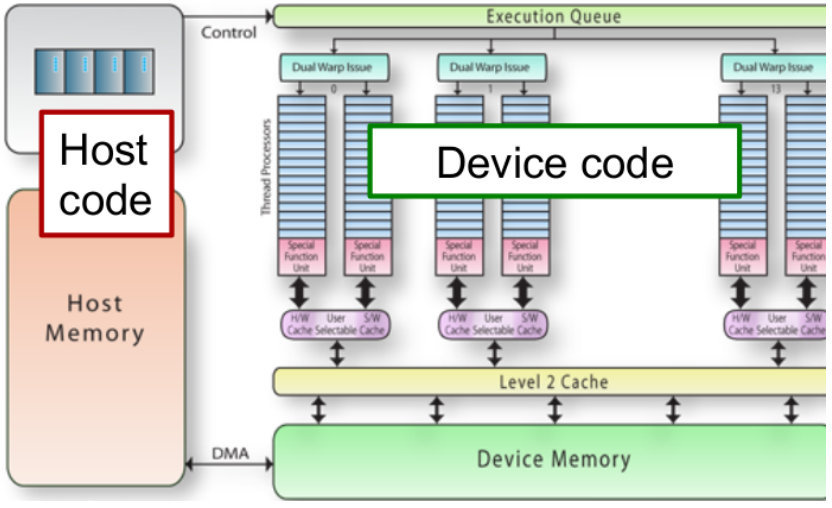
[1] `http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html`

# Parallelization for GPUs

- Parallelization = find a mapping of the problem to the machine model to maximize concurrent execution
- For GPUs with their fine-grained data parallelism
  - Map data and associated work to threads
    - Write the computation **for 1 thread**!
  - Organize threads in blocks and blocks in grids
  - Let the hardware scheduler do the rest
- Assumption: Work is expressed in terms of loops with little or no data dependencies between items

# GPU program organization

- Two types of code:
  - **Device code** = GPU code = kernel(s)
    - A kernel is a sequential program
    - Write for 1 thread, execute for all
  - **Host code** = CPU code
    - Instantiate the "grid" and run the kernel
    - Memory allocation, management, deallocation
    - C, C++, Java, Python, ...
- Host-device communication
  - Explicit or implicit
  - Hardware side:
    - Via PCI/e
    - Via NVLink (if available)



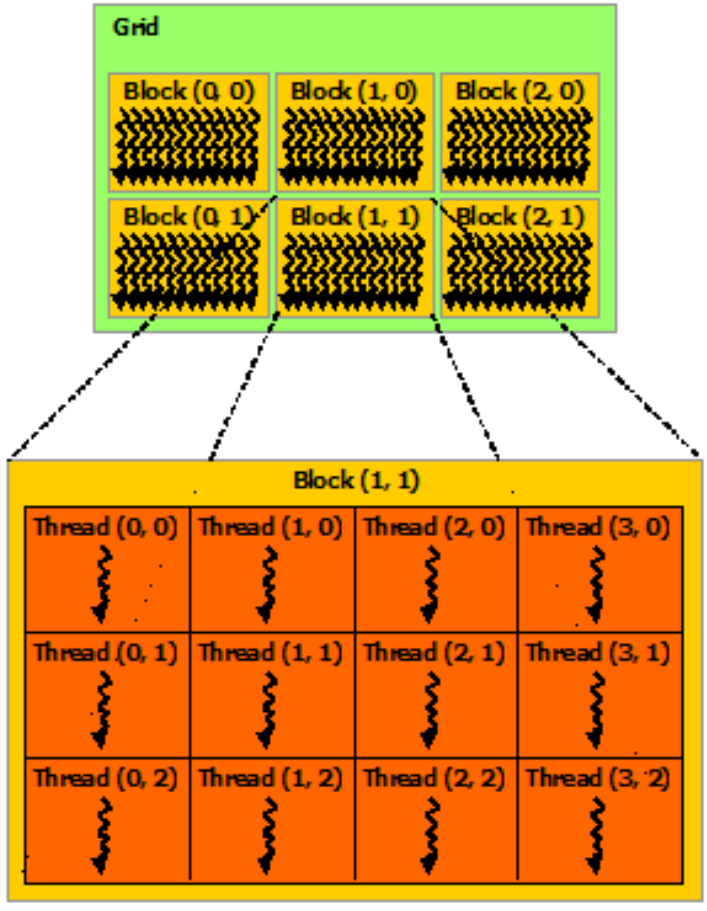
# Thread model on a GPU

**Thread hierarchy:**

- *Threads* execute a *kernel*
- Threads grouped into *blocks*
- Threads in a block run together
- Blocks organized in a *grid*
- Origin of these names: computer graphics of 2D image

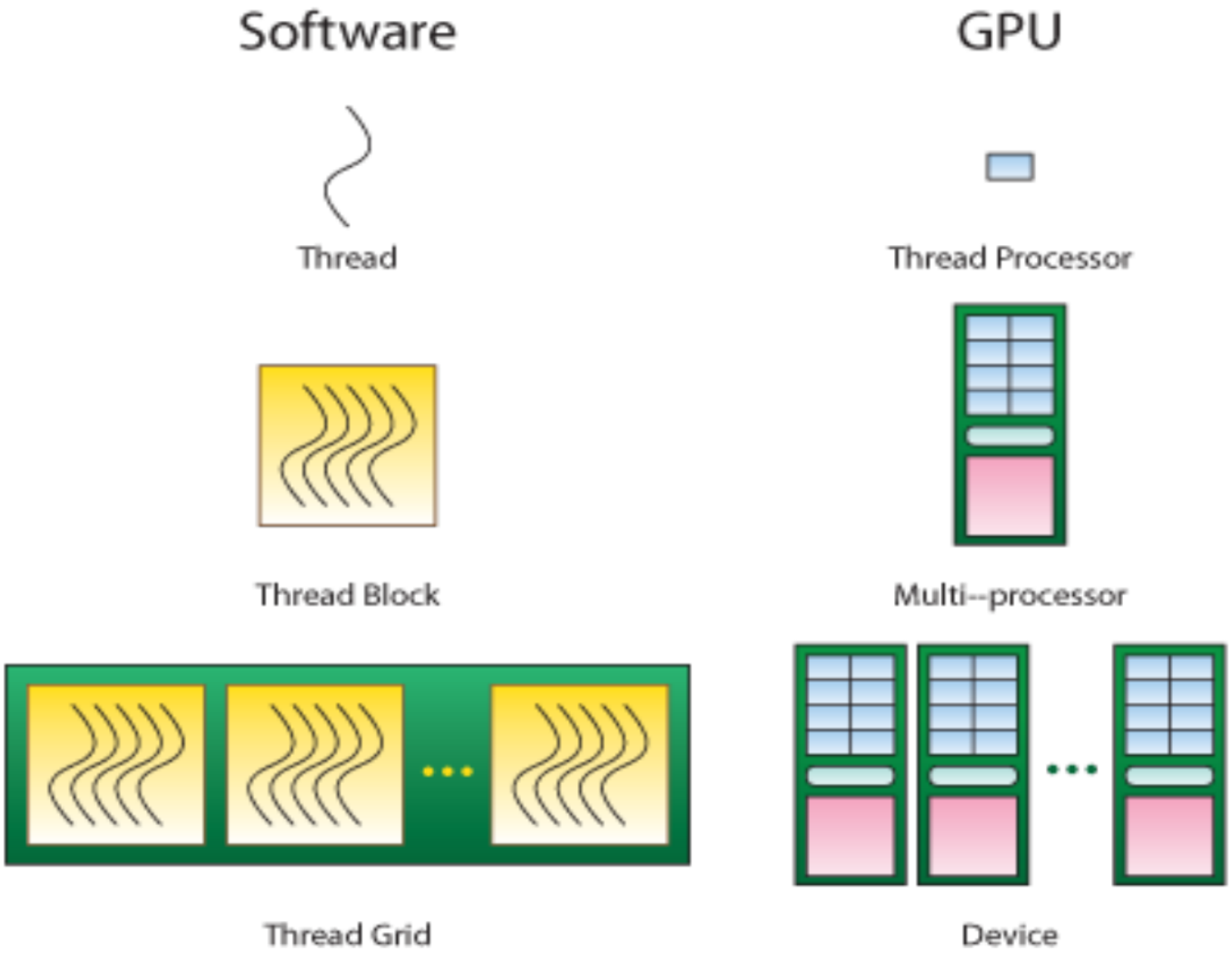
**Block size:**

- Configurable
- Optimum depends on application and hardware
- Often, large blocks are better
- Typically, $\sim$ 256–1024 threads/block



