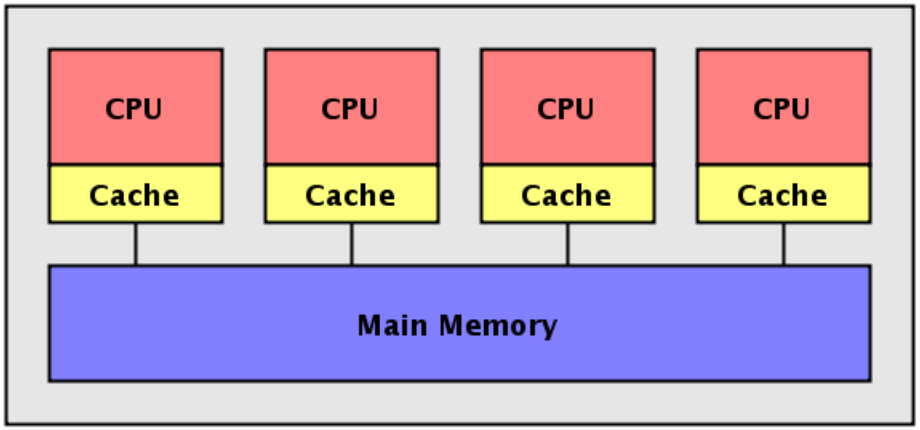
Lecture 13: Parallel Computing with OpenMP and CUDA

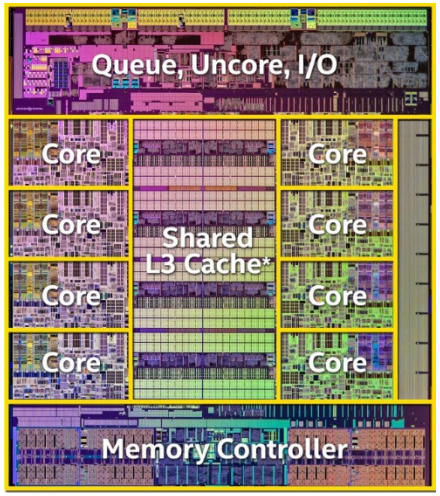
CS286

Fall 2020

Rui Fan

Shared memory multiprocessor





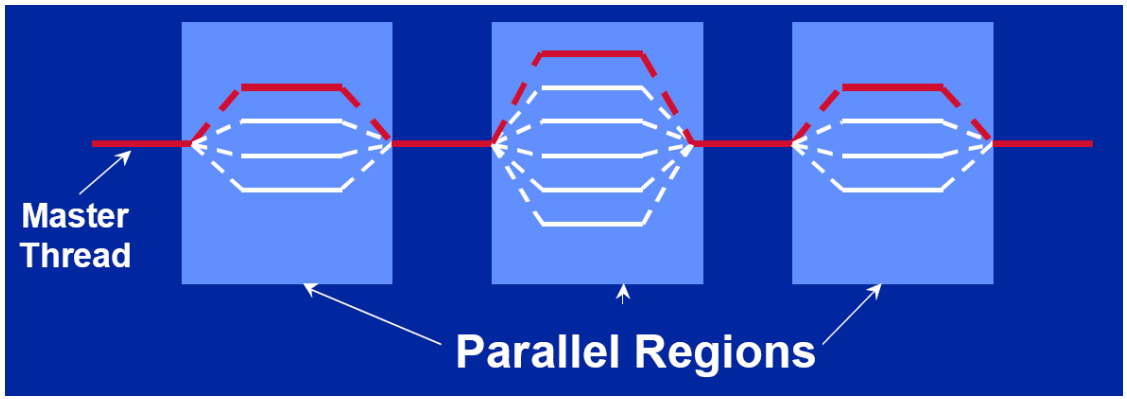
- Any memory location is accessible by any of the processors.
- A single address space exists.
  - Each memory location is given a unique address within a single range of addresses.
- Generally, more convenient than distributed memory programming, but less scalable.

OpenMP

- OpenMP is a standard for shared memory programming adopted by many hardware vendors.
- Can be used with different languages, e.g. C, C++ and Fortran.
- Compiler directives are used to specify parallelism and to indicate shared data.
- An OpenMP compatible compiler produces parallel program using the directives. A noncompatible compiler produces correct sequential program using same code.
  - Several OpenMP compilers available, e.g. Intel C compiler.
- Can be used to add parallelism incrementally to a sequential program, e.g. by parallelizing for loops.
- Underneath, OpenMP still uses threads.
  - OpenMP gives a more convenient, succinct way to manage threads.
  - But it lacks some of the expressiveness of explicit threading.

OpenMP

- OpenMP is based on threads, and uses the “fork-join” model.
  - Initially, a single master thread exists.
  - Parallel regions (sections of code) can be executed by a team of threads.
  - Compiler takes care of creating and coordinating threads.
- Available for C / C++ and Fortran. Documentation at http://openmp.org/wp/openmp-specifications/



Parallel regions

- The `parallel` directive forks a team of threads, each of which executes the following region, enclosed in {...}.

```
#pragma omp parallel
structured-block // { ... code ... }
```

- Threads do a join at end of parallel region, and execution resumes with the single master thread.
- Number of threads can be set by
  - `num_threads` clause after the parallel directive.
  - `omp_set_num_threads()` library routine previously called.
  - Environment variable `OMP_NUM_THREADS.`
  - Recommendation is one thread per processor / core.
- Threads can do the work in the region in parallel.
  - Can do different things based on thread ID.
  - Can share work using `for`, `sections`, `task`, etc. directives.
- Parallel regions can be nested.

Parallel regions

- Example

```
#pragma omp parallel private(iam, np)
{
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello from thread %d out of %d\n",
           iam, np);
}
```

  - All threads in parallel region run this code.
  - `iam` and `np` are private variables (i.e. instance of variable for each thread).
  - `omp_get_num_threads()` returns the number of threads n in the team used for the parallel region.
  - `omp_get_thread_num()` returns thread number (identity) in range 0 to n-1 with master thread 0.
  - Messages printed in arbitrary order.

Work sharing

- Share some work inside a parallel region among threads.
- For example, `for` construct inside a parallel region partitions iterations of the loop among the threads.

```
#pragma omp for
for(i=0; i<n; i++)
    {do_stuff(i);}
```

  - The way in which iterations are assigned to threads can be specified by an additional `schedule` clause.
- For this and other worksharing constructs:
  - Does not start a new team of threads - that is done by an enclosing `parallel` construct.
  - Implicit barrier at the end of the construct unless a `nowait` clause is included. I.e. each thread will wait at end of construct for all other threads to finish.

Schedule clause

- Used for assigning iterations of parallel `for` to threads.
- `schedule(static[,chunk])`
  - Each thread gets a chunk of iterations of size “chunk” – by default chunks approximately equal.
  - Chunks assigned in round robin order.
- `schedule(dynamic[,chunk])`
  - Each time a thread finishes its iterations, grabs “chunks” more iterations, until all have been executed – default is 1.
  - Dynamic scheduling has some overhead, but can result in better load balancing if iterations not all equal sized.
- `schedule(guided[,chunk])`
  - Each thread dynamically grabs iterations where the size starts large and shrinks down to “chunk”.
  - Dynamic load balancing with less overhead.
- `schedule(runtime)`
  - Schedule type and chunk size taken from the OMP_SCHEDULE environment variable.

