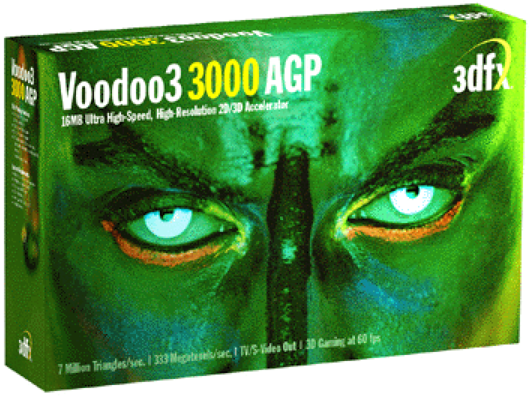
# GPUs and CUDA 1 Threading

CS121 Parallel Computing

Spring 2020

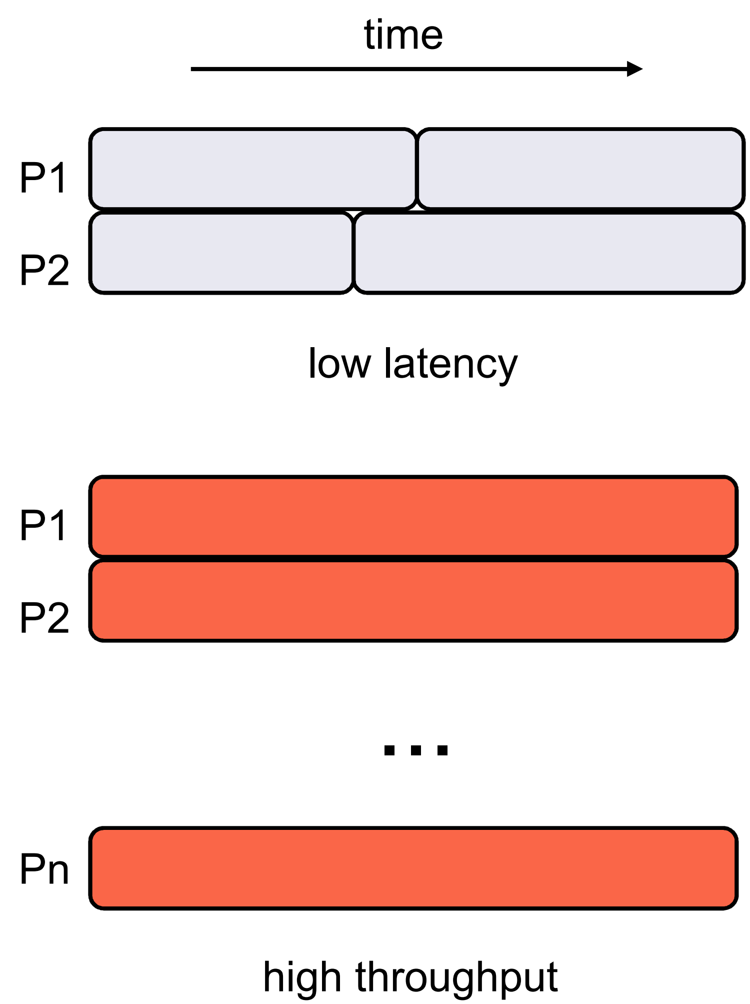
# A brief history

- Graphics processing units (GPU) originally used to speed up 3D games.
- Need high throughput (lots of pixels), but parallelism abundant (compute pixels independently).
- Fancier games required programmable "pixel shaders".
- Around 2006, Nvidia introduced Tesla, a programmable, general purpose GPU (GPGPU).
- GPUs now essential in machine learning, big data and HPC. Large amounts of research.
- GPUs have TFLOPS of performance, "supercomputer on a chip".
- Also more energy efficient than CPUs, which is increasingly important.

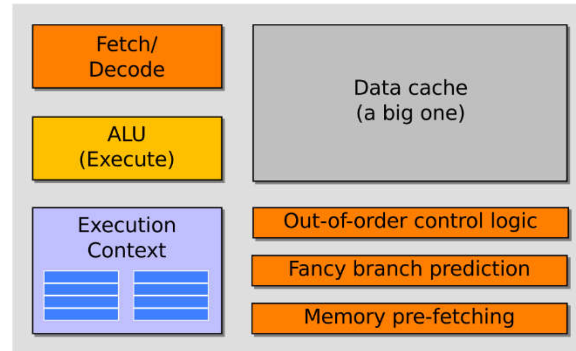
# Latency vs throughput

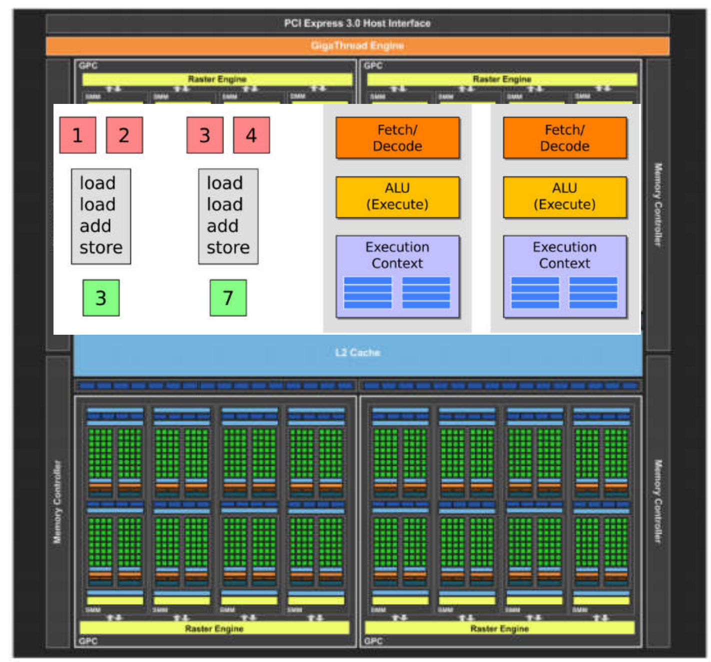
- Up to now we looked at message passing and shared memory parallel computing using standard multicore processors.
- Multicore processors have a few cores, and try to minimize latency on each core.
- Throughput oriented parallel processors do each task slower, but have many cores, and so can do many tasks in parallel.
- Throughput processors can do more work per unit time.