# Model of parallelism vs GPU hardware

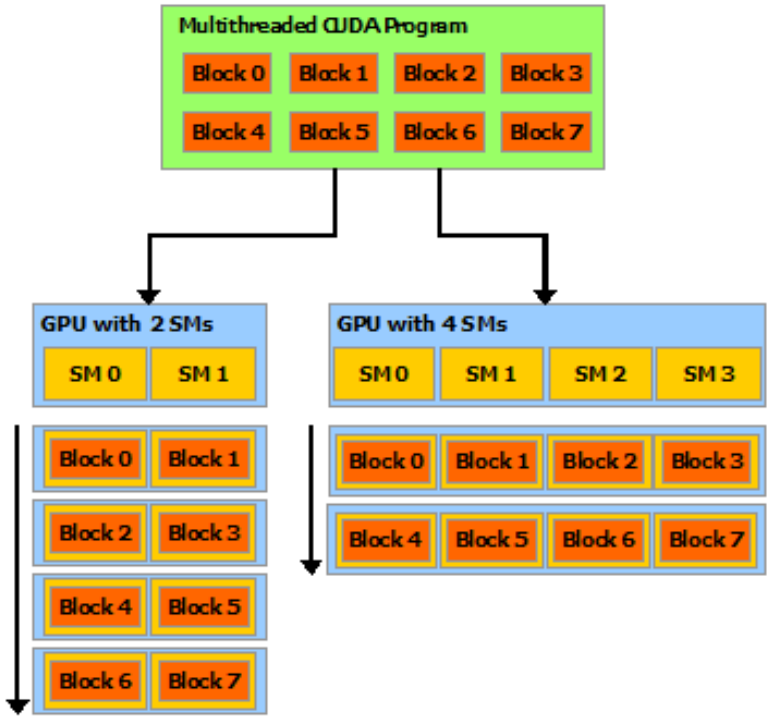




# Scheduling threads to the GPU hardware

**At runtime:**

- Blocks assigned to SMs
- Many blocks ensure good utilization of hardware and hence scalability




**Warps:**

- Smaller subdivision of a block
- One *warp* = 32 threads
- Threads in warp executed in *SIMD* (Single Instruction Multiple Data) fashion

**The hardware schedules warps:**

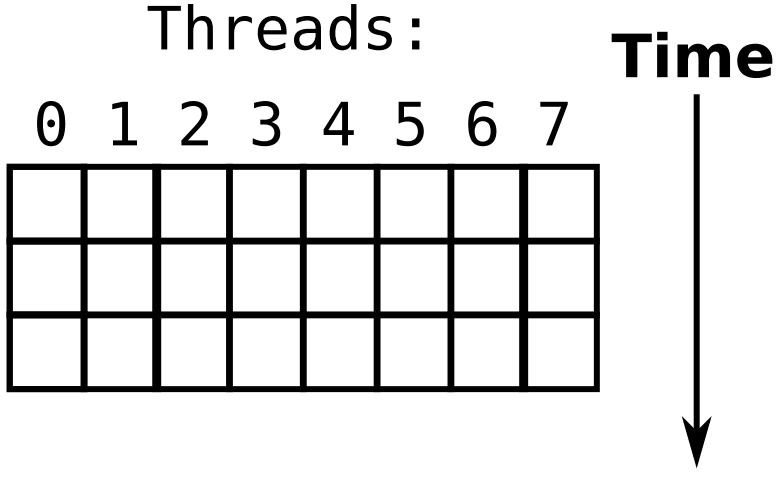
- Want high number of active warps, or *occupancy*
- Usage of resources (registers, shared memory, etc) limits occupancy
- Not handled explicitly when programming, but can matter for performance

**Branch divergence:**

- Threads in warp execute simultaneously on the SM
- Must execute the same instruction
- Branch divergence can hurt performance

**Example:**

```
if(cond(thrid)):
  do A
else: do B
```



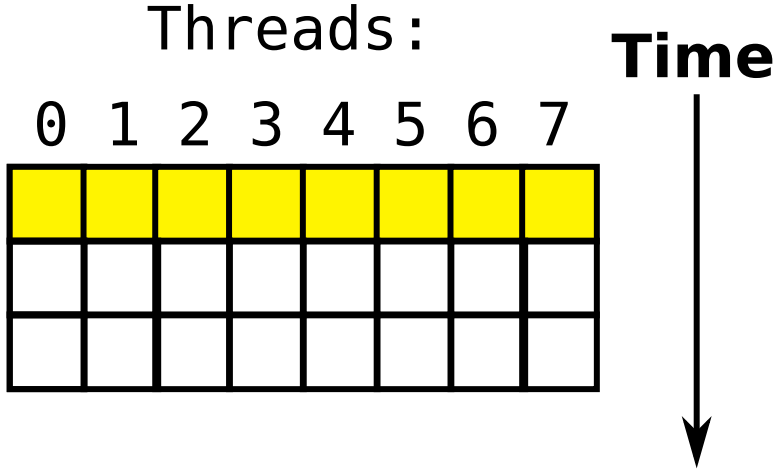
# Thread model

**Branch divergence:**

- Threads in warp execute simultaneously on the SM
- Must execute the same instruction
- Branch divergence can hurt performance

**Example:**

```
if(cond(thrid)):
  do A
else: do B
```



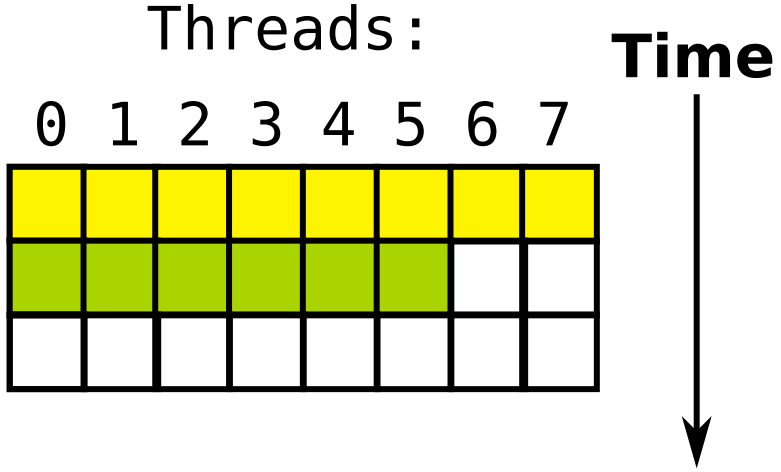
# Thread model

**Branch divergence:**

- Threads in warp execute simultaneously on the SM
- Must execute the same instruction
- Branch divergence can hurt performance

**Example:**

```
if(cond(thrid)):
  do A
else: do B
```

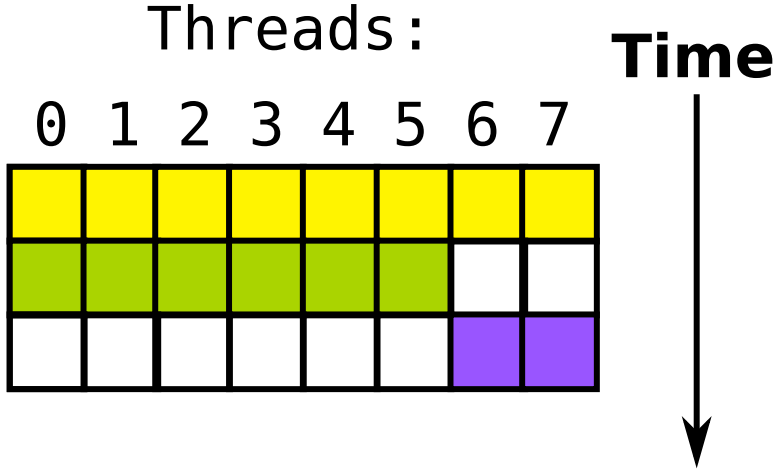


**Branch divergence:**

- Threads in warp execute simultaneously on the SM
- Must execute the same instruction
- Branch divergence can hurt performance

