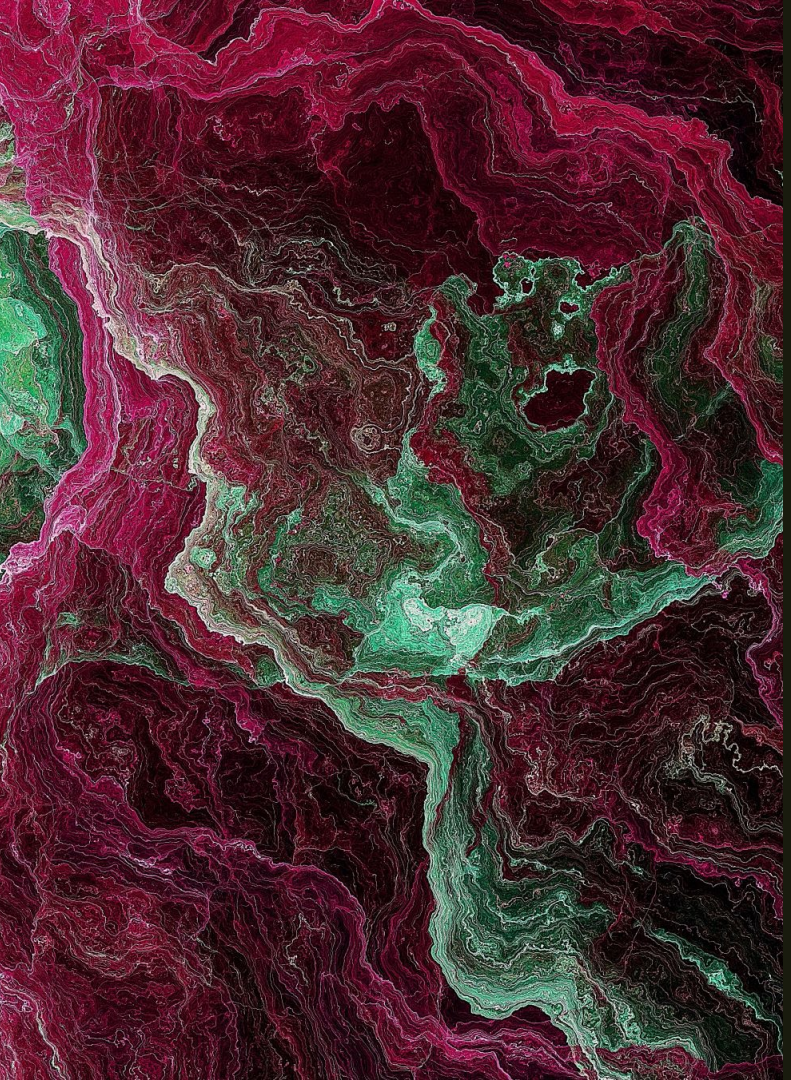
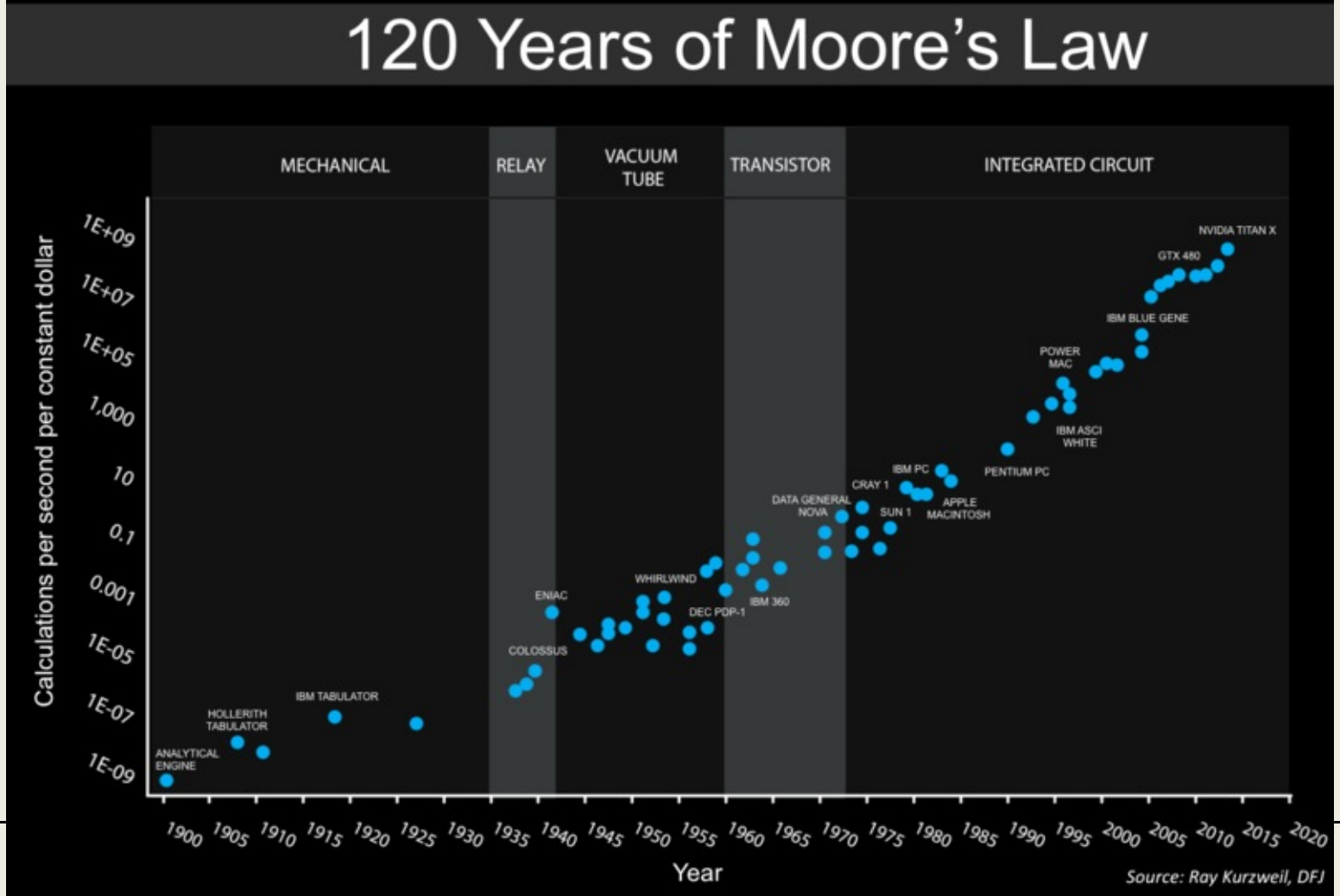