# GPU vs CPU architecture

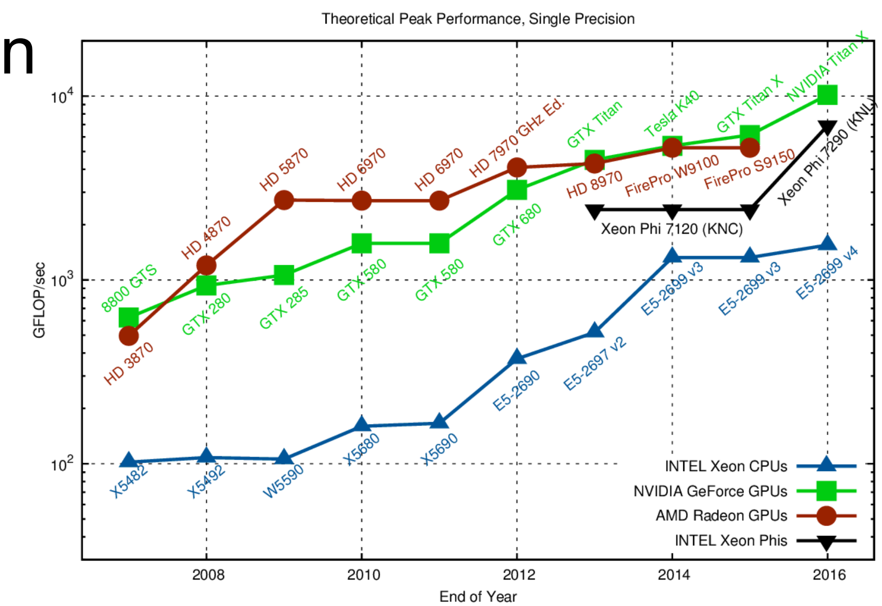
- CPU has many complex features to lower latency.
  - Consumes lots of die space.
  - Less space for compute units.
- GPU only has basic processor units, and shares them among the cores.
  - Each core slower.
  - But lots of them.
- Nvidia Tesla P100 has 56 SMs and 64 cores per SM.
  - Runs 3584 threads simultaneously, 11 TFLOPS of performance.
  - 16 GB of memory, 720 GB/s of bandwidth.
- Intel Xeon E7-8890 v4 runs 48 threads simultaneously (using hyperthreading), about 3 TFLOPS.



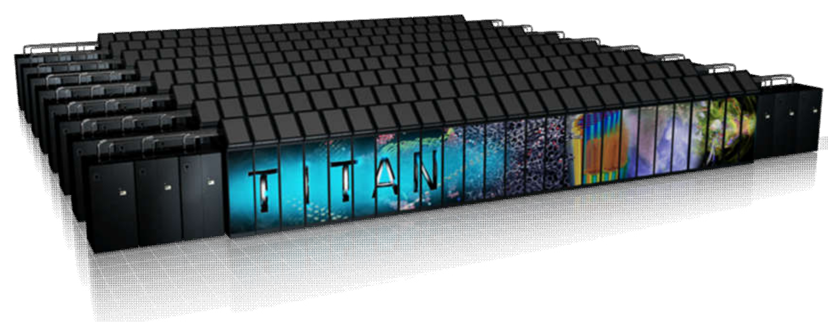



# The right choice(?)

- GPUs >10 times faster than CPUs for many problems.
  - Even more speedup for specialized applications.
- GPUs also much more energy efficient.
- Titan (20 petaflops) uses 18,688 Nvidia Tesla K20X GPUs.
- Best for data parallel tasks.
- GPU is based on SIMD architecture.
- Less effective for irregular computations (branching, synchronization).

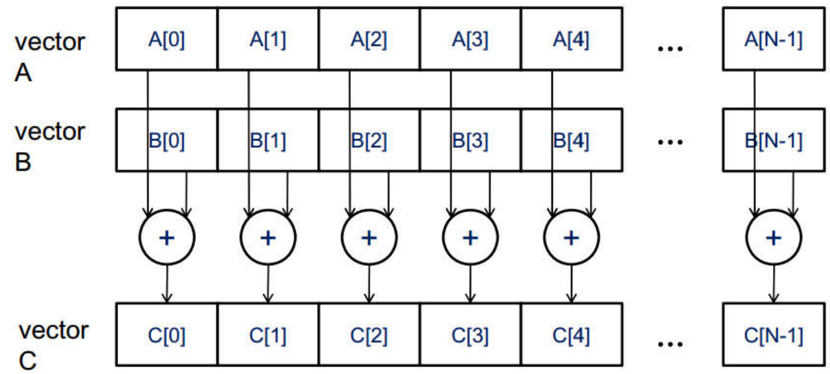



*Source*: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/



# Data parallelism

- Apply same operation to multiple data items.
- Vector addition.