Combined `parallel for`

- If a `parallel` directive is followed by a single `for` directive, they can be combined.

```
#pragma omp parallel for schedule(static)
for (i=0; i<n; i++) { a[i] = a[i] + b[i];}
```

- Several restrictions on structure of `for` loop.
  - Number of iterations `n` must not change.
  - Loop increment must be fixed.
  - Must not exit loop prematurely (with `break`, `goto`, `throw`).
  - Purpose of restrictions is so amount of work in loop can be determined at start.

Different ways to parallelize for

```
// sequential

for (i=0; i<N; i++) {
     a[i] = a[i] + b[i];
}
```

```
// create parallel region
// then do worksharing

#pragma omp parallel {
     #pragma omp for
     for (i = 0; i < N; i++) {
          a[i] = a[i] + b[i];
     }
}
```

```
// create parallel region and do
//worksharing together

#pragma omp parallel for schedule(static)
     for (i = 0; i < N; i++) {
          a[i] = a[i] + b[i];
     }
```

```
// manual parallelization

#pragma omp parallel {
     int id, i, Nthreads, start, end;
     id = omp_get_thread_num();
     Nthreads = omp_get_num_threads();
     start = id * N / Nthreads;
     end = (id + 1) * N / Nthreads;
     for (i = start; i < end; i++) {
          a[i] = a[i] + b[i];
     }
}
```

```
// threads do redundant work

#pragma omp parallel {
     for (i = 0; i < N; i++) {
          a[i] = a[i] + b[i];
     }
}
```

Other work sharing constructs

- Sections construct
  - Each thread assigned some sections of work.
  - Threads can be assigned 0, or multiple sections of work.
  - There's an implicit barrier at end of sections block, i.e. threads wait for each other to finish all sections before executing code after section.
  - Can turn off barrier using `nowait`.

```
#pragma omp parallel {
    #pragma omp sections {
        #pragma omp section
        { // do stuff }
        #pragma omp section
         { // do stuff }
        ...
    }
}
```

Other work sharing constructs

- Single construct
  - Structured block is executed by one thread of parallel region only (not necessarily master thread).
  - Barrier implied unless use `nowait`.
  - For doing tasks that should only be done by one thread when inside a parallel region.
- Master construct
  - Structured block is executed by master thread only. No implicit barrier at end.

```
#pragma omp parallel {
    #pragma omp single {
    // do stuff
    }
}
```

```
#pragma omp parallel {
    #pragma omp master {
    // do stuff
    }
}
```

Data environment

- OpenMP has a shared memory programming model.
  - Some variables are shared and accessible by all threads.
  - Other threads are private, and each thread has its own copy.
- Most variables are shared by default.
  - Global and static variables are shared.
  - Variables declared in master thread shared by default.
- Some variables parallel blocks private by default.
  - Loop index of `for` / `parallel for` construct.
  - Stack variables (e.g. function argument or local variable) created during execution of a `parallel` region.
  - Automatic variables in functions called in `parallel` region.

Data environment

- Reduction combines values from threads.
  - `reduction(op : variable-list)`
    - Variables in the list must be shared in the enclosing `parallel` region.
    - Each thread initially makes a local copy of each list variable and updates it.
    - Local copies are reduced into a single global copy at the end of the construct.
    - More efficient than using a critical section.

```
#pragma omp parallel for reduction (+ : x)
for (i=0; i<n; i++) {
    x = x + a[i]; }
```

```
#pragma omp parallel for
for (i=0; i<n; i++) {
    #pragma omp critical
    {x = x + a[i];}}
```

Synchronization constructs

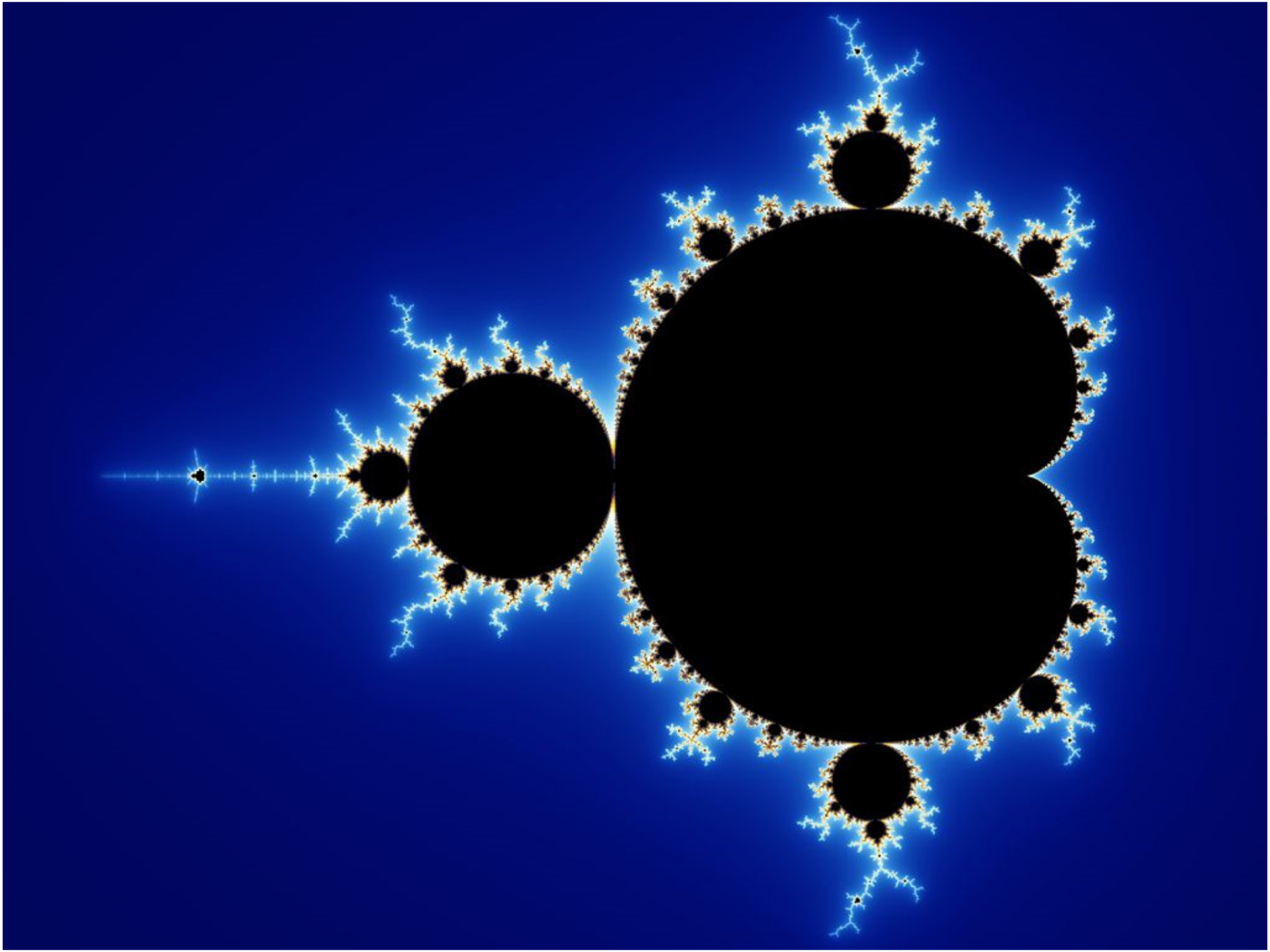
- OpenMP has critical sections and locks to protect accesses.
- Critical sections

  `#pragma omp critical [name] structured-block`
  - Only one thread can execute associated structured block at a time.
  - Name can be used to identify the critical section. Critical sections with no name default to the same.
- Locks

  `omp_init_lock(arg), omp_set_lock(arg), omp_unset_lock(arg), omp_test_lock(arg), omp_destroy_lock(arg)`
  - `arg` is a memory location.
- Critical sections protect sections of code, but locks protect data.
  - Ex Consider a hash function insert routine.
    - A critical section around the routine allows only one thread to insert at a time, even when different threads want to insert to different locations.
    - We only want to prevent concurrent inserts to same table entry. So associate one lock with each table entry.
- Barriers

  `#pragma omp barrier`

  All threads must reach the barrier before any can proceed.