COMPUTING WITH GPUS

Dr. Steve Petruzza

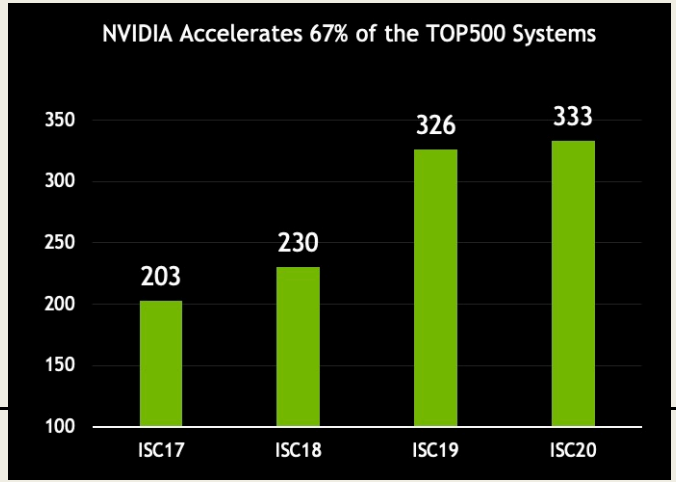
UtahStateUniversity

120 Years of Moore's Law

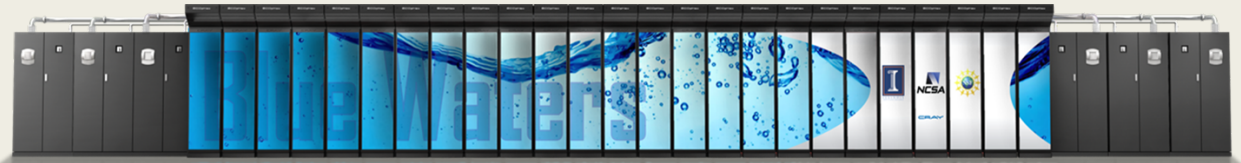


Most computing power in modern supercomputers is provided by GPUs

- Top 500 supercomputers have increased dramatically the number of GPUs: www.top500.org
- Significant research efforts are focused on algorithms implementations for “accelerators”