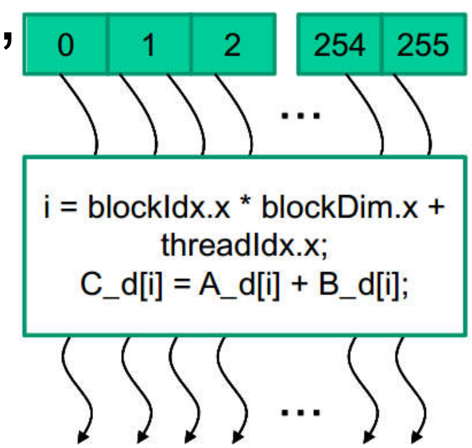

- Linear algebra (matrix-vector, matrix-matrix multiplication).
- Computer graphics.
- Data analysis (convolutions, FFT).
- Finite elements.
- Simulations.
- “Big data”, data mining and machine learning.

# GPU example: vector addition

- Sequential program iterates through the elements.

```
void vecAdd(float* A, float* B, float* C, int n)
{
  for (i = 0, i < n, i++)
    C[i] = A[i] + B[i];
}
```

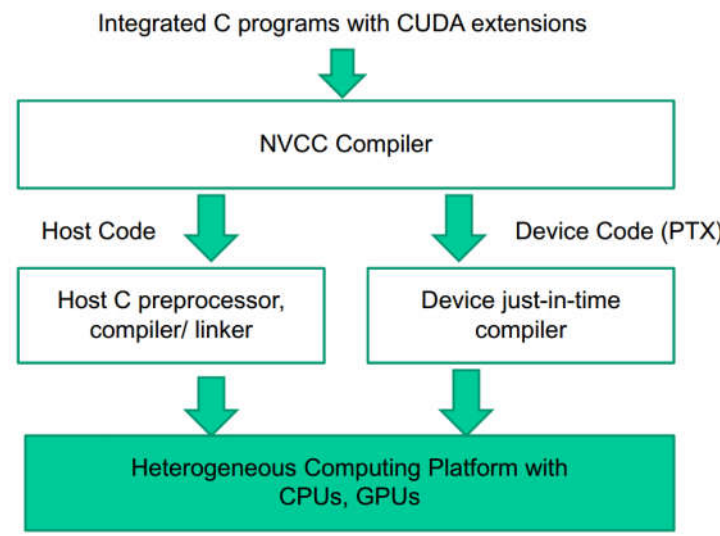
- GPU kernel launches many threads, one for each vector element.
  - Potentially millions of threads.
  - Hardware ensures low (almost zero) overhead thread management.




```
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```

# CUDA

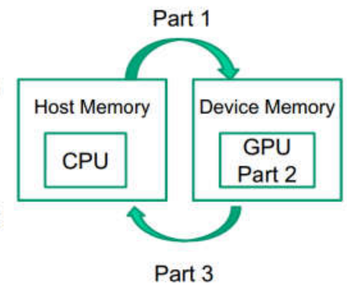
- Compute Unified Device Architecture.
- Easily use GPU as coprocessor for CPU.
- Popular Nvidia platform for programming GPUs.
  - An extension of C language.
  - Compiler, debugger, profilers provided.
- Other platforms include OpenCL and OpenACC.
  - OpenCL is similar CUDA, but more portable.
    - Same source code can be compiled for GPUs, CPUs, FPGAs, etc.
    - Somewhat lower performance than CUDA.
  - OpenACC similar to OpenMP, i.e. write GPU code using simple directives.
    - Compiler takes care of parallelization.
    - Significantly lower performance than CUDA.

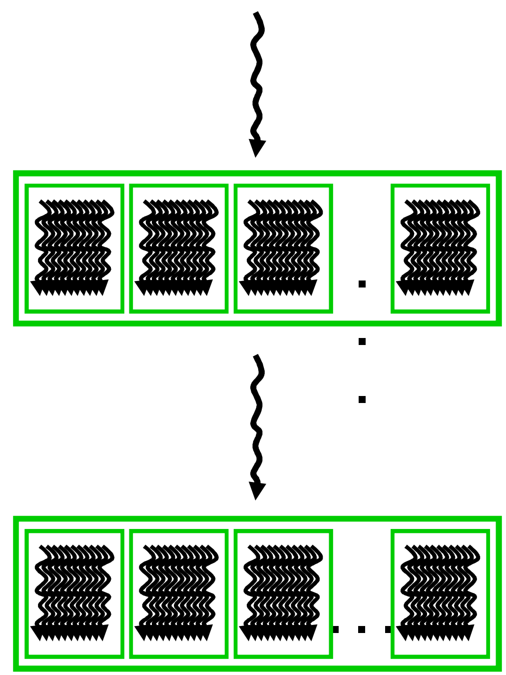


# CUDA steps

- Write C program with CUDA annotations and compile.
- Start CUDA program on host (CPU).
- Run mostly serial parts on host.
- For parallel part
  - Allocate memory on device (GPU).
  - Transfer data to device.
  - Specify number of device threads.
  - Invoke device kernel.
- Can pass control back to CPU and repeat.

```
#include <cuda.h>
…
void vecAdd(float* A, float*B, float* C, int n)
{
  int size = n* sizeof(float);
  float *A_d, *B_d, *C_d;
  …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```



# CUDA functions

- Use labels to declare host and device functions.

| | Executed on the: | Only callable from the: |
|---|---|---|
| `__device__ float DeviceFunc()` | device | device |
| `__global__ void KernelFunc()` | device | host |
| `__host__ float HostFunc()` | host | host |

- Allocate memory on device.

```
cudaMalloc((void **) &x, size)
```

- Transfer memory.
  - Let x be some host data and `d_x` be a pointer to device memory.
  - From host to device (send input).