**Example:**

```
if(cond(thrid)):
  do A
else: do B
```




- *75% efficiency loss!*

**Note:** Threads in the same block but different warps do not have this problem.

# Frameworks considered in course: CUDA, OpenMP Target, Thrust

**OpenMP Target**

- Open standard
- Similar to OpenCL with cross platform aim
  - GPUs, multicores, FPGA, etc
- Low-level (high performance)

**Why CUDA?**

- OpenMP must be tuned to perform
- CUDA performs better
- Not as mature
- Messy to program

**They are similar:**

- Very similar programming models
- Can easily convert $\sim$ CUDA $\Leftrightarrow$ OpenMP Target

**Thrust:**

- Higher-level framework

# Programming with CUDA

**Compute Unified Device Architecture**

- Dedicated framework for GPU programming
- Programming language (C/C++ extension)
- Hardware and thread model
- Development toolkit

**Pros/Cons**

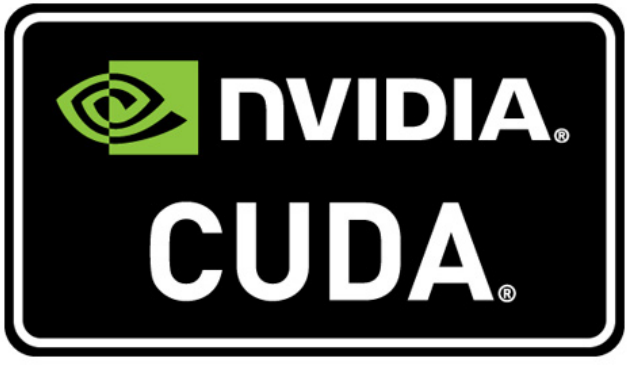
++ Low-level (high performance)

\- Low-level (hard to program)

\+ Mature (debugger, profiler, ...)

\- Nvidia only

\+ Portable across Nvidia GPUs



# First CUDA code

**Vector addition: $x := x + y$**

**CPU function:**

```
void vec_add(int N, float *x, const float *y) {
  for(int i=0; i<N; ++i)
    x[i] = x[i] + y[i];
}
```

**CUDA kernel:**

```
__global__
void vec_add(int N, float *x, const float *y) {
  int i = threadIdx.x;
  x[i] = x[i] + y[i];
}
```

**Call:**

```
vec_add<<<1,N>>>(N, x, y);
```

Additional reading:

https://developer.nvidia.com/blog/even-easier-introduction-cuda/

# Understanding CUDA kernels

**Types of functions:**

`__global__` Device code called from host – *kernel*

`__device__` Device code called from device

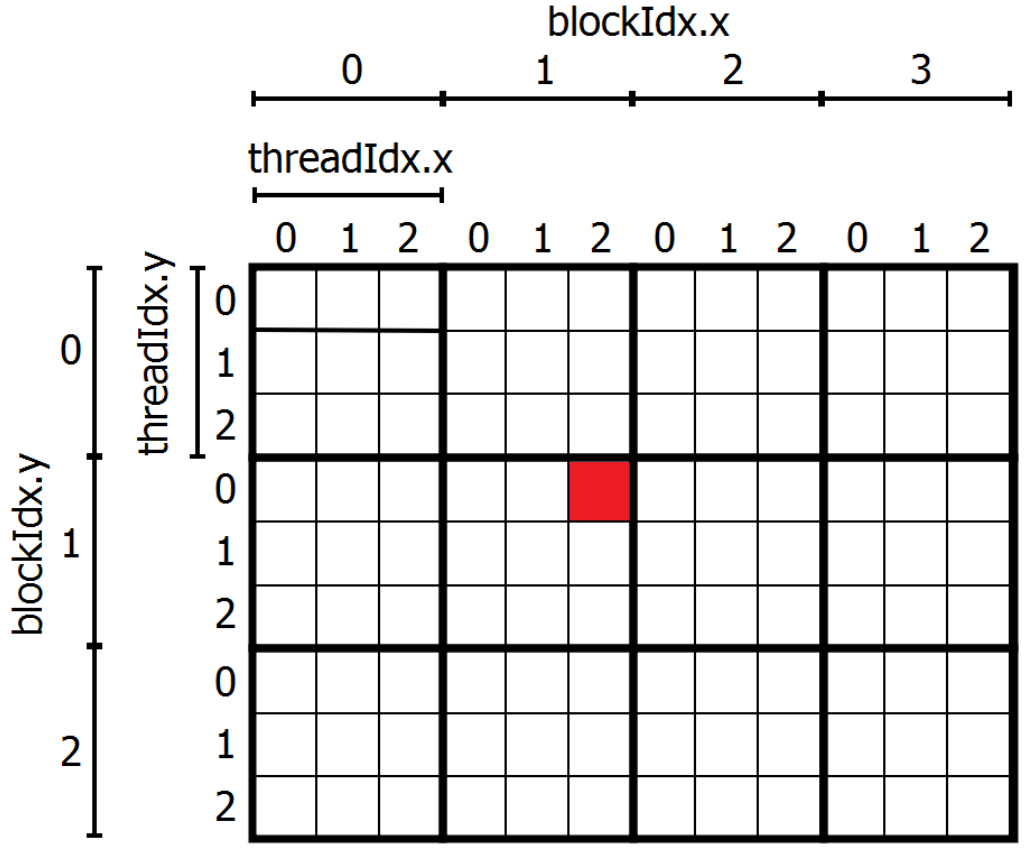
`__host__` Host code called from host (usual functions)

**Identifying threads:**

- Thread in block: `threadIdx`
- Block in grid: `blockIdx`
- Size of block: `blockDim`

`dim3`

- Type: `dim3` – 1D, 2D, or 3D



# Launching CUDA kernels

**Kernel invocation:**

```
kernel<<<grid_dim, block_dim>>>(args);
```

**Configuration parameters:**

- `block_dim` – size of thread block (`dim3`)
- `grid_dim` – blocks per grid (`dim3`)

# Example: matrix addition

## Code:

```
/* kernel */
__global__
void matrix_sum(int N, float *C, const float *A,
                const float *B) {
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  int j = threadIdx.y + blockIdx.y*blockDim.y;
  if(i<N && j<N)
    C[i*N+j] = A[i*N+j] + B[i*N+j];
}
```