Sequential routine

$z_{k+1} = z_k^2 + c$

$z^2 = (a + bi)*(a + bi)$
$= a^2 - b^2 + 2abi$

count gives colour (or intensity) to be displayed

It's known z will diverge if $|z| \geq 2$.

```
structure complex {
    float real;
    float imag;
};

int calpixel(complex c) {
    int count, max;
    complex z;
    float temp, lengthsq;
    max = 256;
    z.real = 0; z.imag = 0;
    count = 0; /* number of iterations */
    do {
        temp = z.real * z.real - z.imag * z.imag + c.real;
        z.imag = 2 * z.real * z.imag + c.imag;
        z.real = temp;
        lengthsq = z.real * z.real + z.imag * z.imag;
        count++;
    } while ((lengthsq < 4.0) && (count < max));
    return count;
}
```

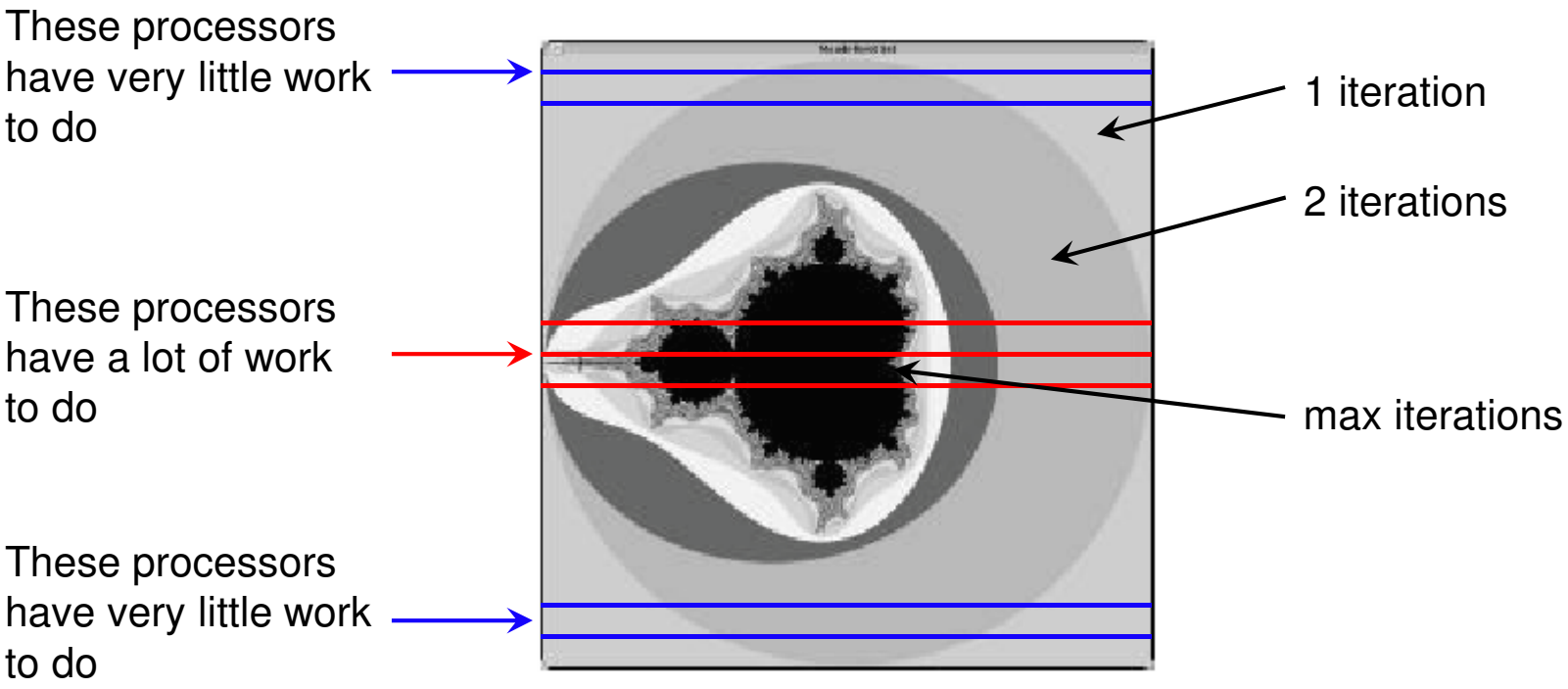
Parallelization of Mandelbrot

- Calculations for each pixel are independent.
  - Sometimes called an embarrassingly parallel computation.
- Static assignment
  - Divide the image into groups of pixels by row and assign each group to a separate thread.
  - By default, group (chunk) size is approximately equal.

```
#pragma omp parallel for private(j) schedule (static)
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        colour[i][j] = calpixel(i,j);
```

  - Not efficient as different pixels require different numbers of iterations and the computation time of different strips will vary considerably.

Static schedule

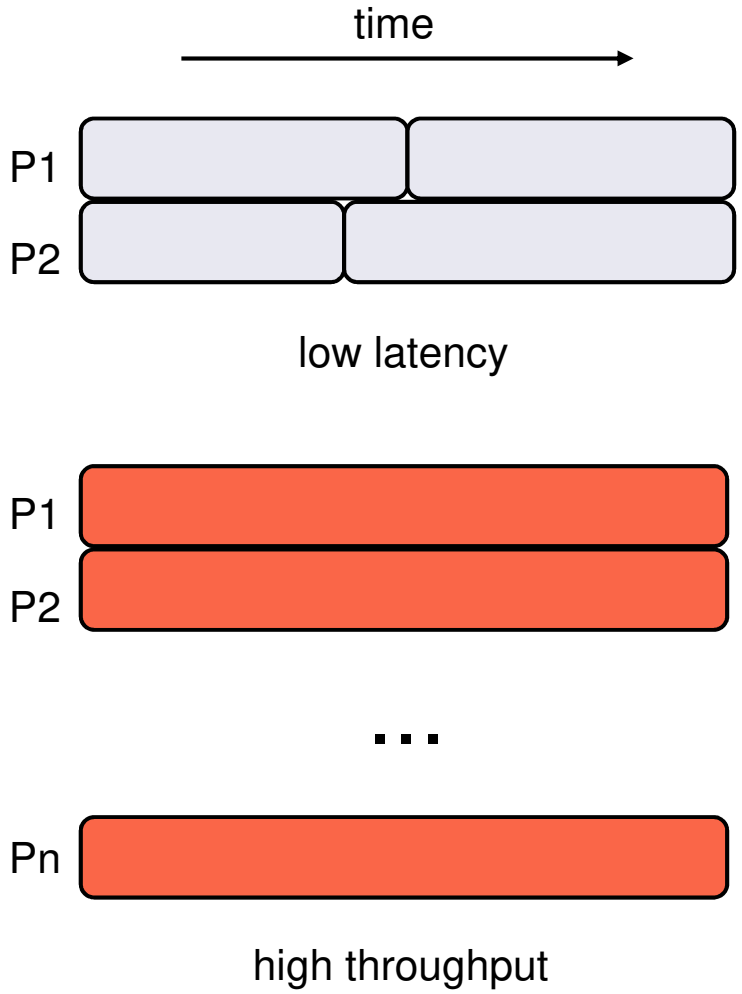


- This is a load balancing problem. Processors for top and bottom rows mostly idle, while processors for middle rows have lots of computation.