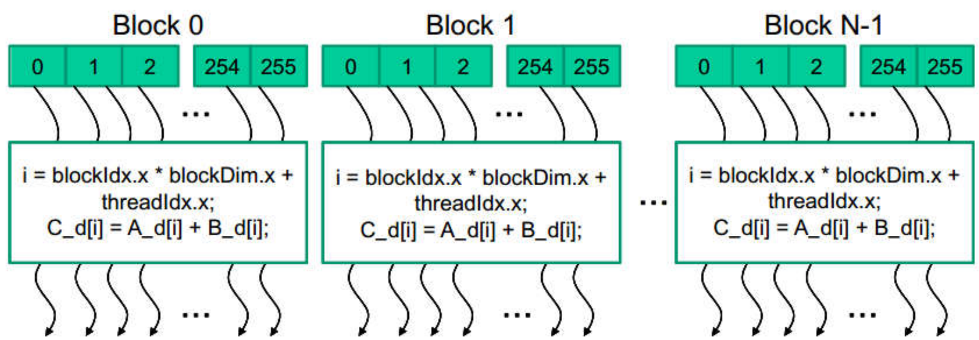
```
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice)
```

  - From device to host (receive output).

```
cudaMemcpy(x, d_x, size, cudaMemcpyDeviceToHost)
```

# CUDA functions

- When calling kernel, must specify number of threads.
  - Threads grouped into blocks.
  - Specify number of blocks, and number of threads per block.



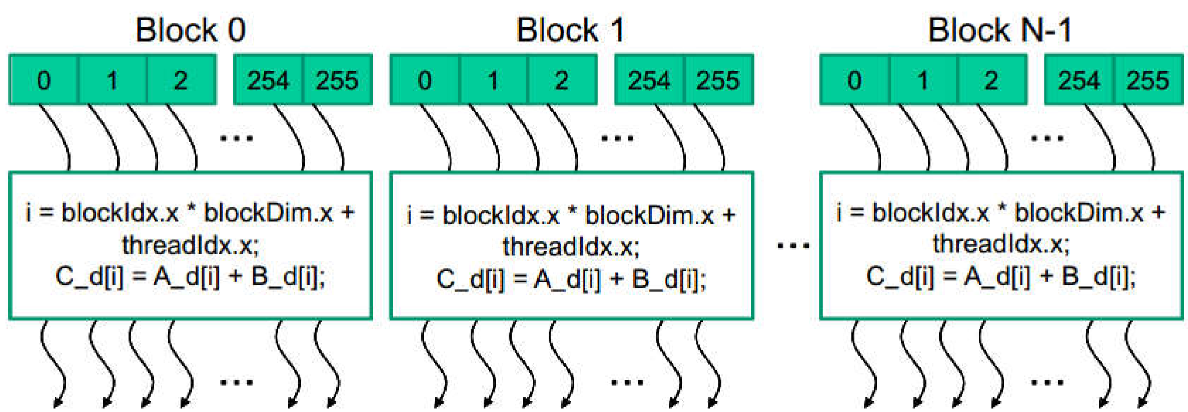

- Invoke kernel.
  - Let `n` be total # threads, `t` be # threads per block.
    - Start `ceil(n/t)`thread blocks with `t` threads each.
  - `KernelFunction<<<ceil(n/t), t>>>(args)`
  - `ceil` ensures we have at least n threads.

# Vector addition code

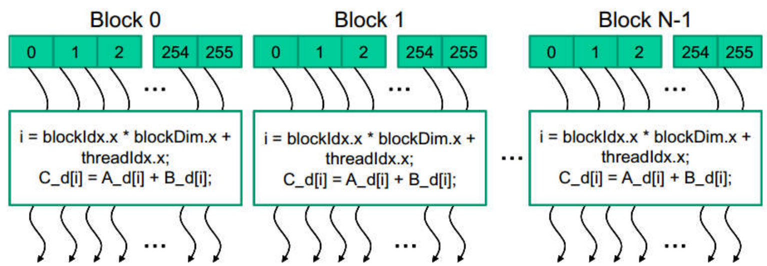
```
__global__
void vecAddKernel(float* A, float* B, float* C, int n) {
    int i = threadId.x + blockDim.x * blockId.x;
    if (i<n) C[i] = A[i] + B[i];
}

void vecAdd(float* A, float* B, float* C, int n) {
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);

    vecAddKernel<<<ceil(n/256), 256>>>(d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
}

int main() {
    vecAdd(A_h, B_h, C_h, N);
}
```

# CUDA thread organization

- All CUDA threads run the same code.
  - But they can operate on different data based on their thread ID.
  - They can also be at different points in the code.
- Threads are organized in two levels.
  - A "grid" containing multiple thread blocks.
  - Each thread block contains a number of threads.
    - All blocks have same size (i.e. number of threads).
  - Grid and blocks can be 1D, 2D or 3D. Let's look at 1D first.
  - Will discuss reason for having two levels later.



# 1D thread mapping




- When kernel is started, all threads assigned a unique (block number, thread number within its block).

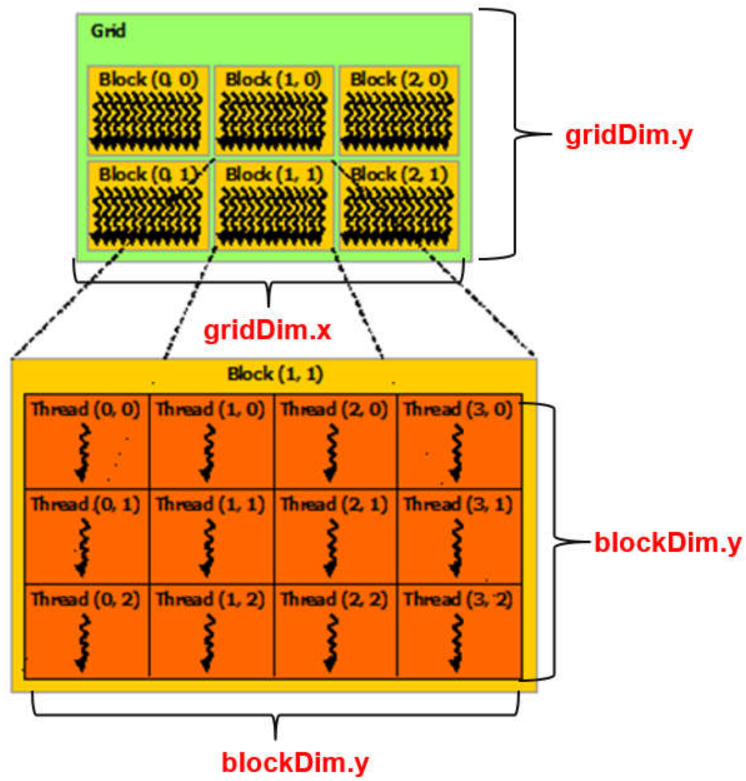
```
KernelFunction<<<ceil(n/t), t>>>(args)
```