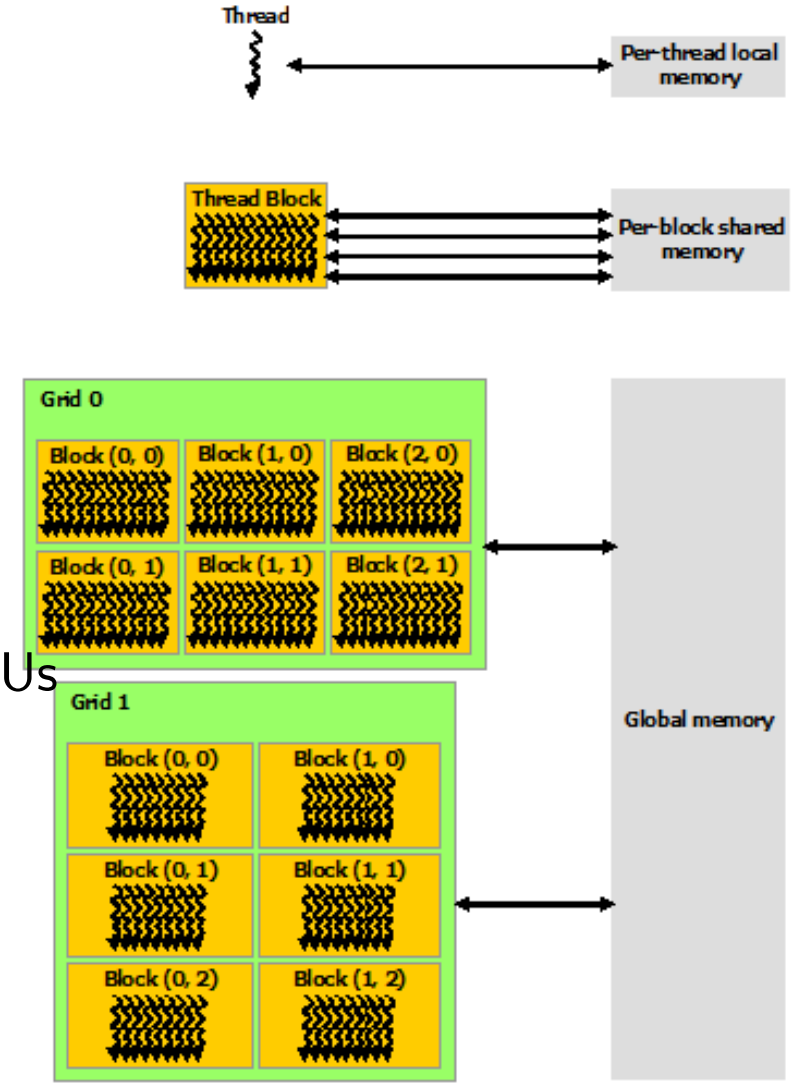
## Invocation:

```
...
  /* kernel configuration */
  dim3 block_dim(8, 8);
  int num_blocks = 1 + (N-1)/8;
  dim3 grid_dim(num_blocks, num_blocks);
  /* kernel launch */
  matrix_sum<<<grid_dim,block_dim>>>(N, C, A, B);
...
```

# Memory in CUDA

**Types of memory:**

- Single thread has *registers*
  - $\leq$ 255 32-bit regs
  - 0 cycle access cost
- Threads in block share memory – *shared* memory
  - up to 192 kB / SM
  - $\sim$ 50 cycles
  - up to $\sim$ 10 TB/s on recent GPUs
- Main device memory – *global* memory
  - $\sim$ 8–40 GB
  - $\sim$ 500 cycles
  - $\sim$ 200–1600 GB/s



# Memory in CUDA

**Allocating device memory:**

- `cudaMalloc` – allocates *global* memory on device
- `cudaFree` – frees it again
- `cudaMemset` – used for initializing memory

**Used exactly like:**

- `malloc`
- `free`
- `memset`

**Device and host pointers:**

- Pointers *either* valid on host or device

**Host-device transfer – `cudaMemcpy`:**

- Host to device:

```
cudaMemcpy(x_dev,x_host,num_bytes,
           cudaMemcpyHostToDevice);
```

- Device to host:

```
cudaMemcpy(x_host,x_device,num_bytes,
           cudaMemcpyDeviceToHost);
```

- Current capabilities see e.g. `https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/`

# Memory in CUDA

**Host-device transfer – `cudaMemcpy`:**

- Host to device:

```
cudaMemcpy(x_dev,x_host,num_bytes,
           cudaMemcpyHostToDevice);
```

- Device to host:

```
cudaMemcpy(x_host,x_device,num_bytes,
           cudaMemcpyDeviceToHost);
```

- Current capabilities see e.g. `https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/`

**Also `cudaMemcpyAsync`:**

- Non-blocking communication
- Overlap communication and computation
- Cf. `MPI_Isend`

**Example:**

```
void main() {
  int n = 256;
  int num_bytes = n*sizeof(int);
  int *x_dev, *x_host;
  /* allocate memory */
  x_host = (int*) malloc(num_bytes);
  cudaMalloc(&x_dev, num_bytes);
  /* set to 0 */
  cudaMemset(x_dev, 0, num_bytes);
  /* copy memory to host */
  cudaMemcpy(x_host, x_dev, num_bytes,
             cudaMemcpyDeviceToHost);
  /* free up memory */
  free(x_host);
  cudaFree(x_dev);
}
```

# Host and device memory

- Separate memory management by `cudaMalloc` and `cudaFree` leads to clear responsibilities
- Software must manage data transfer by `cudaMemcpy`
- Messy for more elaborate data structures (pointers, indirections)
- Coordination between CPU and GPU in `cudaMemcpy`
- A memory address only makes sense within either the CPU (host memory) or the GPU (device memory)
  - Access to wrong memory class invalid!

# Unified memory

- Newer architectures support unified memory, i.e., accessing the same pointer from both host and device
  - See also `https://developer.nvidia.com/blog/parallelforall/unified-memory-in-cuda-6/`
- With CUDA: `cudaMallocManaged` allocates in common address space
- Unified memory pinned, no swapping to disk
- Simpler entry point to GPU programs
  - Developer can concentrate on CUDA kernels
- But the copying still has to happen physically
  - Data locality important: Must do many operations on GPU before GPU makes next access
  - Copying should happen well in advance of use
  - Need `cudaDeviceSynchronize()` call
- Unified memory is tradeoff between simplicity and performance – trade specialization like e.g. `cudaMemcpyAsync` for transparency

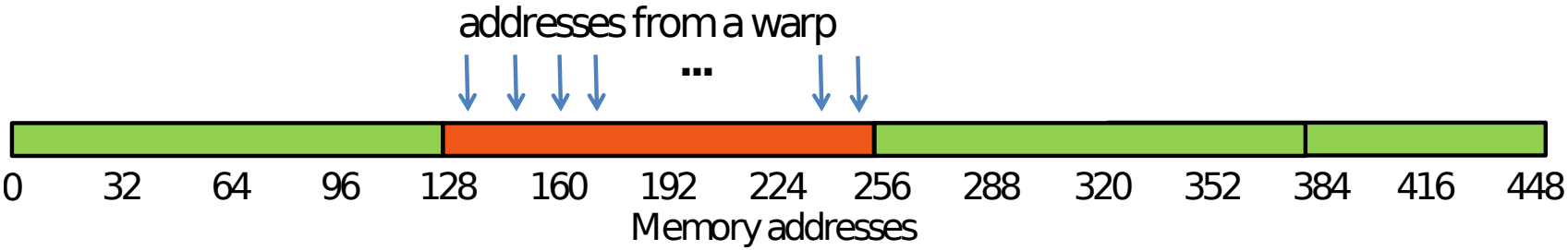
# Memory usage on a GPU

**Global memory access** in GPU memory:

- Memory accessed via 32-, 64-, or 128-byte memory transactions
- Threads in warp access memory together
- Hardware minimizes number of transactions

**Coalesced access 1:**

- Threads read contiguous memory – single transaction
- Memory address aligned by size of transaction



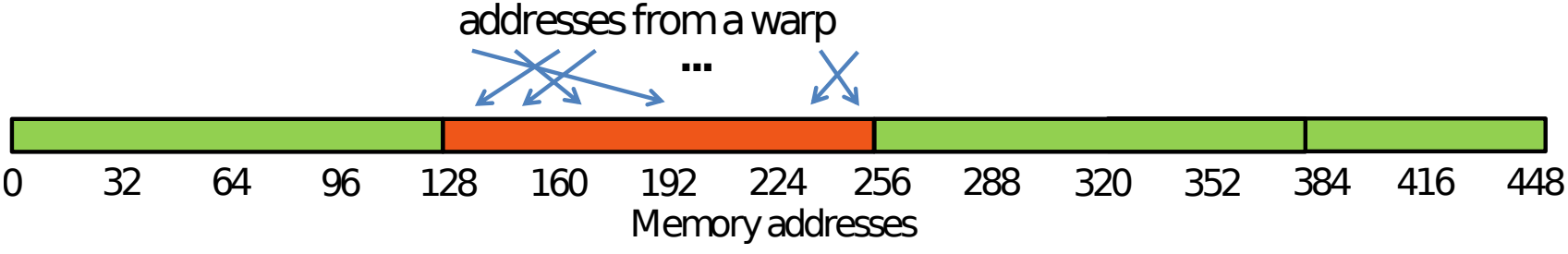
# Memory usage on a GPU

**Global memory access** in GPU memory:

- Memory accessed via 32-, 64-, or 128-byte memory transactions
- Threads in warp access memory together
- Hardware minimizes number of transactions