GPU Computing & CUDA

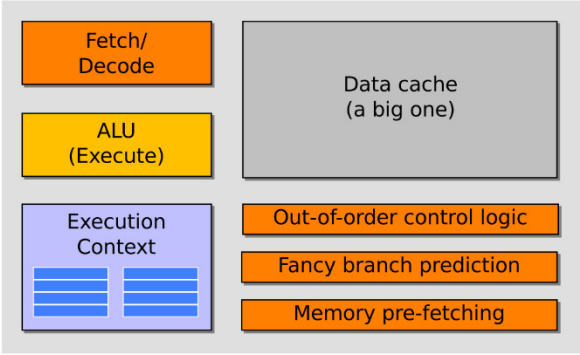
Latency vs throughput

- Multicores and GPUs take different approaches to performance.
- Multicores have a few cores, and try to minimize latency on each core.
- GPUs focus on throughput.
  - Each task is slower, but GPUs have many cores, and so can do many tasks in parallel.
- Throughput processors can do more work per unit time.



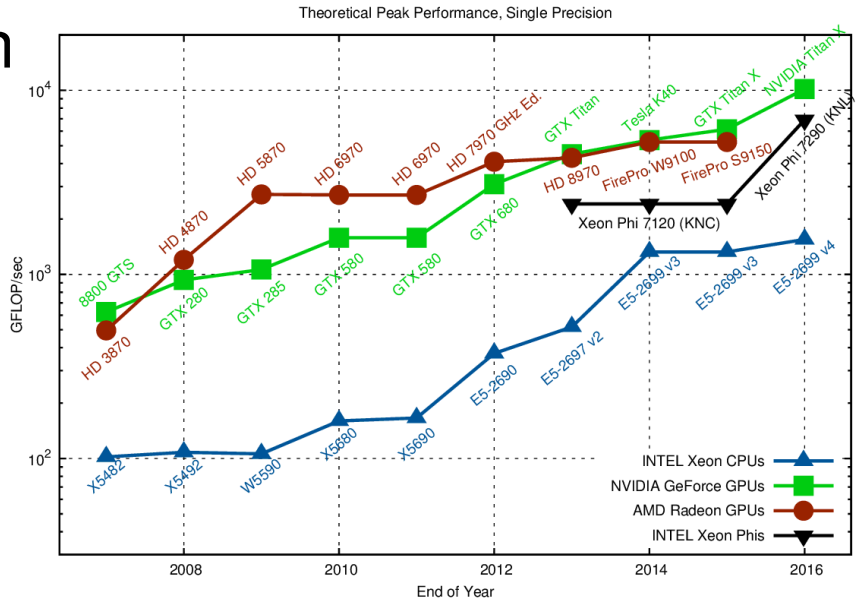
GPU vs CPU architecture

- 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.
  - Also about 50 GB/s of bandwidth, and 100s of GB of RAM.

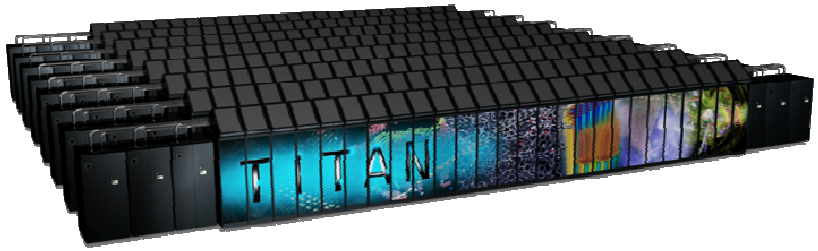


The right choice(?)

- 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).

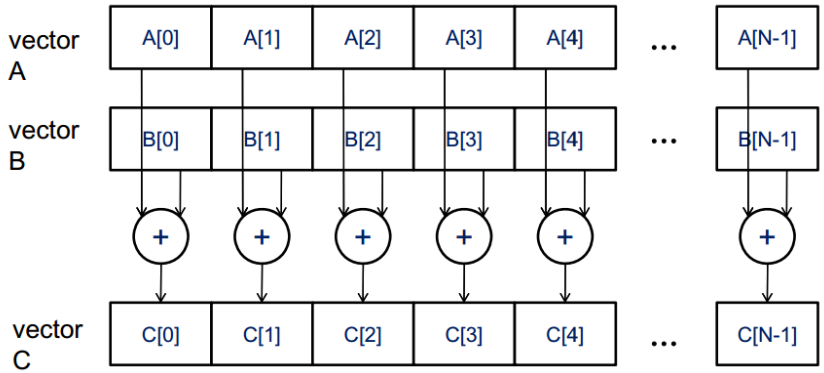


Source: https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/



Data parallelism

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



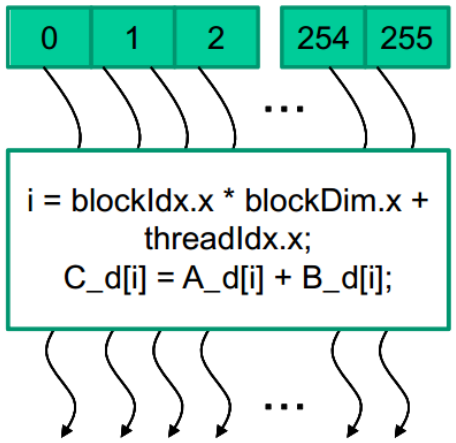
- Linear algebra (matrix-vector, matrix-matrix multiplication).
- Computer graphics.
- Data analysis (convolutions, FFT).
- Finite elements.
- Simulations.
- “Big data”, data mining and machine learning.

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];
}
```

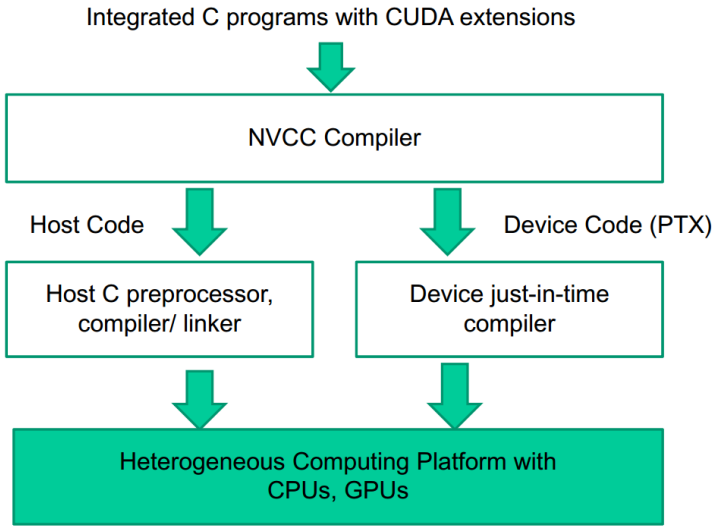
- GPU kernel launches many threads, one for each vector element.
  - Potentially millions of threads.
  - Hardware ensures low (almost zero) overhead thread management.