Blue Waters Hardware



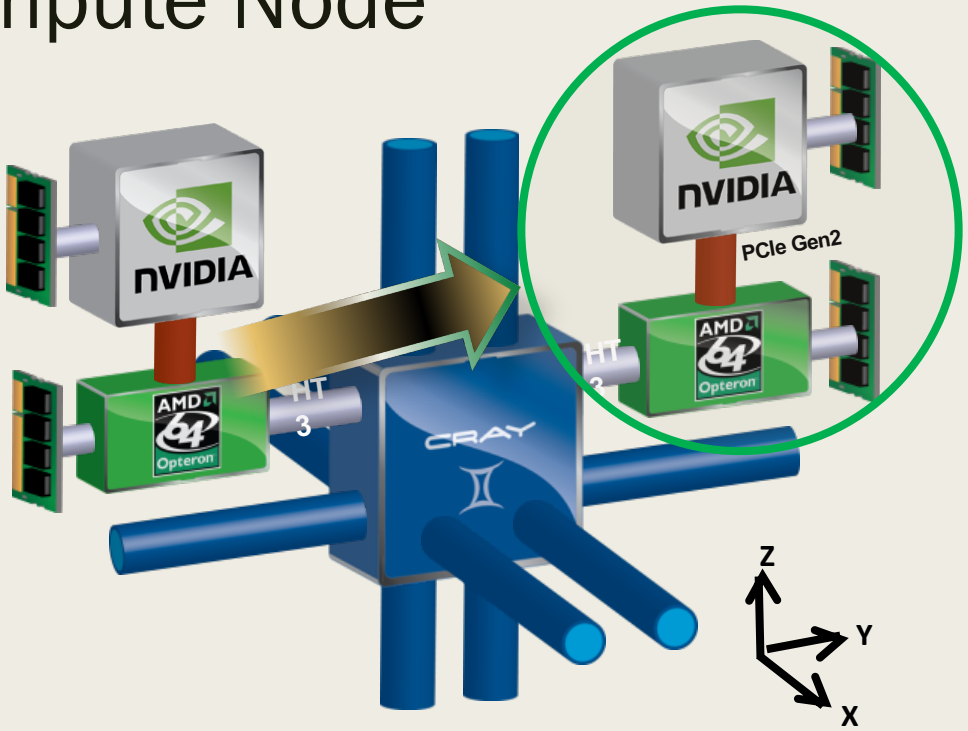
Cray System & Storage cabinets:	• >300
Compute nodes:	• >25,000
Usable Storage Bandwidth:	• >1 TB/s
System Memory:	• >1.5 Petabytes
Memory per core module:	• 4 GB
Gemin Interconnect Topology:	• 3D Torus
Usable Storage:	• >25 Petabytes
Peak performance:	• >11.5 Petaflops
Number of AMD Interlogos processors:	• >49,000
Number of AMD x86 core modules:	• >380,000
Number of NVIDIA Kepler GPUs:	• >3,000

NSF NCSA

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

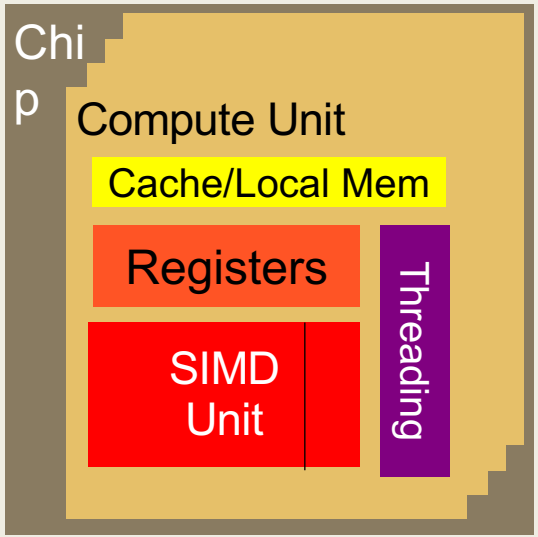
Cray XK7 Compute Node

XK7 Compute Node Characteristics
AMD Series 6200 (Interlagos)
NVIDIA Kepler
Host Memory 32GB 1600 MT/s DDR3
NVIDIA Tesla X2090 Memory 6GB GDDR5 capacity
Gemini High Speed Interconnect
Keplers in final installation