  - So we can uniquely identify a thread by its (blockId.x, threadId.x).
  - Number of threads in a block = blockDim.x.
- Ex For vector addition, want every element to be processed by a thread.
  - Let's map thread (blockId.x, threadId.x) to vector element

    blockId.x * blockDim.x + threadId.x
  - Ex Block size 256. Thread 23 in block 3 maps to element 3*256+23 = 791.
  - Each thread mapped to a different element.
  - Every element from 0 to n-1 assigned a thread.
  - Other mappings also possible, depending on problem requirements.

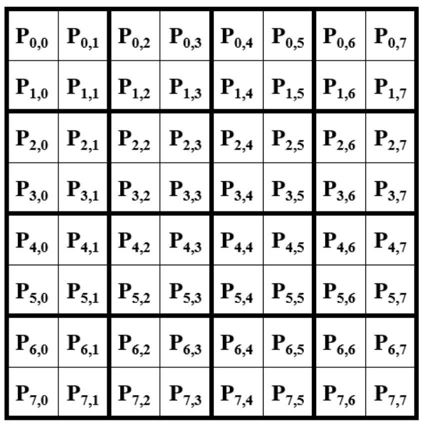
# Multidimensional thread organization

- Since vectors are 1D, natural to use 1D thread organization.
- For 2D (matrices, computer graphics, etc) and 3D (volumetric, 2D + time) data, more natural to use 2D or 3D thread organization.
- The grid of thread blocks can be 1D, 2D or 3D.
- Each thread block within a grid can also be 1D, 2D or 3D.
- The grid and thread block dimensions don't have to be equal.
- Grid and block size should be power of 2.
- Each thread is identified by
  - A block ID (blockId.x, blockId.y, blockId.z).
  - Within its block, its thread ID (threadId.x, threadId.y, threadId.z).

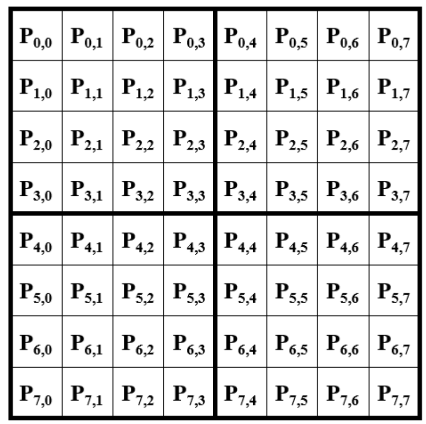


# Starting a 2D thread block

- Map threads to a matrix P of size WIDTH x WIDTH.
- One way is to tile matrix with square thread blocks.
  - Make blocks of size (TILE_WIDTH x TILE_WIDTH).
  - Make WIDTH / TILE_WIDTH blocks in each dimension.




**WIDTH = 8, TILE_WIDTH = 2**
**16 blocks, each with 4 threads**




**WIDTH = 8, TILE_WIDTH = 4**
**4 blocks, each with 16 threads**

- Start kernel using

```
dim3 dimGrid(WIDTH / TILE_WIDTH, WIDTH / TILE_WIDTH, 1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
MatrixMulKernel<<<dimGrid, dimBlock>>>(args);
```

# 2D thread mapping

- Map each thread to an element of P, i.e. a row and a column of P.