```
__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

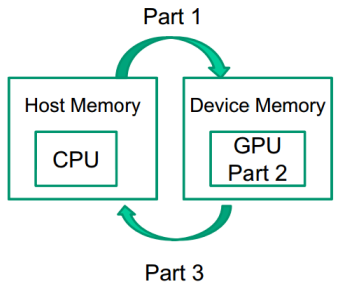
- Compute Unified Device Architecture.
- Easily use GPU as coprocessor for CPU.
- Popular Nvidia platform for programming GPUs.
  - An extension of C language.
  - Compiler, debugger, profilers provided.
- Other platforms include OpenCL and OpenACC.
  - OpenCL is similar CUDA, but more portable.
    - 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.

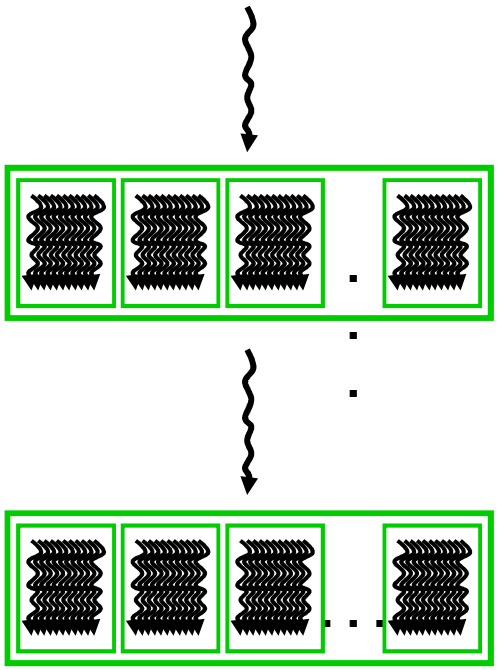


CUDA steps

- 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).

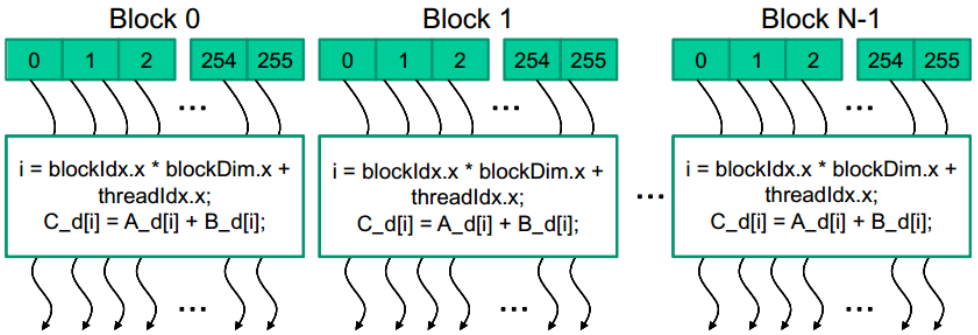
```
cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice)
```

  - 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.



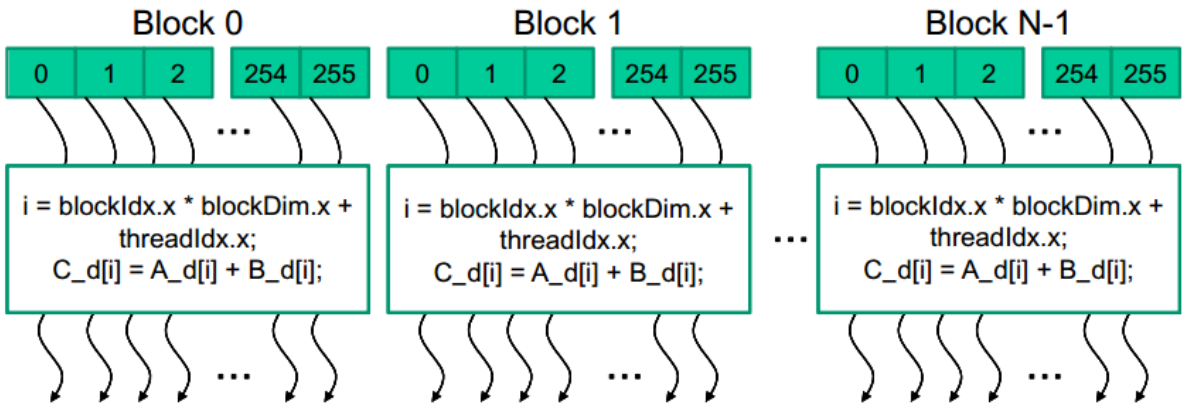
- Invoke kernel.
  - Let `n` be total # threads, `t` be # threads per block.
    - Start `ceil(n/t)`thread blocks with **`t`** threads each.
  - `KernelFunction<<<ceil(n/t), t>>>(args)`
  - `ceil` ensures we have at least n threads.