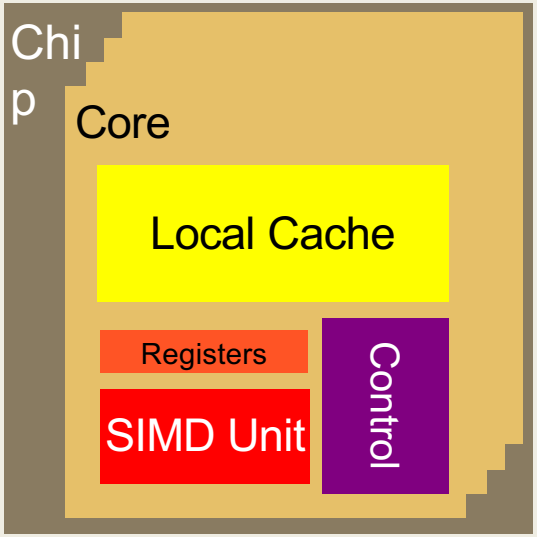


CPU and GPU have very different design philosophy

GPU
Throughput Oriented Cores

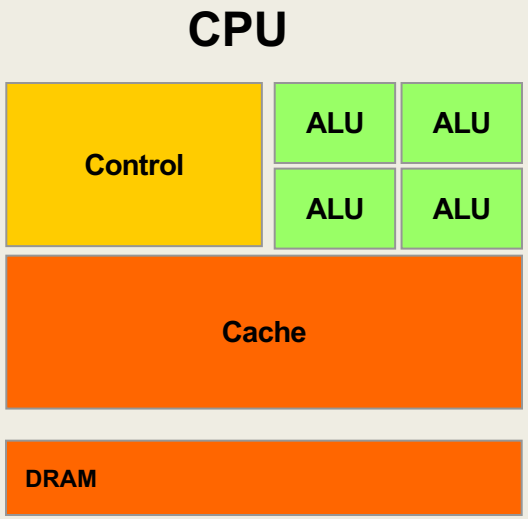


CPU
Latency Oriented Cores



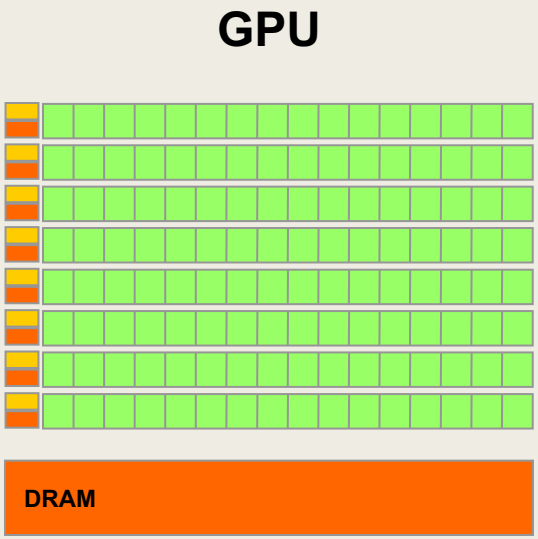
CPUs: Latency Oriented Design

- Large caches
  - *Convert long latency memory accesses to short latency cache accesses*
- Sophisticated control
  - *Branch prediction for reduced branch latency*
  - *Data forwarding for reduced data latency*
- Powerful ALU
  - *Reduced operation latency*