**Coalesced access 2:**

- Contiguous but permuted access – single transaction
- Access by the form `x[indices[threadIdx]]` with indirect addressing index array `indices`



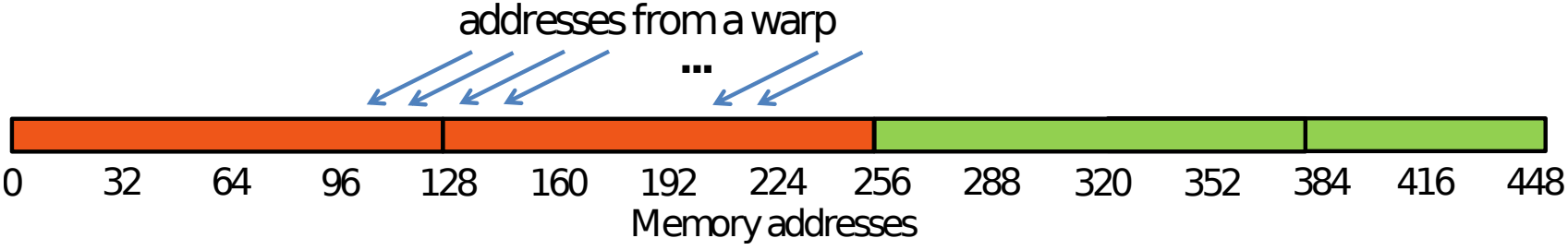
# Memory usage on a GPU

**Global memory access** in GPU memory:

- Memory accessed via 32-, 64-, or 128-byte memory transactions
- Threads in warp access memory together
- Hardware minimizes number of transactions

**Uncoalesced access 1:**

- Contiguous but misaligned – two transactions
- E.g. 128 byte transaction, address not divisible by 128



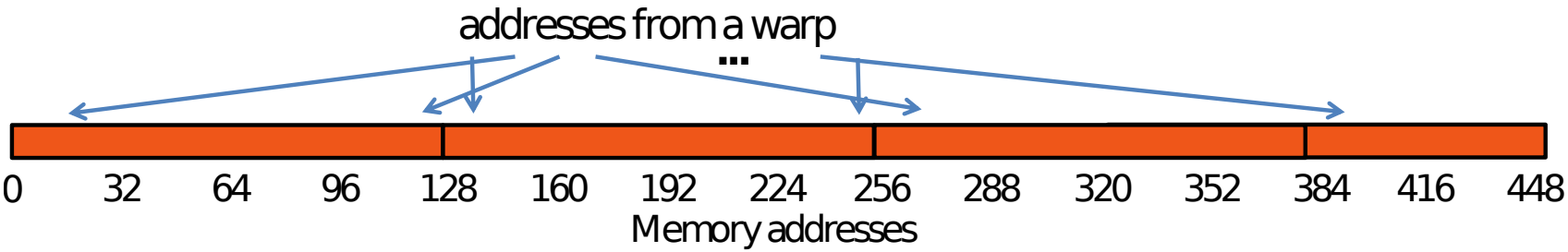
# Memory usage on a GPU

**Global memory access** in GPU memory:

- Memory accessed via 32-, 64-, or 128-byte memory transactions
- Threads in warp access memory together
- Hardware minimizes number of transactions

**Uncoalesced access 2:**

- Non-contiguous access – $N$ transactions




**Note:** Caches help to reduce transactions to memory, but need **temporal locality**

Further reading:

`https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

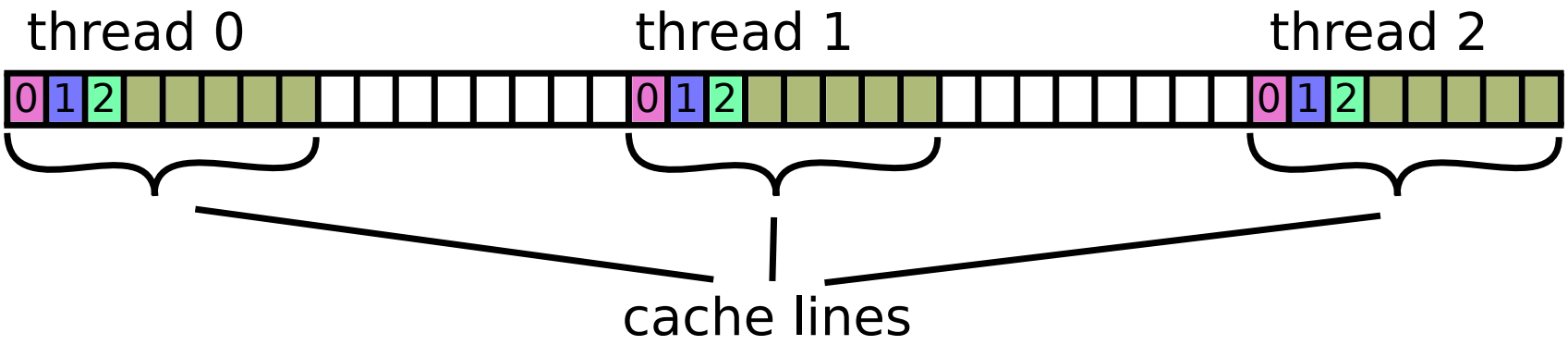
→ Section 5.3.2, Device Memory Accesses

# Comparison between CPU and GPU memory access

**CPU:**

- Local caching for each thread
- Locality for each thread
- Avoid same "cache line" on different threads/cores (ping-pong, false sharing)

**Example:**




**GPU:**

- Threads within an SM prefer nearby access
- *Opposite access pattern* – GPU threads and CPU threads behave differently

# Array-of-Structure vs Structure-of-Array

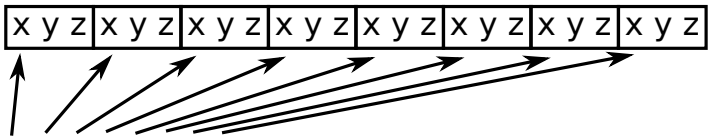
- Access collection of 3D points

```
/* Array of Structure */
typedef struct {
  float x,y,z;
} aos_t;
aos_t aos[1000];

/* access */
float xval = aos[i].x;
float yval = aos[i].y;
float zval = aos[i].z;
```

```
/* Structure of Array */
typedef struct {
  float x[1000];
  float y[1000];
  float z[1000];
} soa_t;
soa_t soa;

/* access */
float xval = soa.x[i];
float yval = soa.y[i];
float zval = soa.z[i];
```

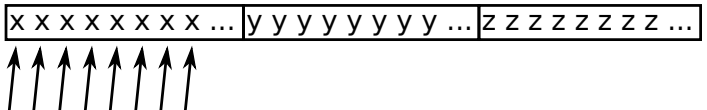
*Array-of-Structure good on CPU*

- Values of point on same cache line
- All data from same stream
- Easy to prefetch when walking over it




*Structure-of-Array good for GPU*

- Contiguous access for multiple threads
- Parallelism across points
- (Preferable for SIMD on CPU, too)



# Shared memory on the GPU

- Small and very fast memory shared by thread block
- E.g. user-managed cache, or scratchpad for cooperative algorithm
- Programmer responsible for avoiding race conditions

**Usage:**

```
__shared__ int buffer[SIZE];
```

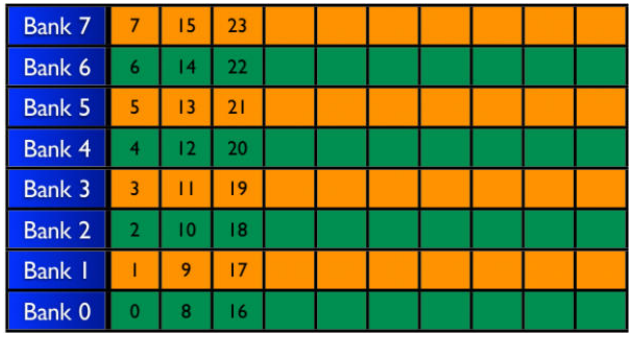
- Automatic L1 cache
- Since 2010 (Fermi architecture)
- Configurable, 16kB+48kB / 48kB+16kB