  row = blockId.y * blockDim.y + threadId.y

  column = blockId.x * blockDim.x + threadId.x

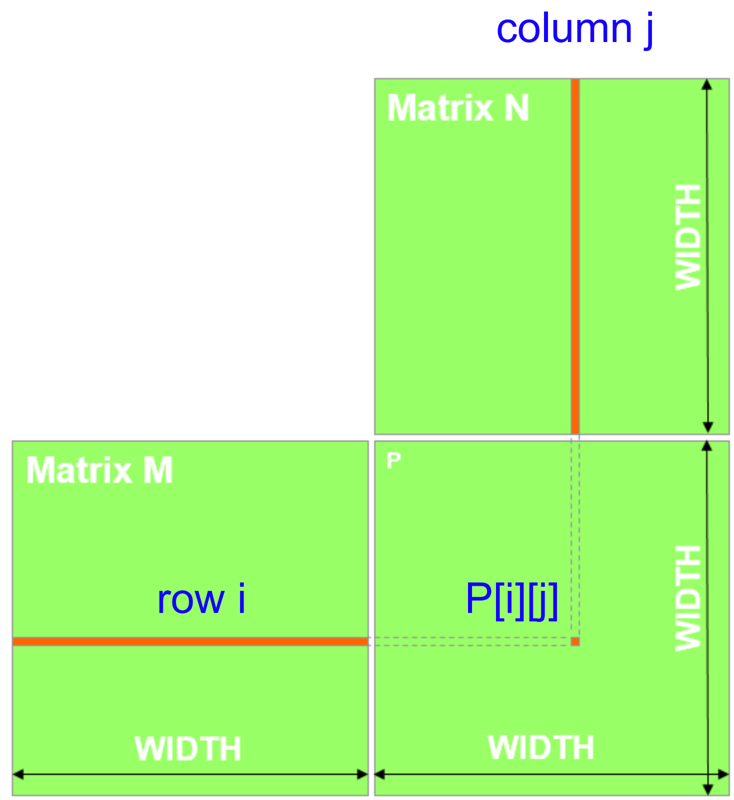
- Ex Thread (2,3) in block (0,1) assigned to row 1*4+3=7, column 0*4+2=2.
- Every thread mapped to unique (row, column).
- Every element of P assigned some thread.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $P_{0,0}$ | $P_{0,1}$ | $P_{0,2}$ | $P_{0,3}$ | $P_{0,4}$ | $P_{0,5}$ | $P_{0,6}$ | $P_{0,7}$ |
| $P_{1,0}$ | $P_{1,1}$ | $P_{1,2}$ | $P_{1,3}$ | $P_{1,4}$ | $P_{1,5}$ | $P_{1,6}$ | $P_{1,7}$ |
| $P_{2,0}$ | $P_{2,1}$ | $P_{2,2}$ | $P_{2,3}$ | $P_{2,4}$ | $P_{2,5}$ | $P_{2,6}$ | $P_{2,7}$ |
| $P_{3,0}$ | $P_{3,1}$ | $P_{3,2}$ | $P_{3,3}$ | $P_{3,4}$ | $P_{3,5}$ | $P_{3,6}$ | $P_{3,7}$ |
| $P_{4,0}$ | $P_{4,1}$ | $P_{4,2}$ | $P_{4,3}$ | $P_{4,4}$ | $P_{4,5}$ | $P_{4,6}$ | $P_{4,7}$ |
| $P_{5,0}$ | $P_{5,1}$ | $P_{5,2}$ | $P_{5,3}$ | $P_{5,4}$ | $P_{5,5}$ | $P_{5,6}$ | $P_{5,7}$ |
| $P_{6,0}$ | $P_{6,1}$ | $P_{6,2}$ | $P_{6,3}$ | $P_{6,4}$ | $P_{6,5}$ | $P_{6,6}$ | $P_{6,7}$ |
| $P_{7,0}$ | $P_{7,1}$ | $P_{7,2}$ | $P_{7,3}$ | $P_{7,4}$ | $P_{7,5}$ | $P_{7,6}$ | $P_{7,7}$ |

**WIDTH = 8, TILE_WIDTH = 4**
**4 blocks, each with 16 threads**

# Matrix multiplication

- Let M and N be square matrices of size WIDTH. Compute P=M x N.
- Can compute in CUDA by mapping one thread to each element in output P.
  - Thread multiplies elements along a row of M and column of N and sums.

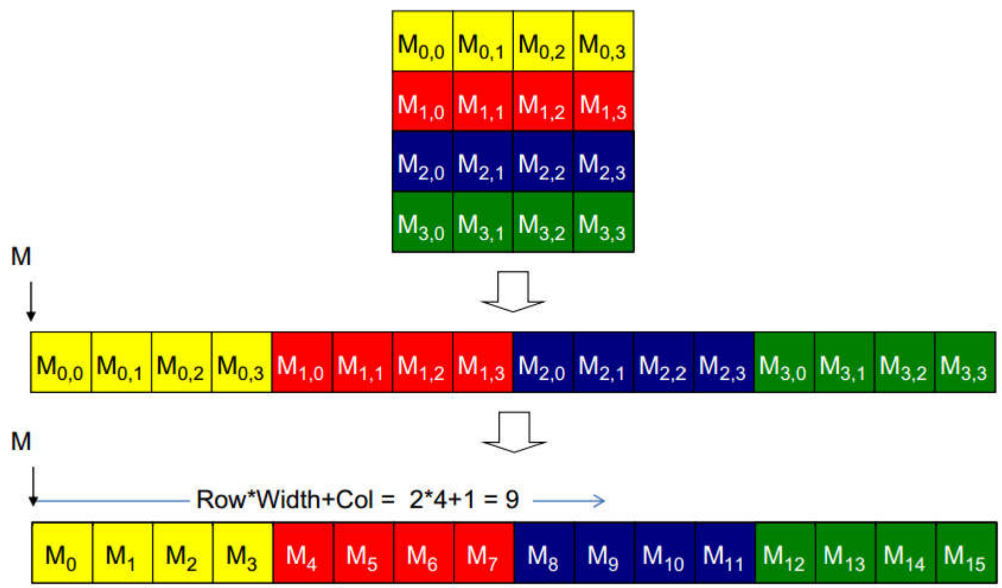



P[i][j] = sum(M[i][k] * N[k][j])
for k=0,...,n-1

# Matrix layout

- Before calling kernel, transfer matrix from host to device.
- Matrix is represented as 1D array in memory.
  - C and CUDA use row-major layout, Fortran uses column-major.
- For row major, map from 2D index to 1D