GPUs: Throughput Oriented Design

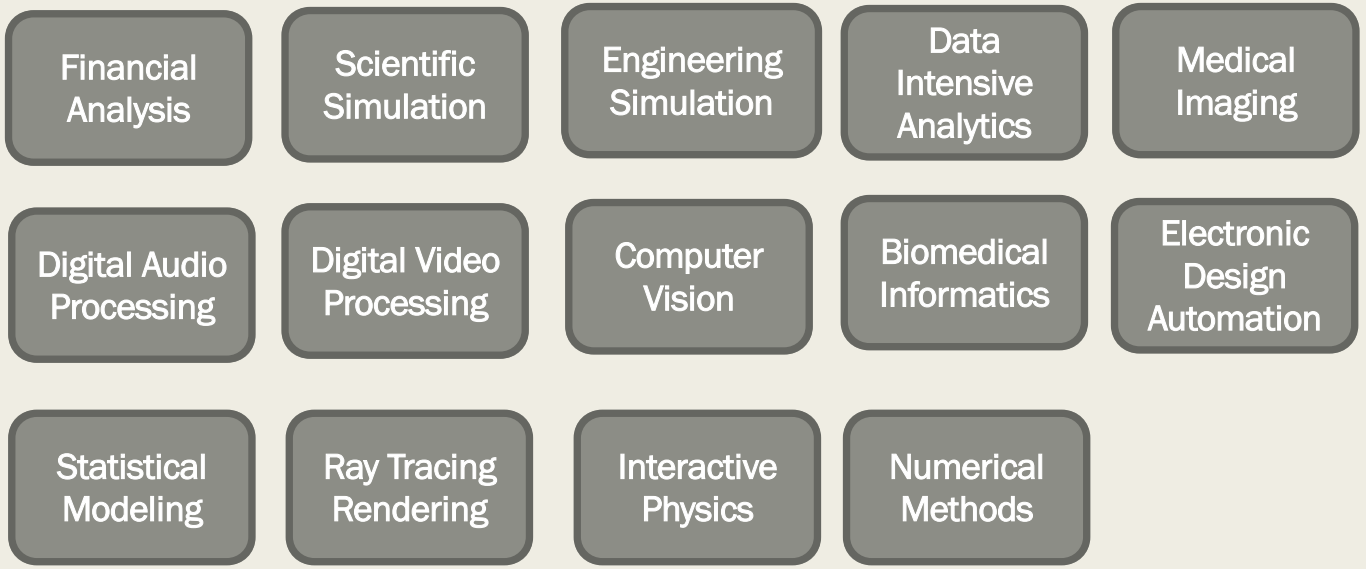
- Small caches
  - *To boost memory throughput*
- Simple control
  - *No branch prediction*
  - *No data forwarding*
- Energy efficient ALUs
  - *Many, long latency but heavily pipelined for high throughput*
- Require massive number of threads to tolerate latencies



Winning Applications Use Both CPU and GPU

- CPUs for sequential parts where latency matters
  - *CPUs can be 10+X faster than GPUs for sequential code*
- GPUs for parallel parts where throughput wins
  - *GPUs can be 10+X faster than CPUs for parallel code*

Heterogeneous parallel computing is catching on.



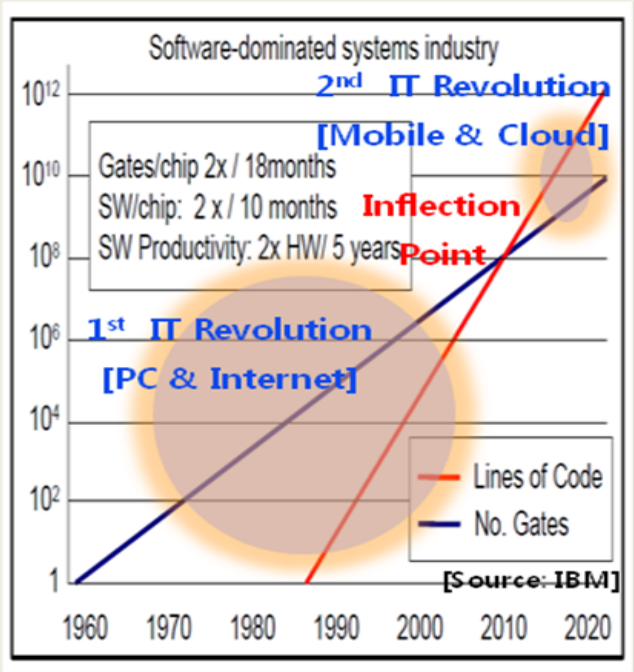
- 280 submissions to GPU Computing Gems
  - *110 articles included in two volumes*

Parallel Programming Work Flow

- Identify compute intensive parts of an application
- Adopt scalable algorithms
- Optimize data arrangements to maximize locality
- Performance Tuning
- Pay attention to code portability and maintainability