# Access of shared memory: banks

**Shared memory banks:**

- 32 banks
- Consecutive 4B-*words* belong to different banks, cyclically

**8-bank example:**



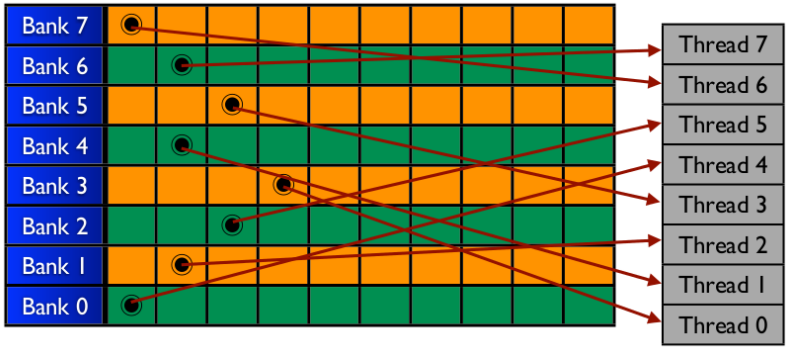

**Bank conflicts:**

- Penalty if threads in warp access same bank
- Access into same bank serialized

# Access of shared memory: possible bank conflicts

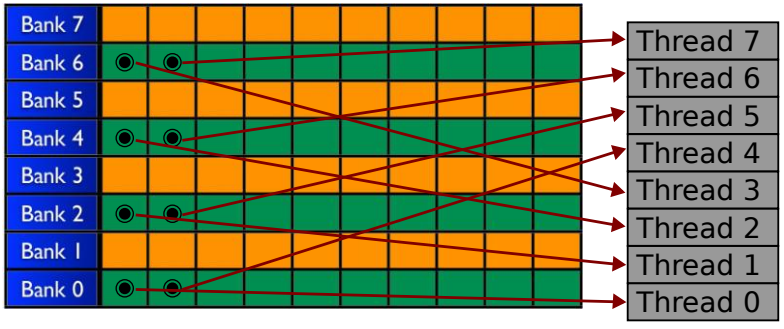
## All different banks:

- No conflicts



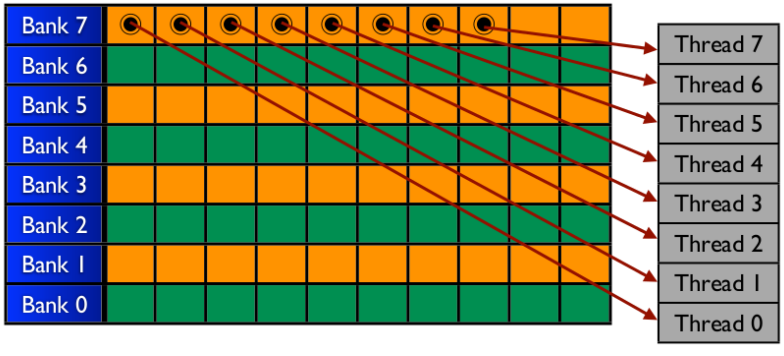

## 2-way bank conflict:

- Half performance



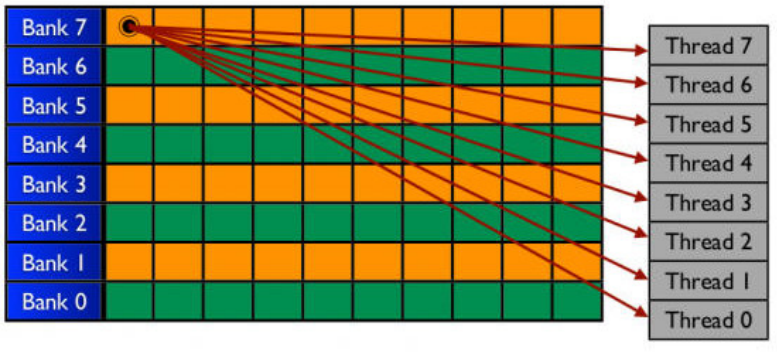

## Full bank conflict:

- 1/8 performance




## Broadcast:

- Threads access *same location* – no overhead



# Memory access problems by type of memory

- Bank conflicts: transfer between GPU cores and shared memory
- Uncoalesced access: transfer between GPU cores and global device memory
- Slow PCI-e bus: transfer between GPU and host

# Synchronization in CUDA

- **Global synchronization** across full GPU does not exist
- Implicit global synchronization at kernel boundaries
  - A kernel does not return anything
  - Synchronization: wait for kernel on CPU, send out next kernel, etc.
  - Feed data for iteration $i+1$ while kernel for iteration $i$ is running

**Synchronization within block:**

- `__syncthreads()`
- Barrier for threads in block
  - When one thread calls `__syncthreads()`, all threads in block must do so
- Avoid data race when using `shared` memory

**Example: array reversal**

```
__global__ void reverse(int *d,
                        int n)
{
  __shared__ int s[64];
  int t = threadIdx.x;
  int tr = n-t-1;
  s[t] = d[t];
  __syncthreads();
  d[t] = s[tr];
}
```

# Atomic operations in CUDA

- Perform a combined read/write operation in thread-safe manner
- `atomicAdd`, `atomicSub`, `atomicMax`, etc
- Typically resolved by cache system
  - Instruction returns immediately
  - Conflict resolution on store
  - Fire-and-forget semantic
- Besides device-wide atomics, there are also block-wide atomics (suffix `_block`, e.g. `atomicAdd_block`) or system-wide atomics across all GPUs and CPUs (suffix `_system`)

# Parallel reductions

**Vector summations, min/max** important algorithms (dot product, norms, means, ...)

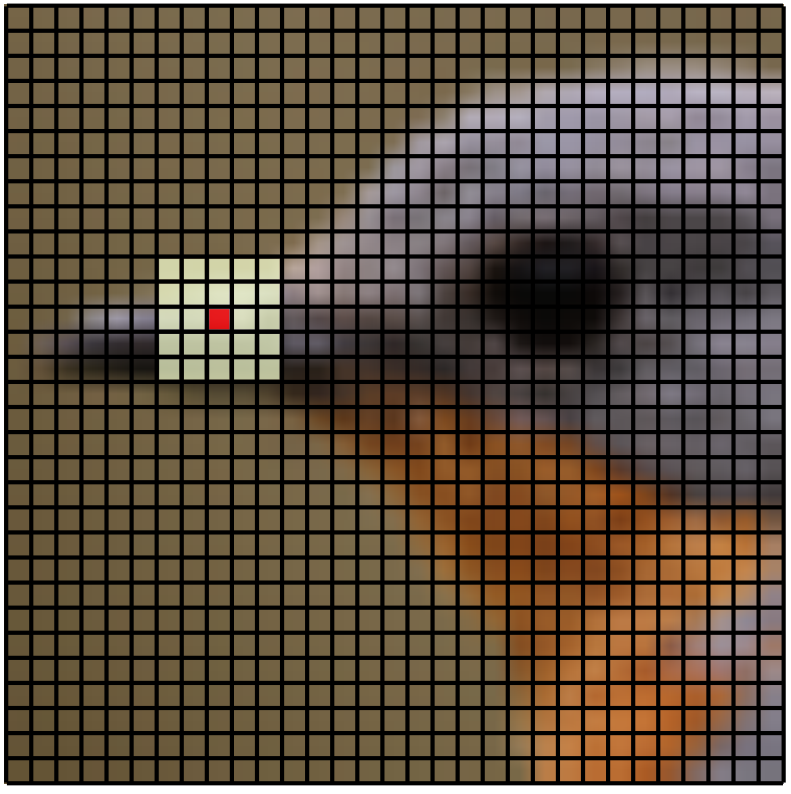
```
sum=0;
for(int i=0; i<N; ++i)
  sum += x[i];
```