$$(row, col) \rightarrow row * width + col$$



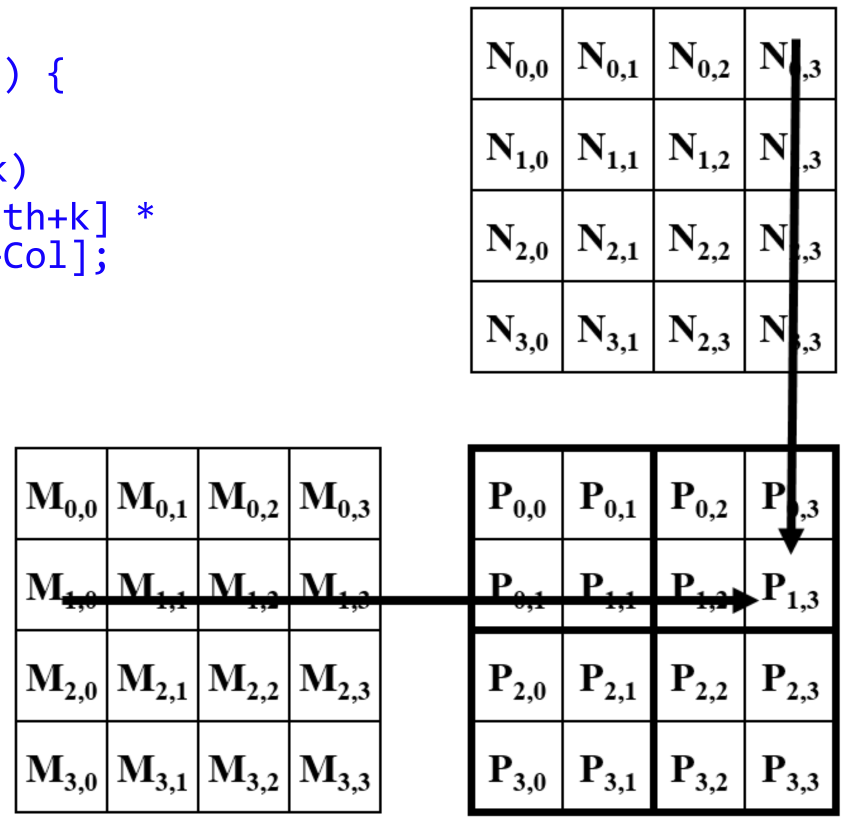
# Matrix multiplication

```
__global__ void MatrixMulKernel(float* d_M, float* d_N,
    float* d_P, int Width)
{
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        for (int k = 0; k < Width; ++k)
                Pvalue += d_M[Row*Width+k] *
                          d_N[k*Width+Col];
        d_P[Row*Width+Col] = Pvalue;
    }
}
```

row = blockId.y * blockDim.y + threadId.y

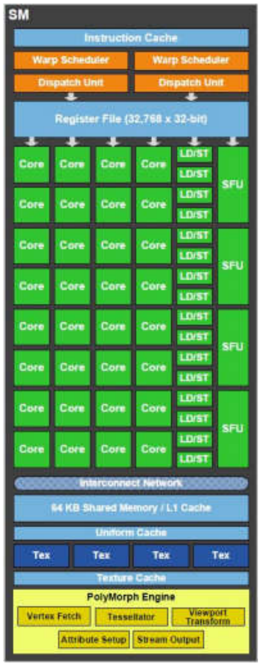
column = blockId.x * blockDim.x + threadId.x

(row, col) → row * width + col



# Why two levels of threads?

- A grid of thread blocks is easier to manage than one big block of threads.
- GPU has 1000's of cores, grouped into 10's of streaming multiprocessors (SMs).
  - Each SM has its own memory, scheduling.
  - Each SM has e.g. 64 cores (P100 architecture).
- GPU can start millions of threads, but they don't all run simultaneously.
- Scheduler (Gigathread Engine) packs up to ~1000 threads into one block and assigns the block to an SM.
  - The threads have consecutive IDs.
  - Several thread blocks can be assigned to an SM at same time.
  - Threads in a block don't execute simultaneously either.
    - They run in warps of 32 threads; more later.