Vector addition code

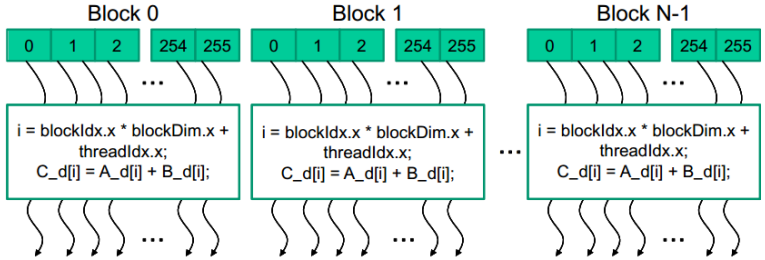
```
__global__
void vecAddKernel(float* A, float* B, float* C, int n) {
    int i = threadId.x + blockDim.x * blockId.x;
    if (i<n) C[i] = A[i] + B[i];
}

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).

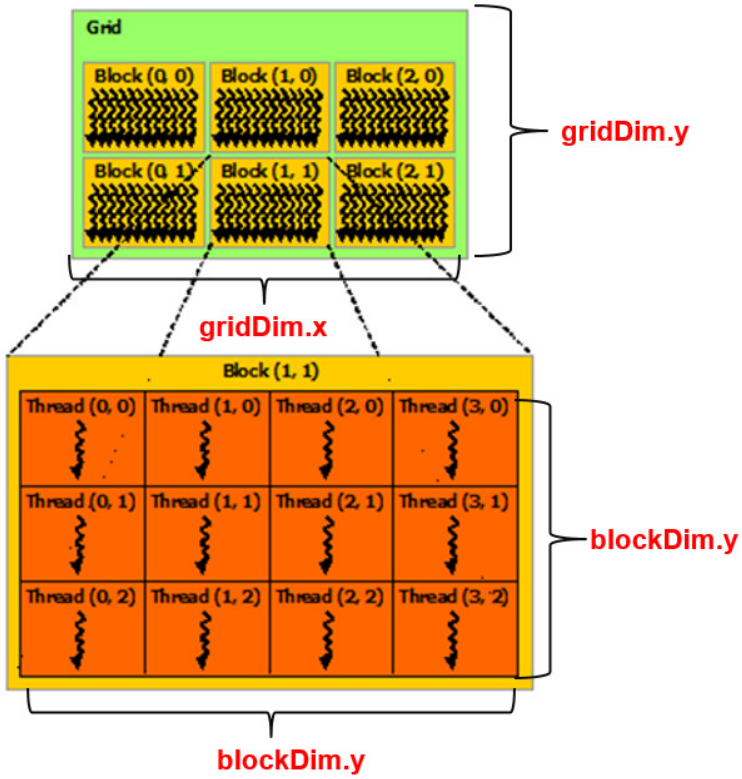
```
KernelFunction<<<ceil(n/t), t>>>(args)
```

  - 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.

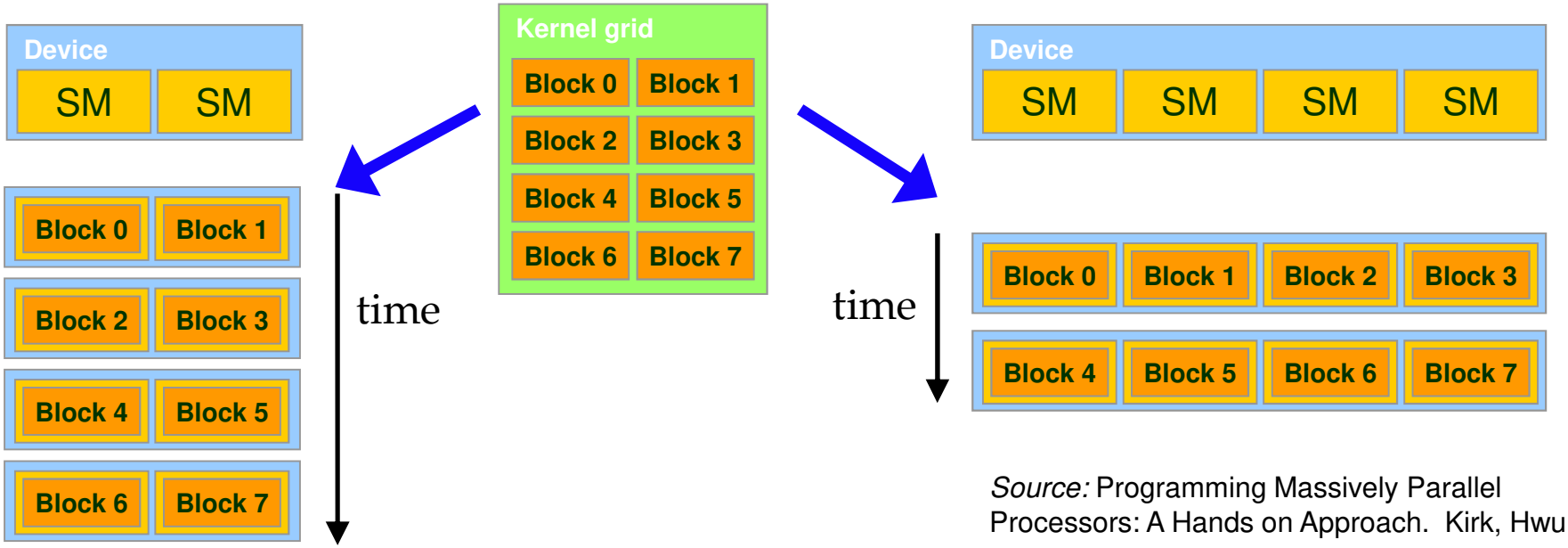
Multidimensional thread organization

- 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).



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.

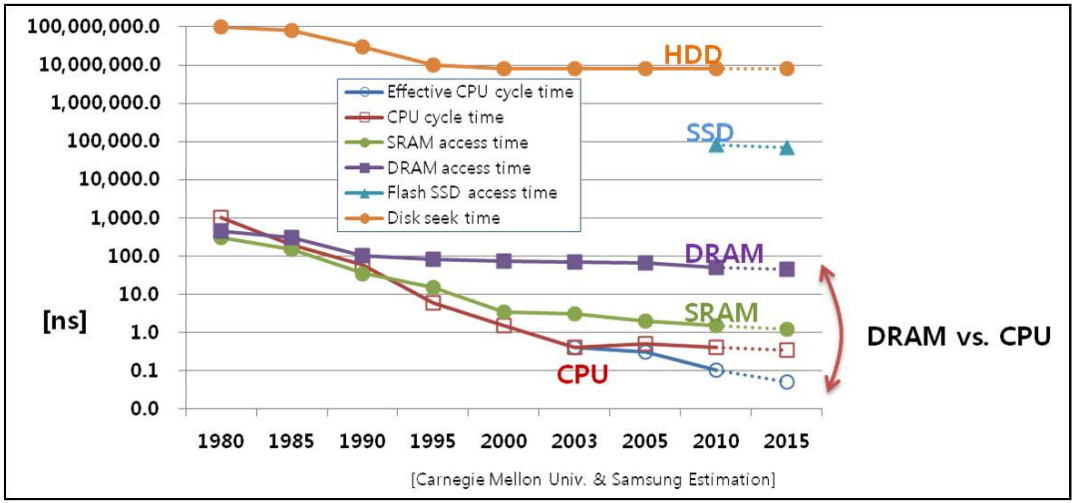


Source: Programming Massively Parallel Processors: A Hands on Approach. Kirk, Hwu

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.

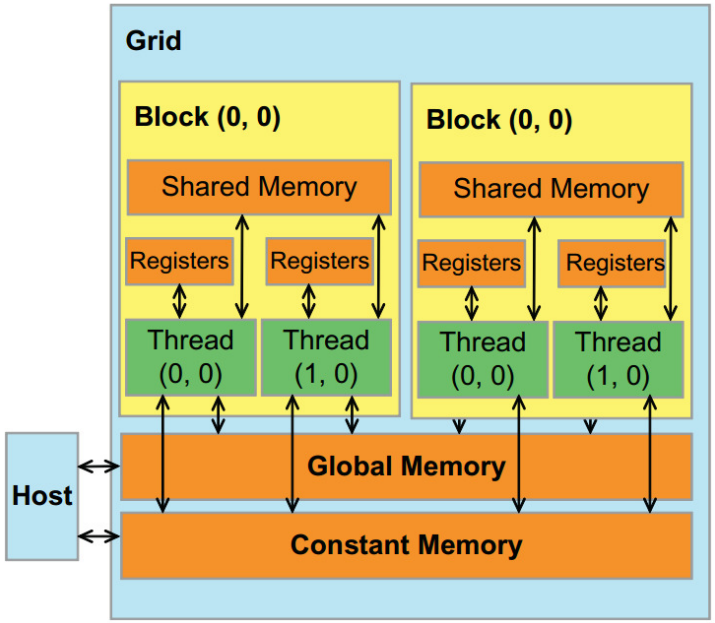
Need for speed



- Speed of code determined by amount of computation and memory accesses.
- Computation speed has been improving much faster than memory latency and bandwidth.
- Today, the main bottleneck is high memory latency and low memory bandwidth relative to CPU.
- But processors can access many different types of memory.
- Can write fast code if use right memory at right time.

GPU memory organization

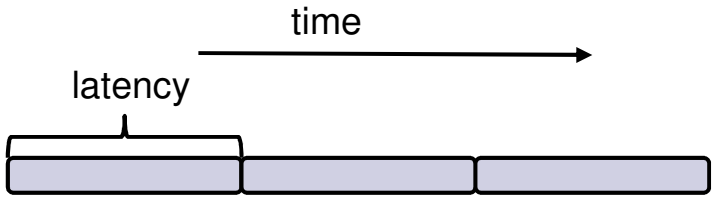
- GPU has several types of memory.
  - Different size, latency, bandwidth and scope.
  - Generally, the larger the size and scope, the slower and less bandwidth.
- Registers, shared memory, L1 cache are on-chip, much faster and higher bandwidth than global memory.
- L1 cache is controlled by hardware.
- In contrast, programmer controls what's stored in shared memory.
- Shared memory size + L1 cache size = 64KB. User configurable.

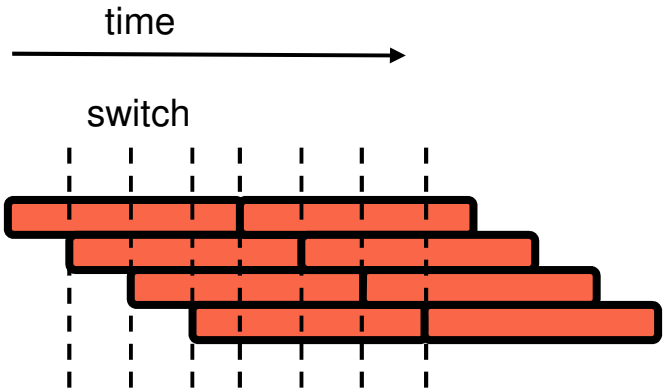