- Uses concepts of shared memory, synchronization and atomic operations due to concurrent access
- Presentation: "Optimizing Parallel Reduction in CUDA", Mark Harris `http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`

# Example algorithm: Gaussian blur

**Blur an image:**

- Average neighboring 5x5 pixel values



**Algorithm:**

```
for(int i=2; i<Height-2; ++i)
  for(int j=2; j<Width-2; ++j) {
    float tmp=0;
    for(int ii=-2; ii<=2; ++ii)
      for(int jj=-2; jj<=2; ++jj)
        tmp +=
         Weight[(ii+2)*5+(jj+2)]*
         Ain[(i+ii)*Width+j+jj];
    Aout[i*Width + j] = tmp;
  }
```

**Idea:**

- All pixels are independent
- Both `i` and `j` loops are perfectly parallel

# Example algorithm: Gaussian blur

- On the GPU, want as many threads as possible
- One thread per pixel
- 2D blocks and grid

**Kernel code:**

```
__global__
void blur_kernel(int width, int height, float *W,
                 float *Ain, float *Aout)
{
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  int j = threadIdx.y + blockIdx.y*blockDim.y;

  if( (i>1 && i<height-2) && (j>1 && j<width-2) )
  {
    float tmp = 0;

    for(int ii=-2; ii<=2; ++ii)
      for(int jj=-2; jj<=2; ++jj)
        tmp += W[(ii+2)*5+(jj+2)]*Ain[(i+ii)*width+j+jj];

    Aout[i*width + j] = tmp;
  }
}
```

# Example algorithm: Gaussian blur

**Host code** to run the Gaussian blur filter:

```
// W, Ain, Aout are device arrays

// 2D grid and blocks
int nblocks_w = 1 + (width-1)/block_size;  // int division
int nblocks_h = 1 + (height-1)/block_size; // rounding up
dim3 gd_dim(nblocks_h, nblocks_w);
dim3 bk_dim(block_size, block_size);

blur_kernel<<<gd_dim,bk_dim>>>(width,height,W,Ain,Aout);
```

# What can be done and cannot be done in CUDA

## Possible with CUDA

- Memory allocation
- Recursive function calls
- Polymorphism/virtual methods (object on GPU)
- Call other CUDA kernels
- Some header-based C++ libraries

## Impossible with CUDA

- C++ exception handling
  - Even if it compiles, will not throw properly
  - Cannot catch exceptions
- Exceptions also bad for performance in CPU programming
  - Might want to use them as little as possible anyway
- Parts of C++ library, standard allocations, very new C++ features

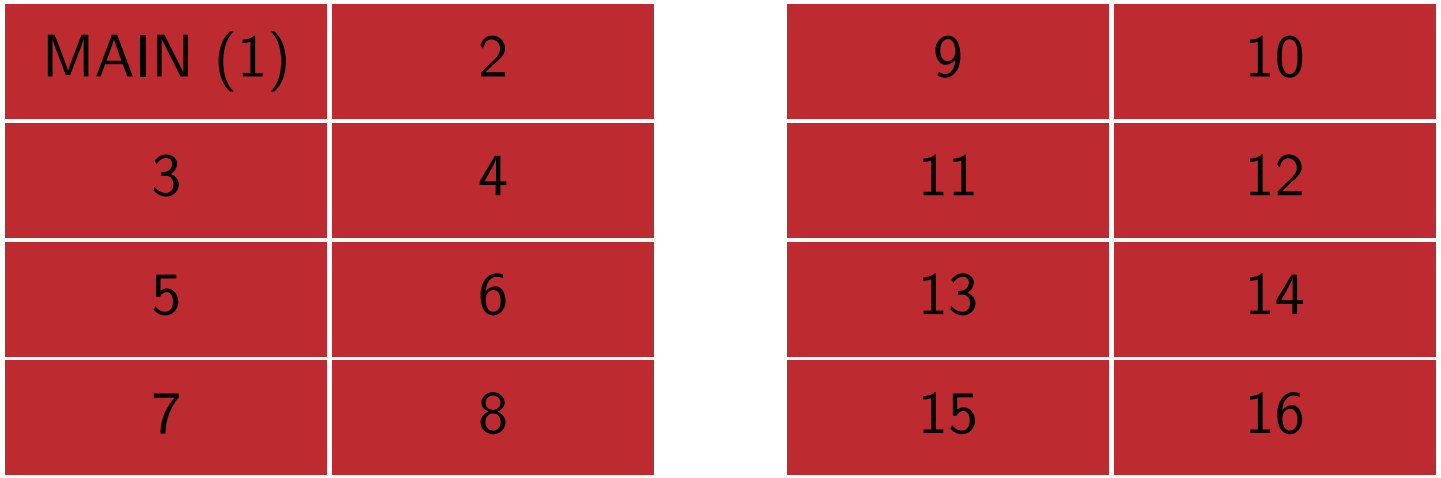
# OpenMP Target

# OpenMP Target

- OpenMP is a way to express thread-parallelism in a compiled environment
  - Most common concept for programming multi-core CPU (shared-memory parallelism)
- Standardized
- Main concept: add OpenMP pragmas to code
  - Same code compiled without OpenMP support will run ordinary serial program
- One way to obtain performance for compiled languages

# OpenMP structure

- Classic OpenMP centers on using multiple threads within the same device
  - The original main thread is running on an identical CPU, with an identical memory address space, as the other threads
  - All those threads can communicate freely
    - Although, in practice, some have more shared resources than others

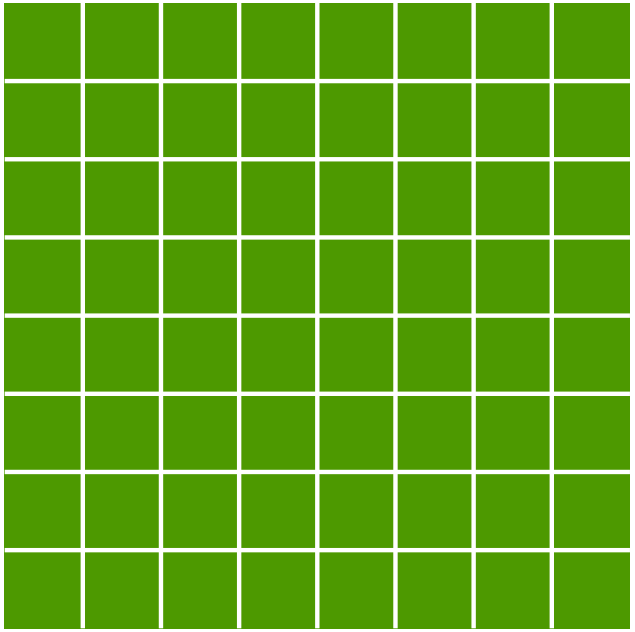


# OpenMP vs OpenMP Target

- OpenMP Target allows offloading computations to a different unit with and independent memory space

| MAIN (1) | 2 |
|---|---|
| 3 | 4 |
| 5 | 6 |
| 7 | 8 |

| 9 | 10 |
|---|---|
| 11 | 12 |
| 13 | 14 |
| 15 | 16 |



# Different levels of execution

- OpenMP generally focuses on one type of parallelism
  - But a GPU has two main levels of parallelism
    - Grid (e.g. 100 blocks per grid)
    - Block (e.g. 256 threads per block)
    - Additional level: wave of blocks (running on same SM)
    - Hardware execution at level of warps (32 entries)
- In OpenMP Target, the grid level is represented as "teams"
  - The team set is the grid
  - Each block contains one OpenMP master thread
  - That thread can then denote parallel sections

# Access to data

- In standard OpenMP, can read from data arrays and scoped variables directly
  - Variables can be explicitly set as `private`, `shared` etc
    - Private implies that data is copied to each child thread
- In OpenMP Target, *any* data needs to be copied
- From the prelab

```
#pragma omp target teams \
  map(to:origdata[:elems*side*side]) \
  map(from:divergence[:elems*elems])
```