Software Dominates System Cost

- SW lines per chip increases at 2x/10 months
- HW gates per chip increases at 2x/18 months
- Future system must minimize software redevelopment



Scalability and Portability

- Performance growth with HW generations
  - *Increasing number of compute units*
  - *Increasing number of threads*
  - *Increasing vector length*
  - *Increasing pipeline depth*
  - *Increasing DRAM burst size*
  - *Increasing number of DRAM channels*
  - *Increasing data movement latency*
- Portability across many different HW types
  - *Multi-core CPUs vs. many-core GPUs*
  - *VLIW vs. SIMD vs. threading*
  - *Shared memory vs. distributed memory*

Keys to Software Cost Control

- Scalability
- Portability
  - *The same application runs efficiently on different types of cores*

Keys to Software Cost Control

- Scalability
- Portability
  - *The same application runs efficiently on different types of cores*
  - *The same application runs efficiently on systems with different organizations and interfaces*

Global Memory Bandwidth

Ideal

Reality

Conflicting Data Accesses Cause Serialization and Delays

- Massively parallel execution cannot afford serialization

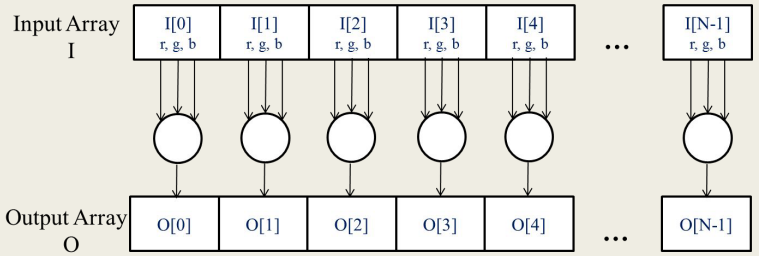
- Contentions in accessing critical data causes serialization

Data parallelism

- Parallelization across parallel computing units which operates on the data in parallel
- A data parallel algorithm which does not require any communication or synchronization between tasks is often referred also as “embarrassingly parallel”

A Data Parallel Computation Example: Conversion of a color image to grayscale

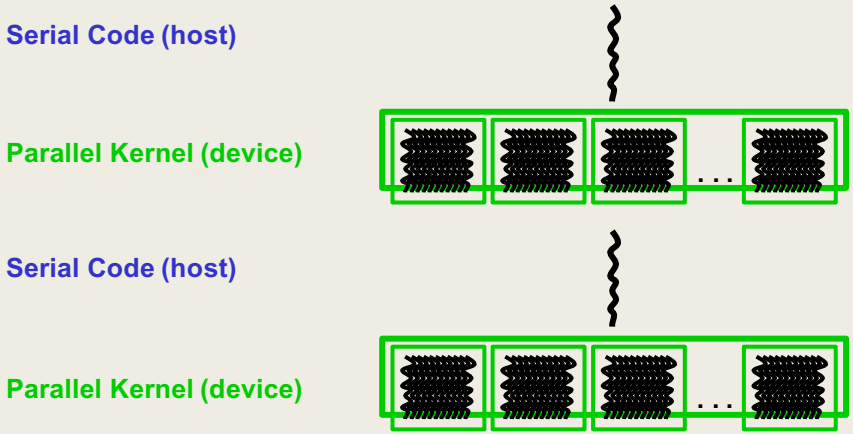
```
for each pixel {
     pixel = gsConvert(pixel)
}
// Every pixel is independent
// of every other pixel
```



3

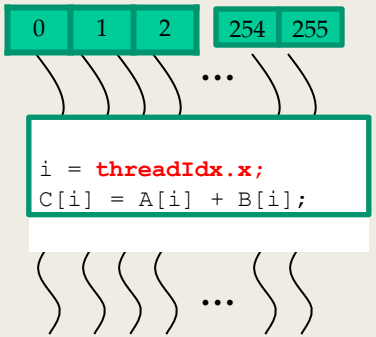
CUDA/OpenCL – Execution Model

- Typical CUDA C code (host + device)
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** C code (kernel)