Type	Size	Latency (cycles)	Bandwidth	Visibility
Global	1-24 GB	300-800	900 GB/s	grid
Constant	64KB	cached		grid, read-only
Shared	96KB/32KB per SM	~20	14 TB/s	block
L1 cache	96KB/32KB per SM	~20	14 TB/s	block
Registers	64K per SM	~1		thread

Global memory latency

- Global memory has very high latency.
- If each thread waits (blocks) for a global memory operation to finish before doing the next operation, performance is very poor.
- Solution is to keep large pool of active threads.
- When one thread blocks doing a memory operation, switch to another thread.
  - “Massive multi-threading” (MMT).
- Total throughput high, even though each thread has high latency.





Global memory latency

- Each SM has own scheduler to do thread switching.
  - Different from device Gigathread scheduler, which allocates thread blocks to SMs.
- Each thread's context (program counter, registers) always maintained in the SM.
  - SM has ~64K registers to allocate to ~1000 threads in a thread block.
  - Very fast, "zero overhead" thread switching.
- SM scheduler has "scoreboard" to keep track of which threads assigned to the SM are blocked / unblocked.
  - Keeps picking unblocked threads to run.
- Only effective if SM has many threads, so that there always exists some unblocked threads.
  - This is why SM can run ~1000 threads, though it only has ~30 cores.
- For high performance need many threads per SM.
  - High "occupancy".

Global memory bandwidth

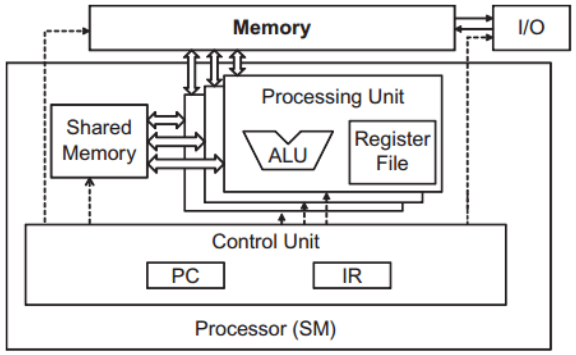
- Massive multithreading not enough for performance.
  - Only addresses latency.
  - But doesn't help with other bottleneck, bandwidth.
- GPU's computing power is much higher than its global memory bandwidth.
  - Ex Compute:1.5 TFLOPS. Bandwidth: 200 GB/s.
- Recall matrix multiplication