- Copy specified range of array **to** the GPU before the block starts
- Copy specified range of another array **from** the GPU at the end

# Loops in OpenMP Target

- `omp distribute`
  - Run loop over all teams
- `omp parallel for`
  - Run loop over all threads in team
- `omp distribute parallel for`
  - Run with maximum parallelism
- `collapse(n)`
  - Use same interpretation for several nested loops
- `reduction`
  - Specify variables that are to be summed/maximized/combined in some other way between threads
- `schedule`
  - Specify a scheduling regime

# Summary of OpenMP Target

## Pros

- OpenMP Target allows to work with the regular language (e.g. C++)
- The same code can also be a performant CPU parallelization
- No need to go into warp scheduling details as with CUDA

## Cons

- Dependent on compiler for performance, limited ability to change
  - If compiler does not see a nice parallelization layout, one is essentially lost
  - Different compilers might require different layouts
  - Different hardware might require different layouts for good performance – how to create portable code for various hardware?
- Limited access to more advanced synchronization/reduction features
- Mostly support for low-level features

# Thrust

- Template-based library
  - Mostly directed towards users familiar with C++ templates
  - Uses iterators and functors to express logic
- In C++ standard library, there is

```
std::accumulate(vec.begin(), vec.end(), 0,
                std::plus<int>());
```

- In Thrust, there is

```
thrust::reduce(vec.begin(), vec.end(), 0,
               thrust::plus<int>());
```

- Documentation:
  `https://docs.nvidia.com/cuda/thrust/index.html`

# Features of Thrust

- Can execute code on host and on device
  - Host framework does not provide optimal performance
  - Still reasonable with thread parallelism
- Efficient tools for sorting, sparse lookup or similar algorithms are hard to write massively parallel
- Fixed function approach, but allow for "kernel fusion"
  - We compute the differences between two digits and square result
  - In `numpy`, this might be written as
    - `diff = sum((A - B)**2)`
    - In many cases, this will create a temporary matrix `A - B`, another temporary object with element-wise squares, before final summation is done
    - Looping over the same data three times

# Iterators in Thrust

- Iterators are object that return elements
  - Also returned by C++ standard library classes, e.g. `vec.begin()`
- Work similar to pointers, but much broader use cases
  - An `operator++` might do much more than just increment address
- Thrust brings additional functionality such as
  - `transform_iterator`: Accept elements from one iterator and call a functor on each data entry (like creating a "virtual" array where element is squared)
  - `zip_iterator`: Combine elements from two iterators into tuples of elements
  - `constant_iterator`: Return the same element over and over again
  - `counting_iterator`: Return a fixed sequence of numbers, like range in Python
- The algorithm does not really care what kind of iterator you give it

# Functors in Thrust

- An object that can be called as function
  - Classic way is to overload the codeoperator() in C++
- You can also do it using Lambda expressions

```
thrust::reduce(vec.begin(), vec.end(), 0,
               [] (int a, int b) { return a * b; });
```

- Returns the product of all elements
- With transform iterators and hook-in points for functors, Thrust is very flexible
- Used in pre-lab 3 code `thrustmax.cu` to compute the maximum of the divergences, without storing them in an array first.

# Summary of Thrust

## Pros

- Provides efficient algorithms for many hard tasks
  - Complex reductions, sorts, scans, ...
- Plug in your own algorithms and data structures
- Can co-exist with "raw" CUDA code
- Somewhat efficient CPU implementation allows shared source code

## Cons

- Awkward if you are not into the iterator style of the C++ STL
- Performance might not improve a lot over OpenMP Target if you are mostly doing loops anyway $\rightarrow$ helpful for more complicated layouts

# Other domain-specific abstractions

# CUB

- CUB is a library for doing warp-wide, block-wide, and device-wide operations
- Parts of it are used in the Cuda backend for Thrust
- Reduces risk for errors for slightly more complex synchronizing operations
  - Simplify operations like “use all threads in the block to collaborate on loading data into shared memory”

# Kokkos: An HPC-oriented C++ abstraction library

- Similarly to Thrust, targets a single-source implementation with C++ programming language
- Create *execution policy* to important computational kernels (for loop, reduction, ...) and express work as *computational body* (code to perform unit of work)
- Descriptive programming model
- Compile the code for different targets
  - CPU target
  - GPU target
- Select *different data layouts* adapted to hardware architecture
  - Map algorithm parallelism to hardware parallelism
  - Granularity of data access from different threads
- Goals of Kokkos:
  - Ease of use
  - Flexibility also to large codes
  - Performance

$\rightarrow$ *performance portability*

Model for **data parallelism**

- Use **parallel patterns** and **execution policies** to execute **compute bodies**
- Example 1: parallel loops with the `parallel_for` pattern (e.g. vector add)

```
parallel_for ( "stream_triad" , N , [=] ( int i ) {
                 z(i) = a * x(i) + y(i);
               });
```

- Example 2: reductions combine results from loop iterations

```
float result;
parallel_reduce ( "summation" , N ,
                  [=] ( int i, float &lres ) {
                    lres += x(i);
                  }, result);
```

- Flexibly choose data-layout problem with **multi-dimensional array** abstraction
- Execution and memory spaces to control
  - where data lives
  - where code executes

# Kokkos resources

- Extensive tutorial material provided by Kokkos team
- Extensive lecture material: `https://github.com/kokkos/kokkos-tutorials/tree/main/LectureSeries`
  - Recommended reads
    - Lecture 1: Introduction
    - Lecture 2: Memory views and spaces
    - Lecture 3: Multi-dimensional loops and data structures
- Condensed introduction: `https://github.com/kokkos/kokkos-tutorials/blob/main/Intro-Short/KokkosTutorial_Short.pdf`
- Kokkos source code:
  - `https://github.com/kokkos/kokkos`
- Note that Kokkos team aims to integrate functionality into C++ standard library

# Other frameworks

- OpenCL
  - Standardized "alternative" to CUDA, but much worse language support
  - Started out from the HLSL/GLSL concept of "sending a string of source code to the driver"
- SyCL
  - Attempt to bring sort of standard C++ to GPU, but not super clean. Mostly vector-specific implementations. Intel pushes SyCL through their oneAPI framework
  - CUDA has a relatively mature parallel implementation in the Clang compiler
- OpenACC
  - Similar to OpenMP Target, but more heavily geared towards GPUs. Some proprietary implementations might be faster than OpenMP Target, both Clang and GCC base their OpenACC support on mapping it to OpenMP equivalents.
- Standard C++
  - Nvidia bought compiler vendor PGI.
  - Promised feature in their HPC SDK 20.05 to have a compiler where standard STL features are executable on GPU.
    - Mostly removing the need for a pseudo-specific library like Thrust.
    - Improved support for corner cases for unified memory will help.

# My experience (biased towards HPC)

- As a code grows larger, want to run it on more than one hardware architecture
  - Classical CPU-based systems (Intel, AMD, arm)
  - NVIDIA GPUs
  - AMD GPUs
  - Intel GPUs
  - Some other architecture acquired by your computing center
- How to address difference in architectures?
  - Separately implementing everything on multiple architectures not realistic
    - Write and maintain code with multiple paths – `#ifdef`?
    - Needs highly skilled developers for several architectures
    - Directive-based approach very brittle
  - Separately implementing important algorithms on multiple architectures works sometimes
    - Needs big enough team to drive it
    - Separation of concerns important: Multiple "back-ends" (CPU, GPU) for key algorithms, but abstract it away from application code
    - How to flexibly develop application?

# Our recommendation

- Current state of the art:
  - For open standards, go for OpenMP Target
  - If you need to maximize performance, give CUDA a look
    - Possibly using CUB or Thrust when those map well, avoid repeating your own low-level code when you can (maintenance burden!)
- Languages standardizing the features needed to coordinate light threads properly upcoming
  - Some are in C++20, some will come in C++23
  - These will benefit other heterogeneous architectures to some extent as well
    - C++ on FPGA?! High-level synthesis