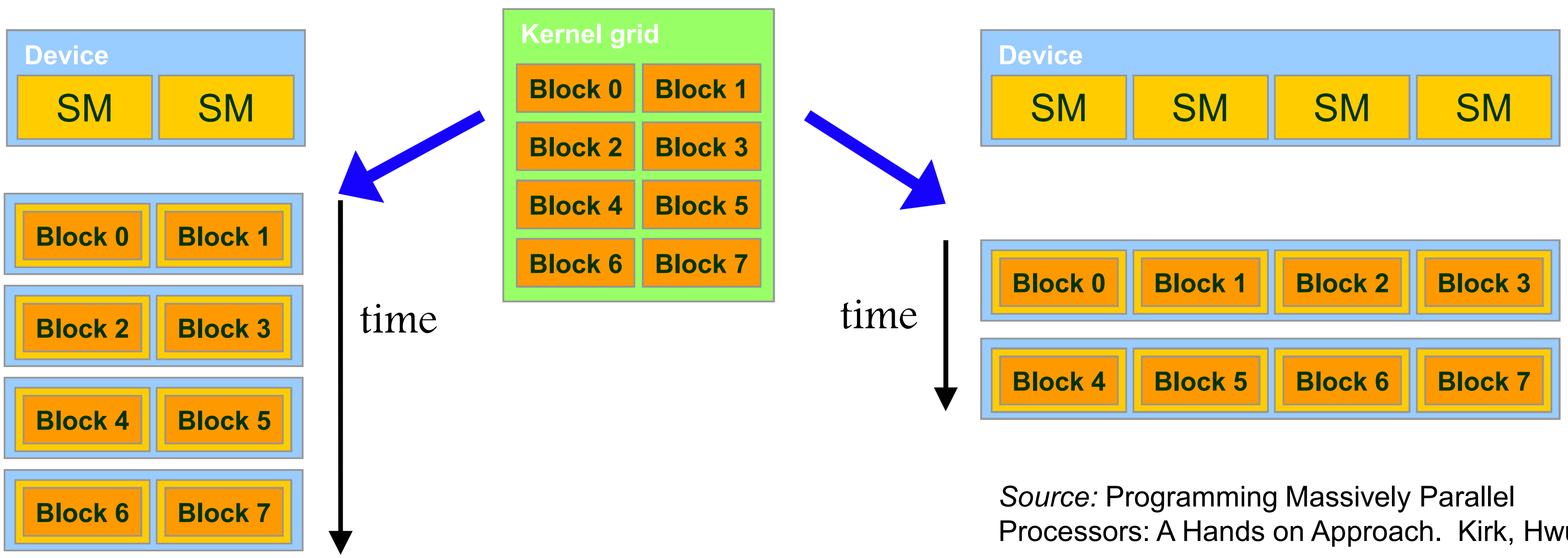


# Why two levels of threads?

- A thread block assigned to an SM uses resources (registers, shared memory) on the SM.
  - All assigned threads are pre-allocated resources.
    - Since we know the block size when we invoke the kernel, the SM knows how much resources to assign.
  - This makes switching between threads very fast.
    - No dynamic resource allocation.
    - SM has huge number (e.g. 64K) of registers, so no register flush when switching threads.
- Each SM has its own (warp) scheduler to manage threads assigned to it.
- When all threads in a block finishes, the  resources are freed.
- Then Gigathread Engine schedules a new block to the SM, using the freed resources.
- At any time, SM only needs to manage a block of a few thousand threads, instead of entire grid of millions of threads.

# Synchronization

- Different blocks can execute in any order.
  - Allows CUDA to easily scale to more SMs on higher end GPUs.
  - Ex For 2 SM GPU, can assign blocks 0,1,2,3,4,5... For 4 SM GPU, assign 0,1,2,3,4,5,6,7...
- Drawback is different blocks can't synchronize, e.g. can't force block 2 to run after block 1 finishes.
  - Your code must not depend on a particular block ordering.




*Source:* Programming Massively Parallel Processors: A Hands on Approach. Kirk, Hwu

# Synchronization

- Suppose you want to synchronize blocks, e.g. make sure some blocks do statement 1 before other blocks do statement 2.
- Can only do this by putting 2 statements in different kernels.
  - Launch first kernel with all blocks doing statement 1.
  - Then launch second kernel with all blocks doing statement 2.
  - Kernel launches relatively expensive, so this is an expensive form of synchronization.
- Threads within a block can do barrier synchronization using `__syncthreads()`.
  - More on this in later lecture.

# Choosing the right block size

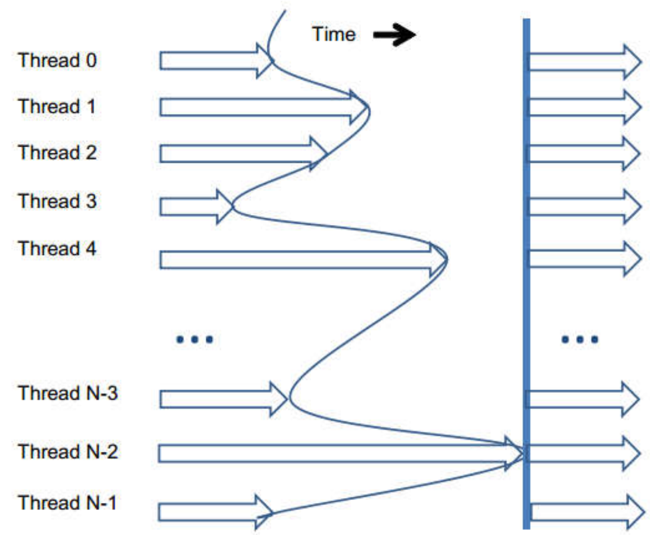
- Ex In matrix multiplication, should we use 8x8, 16x16 or 32x32 blocks?
- This is based on 3 main considerations.
- Goal Maximize number of simultaneously active threads (occupancy) on each SM.
  - More on reasons why next time.
- Consideration 1 Must satisfy several hardware constraints.
  - Following numbers are examples.
  - $\leq$ 1536 threads assigned to an SM at once.
  - $\leq$ 8 blocks assigned to an SM at once.
  - $\leq$ 512 threads per block.
- If 8x8 blocks, then 64 threads/block. Need 1536 / 64 = 12 blocks to fully occupy SM. Too many blocks.
- If 16x16 blocks, then 256 threads/block. Use 6 blocks to occupy SM. OK.
- If 32x32 blocks, then 1024 threads/block. Too many threads per block.

# Choosing the right block size

- Consideration 2 The complexity of each thread.
- Suppose each SM has 16K registers, and each thread uses 20 registers.
- If 256 threads / block, each block uses 5120 registers.
  - Can run 3 blocks = 768 threads. 50% occupancy, since SM can run 1536 threads.
- If 512 threads / block, each block uses 10240 regs.
  - Can run only 1 block = 512 threads. 33% occupancy.
- Nvidia provides a "CUDA Occupancy Calculator" to help calculate number of runnable threads based on your kernel and hardware.

# Choosing the right block size

- Consideration 3 Thread work imbalance.
- Scheduler only frees block from SM when all threads in block finish.
- With big blocks, more likely to have straggler threads.
  - Even though threads run same code, due to branching some code paths can be longer.
  - Stragglers prevent SM resources from being freed.
  - But they also don't occupy the SM, leading to waste.
- With smaller blocks, more likely threads finish at similar times. Less waste.
- Barrier synchronization within block can also cause threads to wait for each other, i.e. waste.



# Finding hardware parameters

- Hardware parameters saw change over time. To get parameters for your device, use:

```
cudaDeviceProp dev_prop;
cudaGetDeviceProperties(&dev_prop, 1)
dev_prop.maxThreadsPerBlock
dev_prop.multiProcessorCount     // how many total SMs
dev_prop.maxThreadsDim[i]        // i = 0,1,2 for x,y,z
dev_prop.maxGridSize[i]          // i = 0,1,2 for x,y,z
dev_prop.clockRate
```

- Many other parameters. See *CUDA Programming Guide.*