```
Pvalue += M[Row*Width+k] * N[k*Width+Col]
```

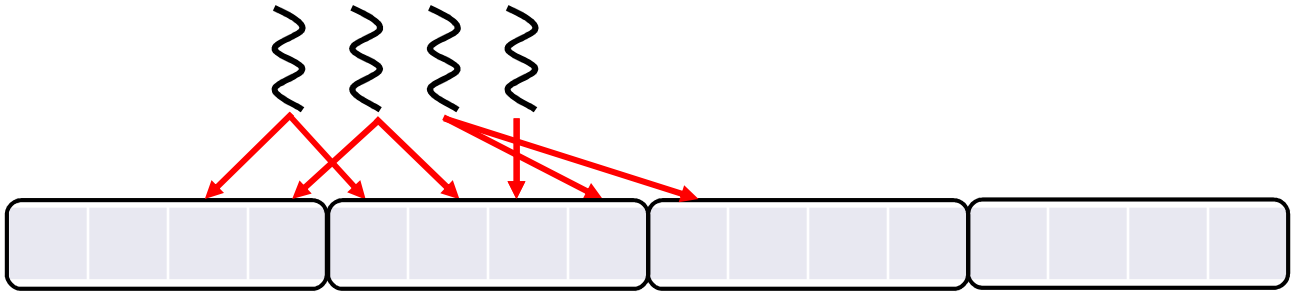
- 6 floating point ops (+, *) for 2 memory ops (read M and N).
  - Compute to global memory access (CGMA) ratio 3:1.
- 200 GB/s = 50G floating point vals / sec $\Rightarrow$ 150 GFLOPS.
  - 1/10 of theoretical peak!

Thread warps

- An SM contains one or more SIMD (single instruction multiple data) processors.
  - Each SIMD processor contains multiple cores that run the same command on different data.
- The unit of “SIMDness” is a warp of 32 threads.
  - An entire warp of threads runs at a time.
  - A thread block is divided into warps with consecutive threadIdx.x values.
- Execution is fast when entire warp “does the same thing”.
  - Different warps can do different things without performance loss.
- It’s much slower when there’s non-coalesced memory accesses, control flow divergence or bank conflicts.

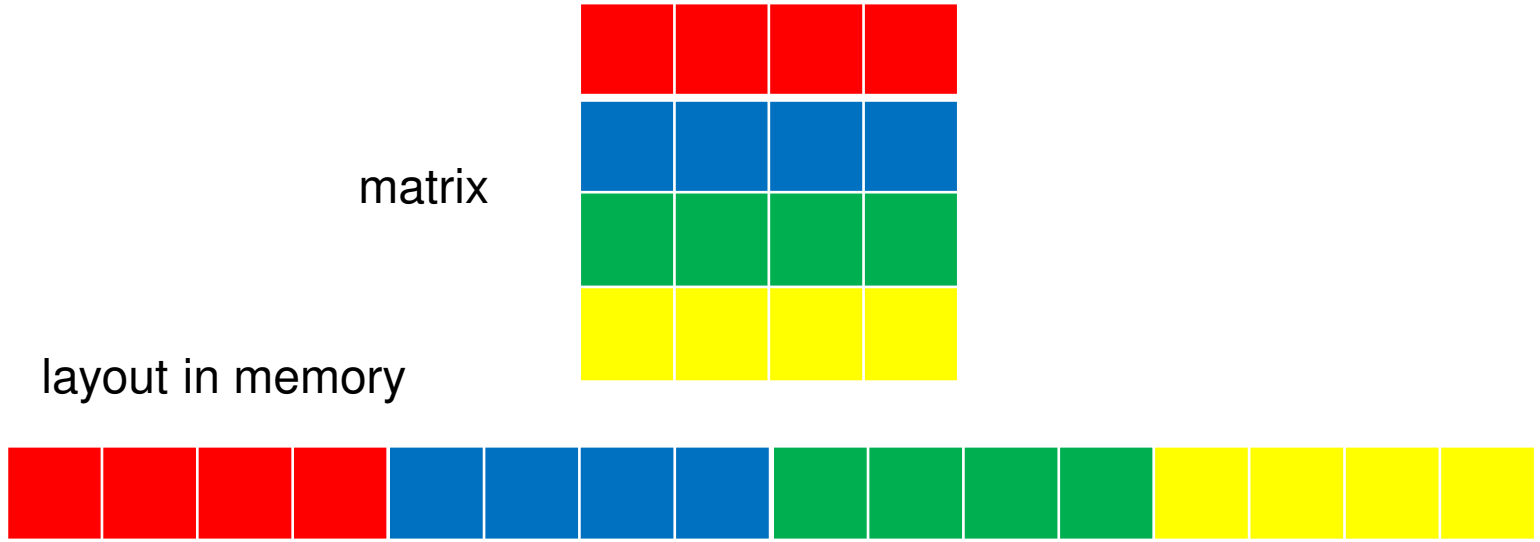


Memory coalescing



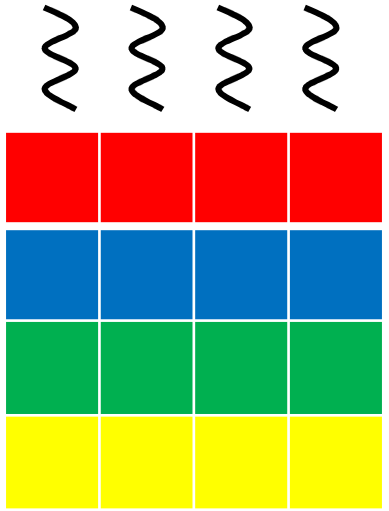
- Global memory divided into segments of 128 B (= 32 floats).
- Suppose SM executes a warp of 32 threads all executing a SIMD instruction reading from global memory.
  - If all 32 locations being read lie in one segment, hardware detects this and only transfers one segment (128 B) from global memory to SM.
    - Access is coalesced.
  - If locations lie in k different segments, k*128 B are transferred.
    - Access is uncoalesced.
  - In worst case, transfer 32*128 B = 4KB to read 32 floats!
    - Huge waste of limited global memory bandwidth.
- For good performance, make global memory accesses as coalesced as possible.

Coalescing example



- Say we have 4x4 matrix, stored in row major format.
- Suppose segments are 4 elements wide.
  - I.e. can transfer 4 consecutive elements in one step.
- We have warp of 4 threads, and want to iterate through matrix either row by row, or column by column.

Coalescing example

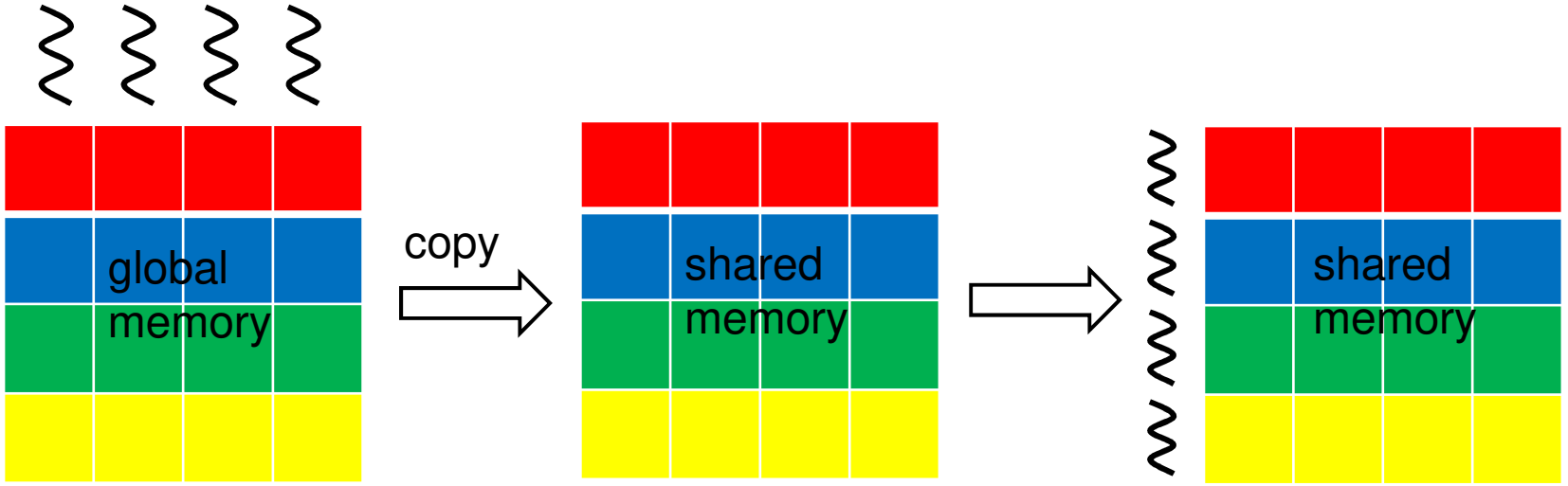


- When iterating by row, we naturally map one thread to each column.
  - Need 4 iterations in total.
- Numbers show locations accessed each iteration.
  - Locations all consecutive. All iterations coalesced.

Coalescing example

- When iterating by column, map one thread per row.
  - In iteration 1, access locations 0,4,8,12.
  - In iteration 2, access locations 1,5,9,13. Etc.
  - Each iteration accesses nonconsecutive locations.
    - All accesses noncoalesced.

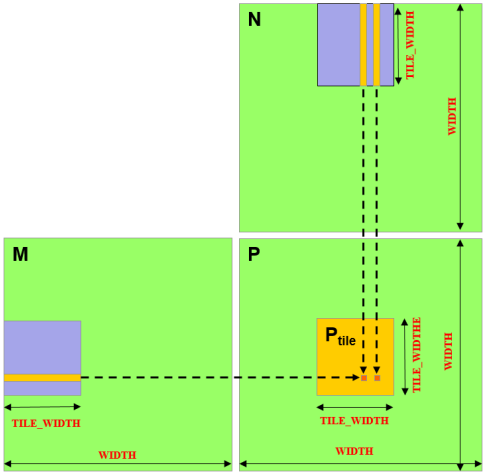
Improving coalescing



- Only global memory has bandwidth penalty for noncoalesced accesses.
- Shared memory has much smaller penalty for scattered accesses.
- To improve coalescing, first do coalesced read from global to shared memory. Then make scattered accesses to shared memory.
- Ex To read matrix by column, first read it by row and copy to matrix in shared memory. Then read shared memory matrix by column.
- Once again shows flexibility of shared memory vs global memory.

Other performance considerations

- Choosing the right block size.
  - Maximize block size subject to hardware constraints.
- Shared memory and exploiting memory reuse.
  - Saves memory bandwidth.
- Reducing warp divergence
  - Threads from warp performing k > 1 operations incur k times slowdown.
- Reducing bank conflicts
  - Shared memory is grouped into (32) banks.
  - k threads from warp accessing memory from same bank incurs k times slowdown.




```
if (threadIdx.x % 3 == 0)
      i += 1;
else if (threadIdx.x % 3 == 1)
      i -= 1;
else
      i *= 2;
```