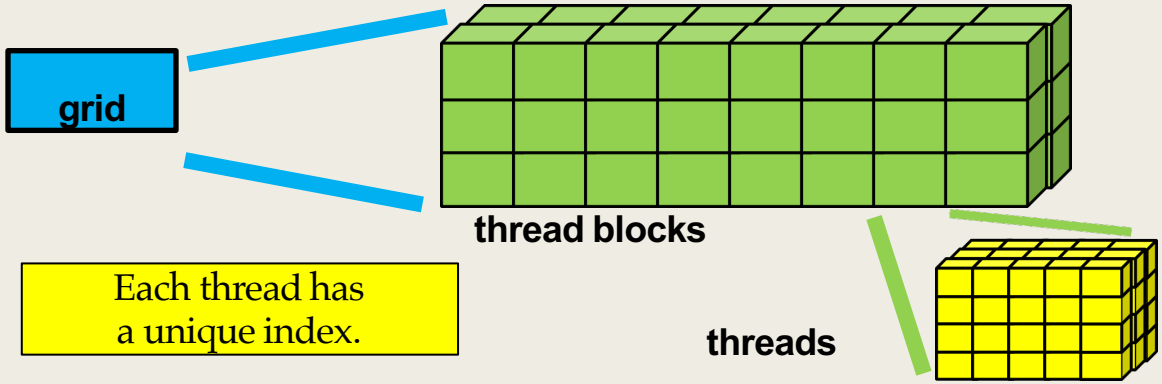
Arrays of Parallel Threads

- A CUDA kernel is executed as a grid (array) of threads
  - All threads in a grid run the same kernel code
  - Single Program Multiple Data (SPMD model)
  - Each thread has **a unique index** that it uses to compute memory addresses and make control decisions



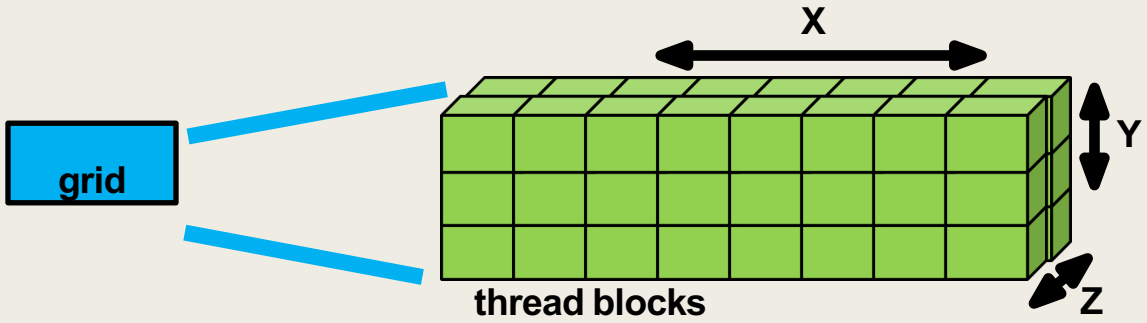
Logical Execution Model for CUDA

- Each CUDA kernel
  - is executed by a **grid**,
  - a 3D array of **thread blocks**, which are
  - 3D arrays of **threads**.



blockIdx is Unique for Each Block

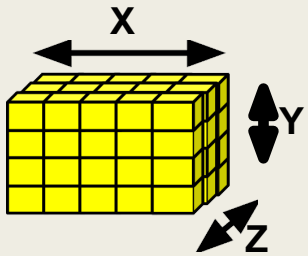
- Each block has a unique index tuple
  - blockIdx.x (from 0 to (gridDim.x – 1))
  - blockIdx.y (from 0 to (gridDim.y – 1))
  - blockIdx.z (from 0 to (gridDim.z – 1))



blockDim: # of Threads per Block

- Number of blocks in each dimension is
  - blockDim.x … 5
  - blockDim.y … 4
  - blockDim.z … 3

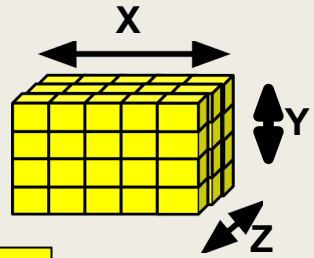
threads



For 2D (and 1D blocks), simply use block dimension 1 for Z (and Y).

threadIdx Unique for Each Thread

- Each thread has a unique index tuple
  - threadIdx.x (from 0 to (blockDim.x – 1))
  - threadIdx.y (from 0 to (blockDim.y – 1))
  - threadIdx.z (from 0 to (blockDim.z – 1))

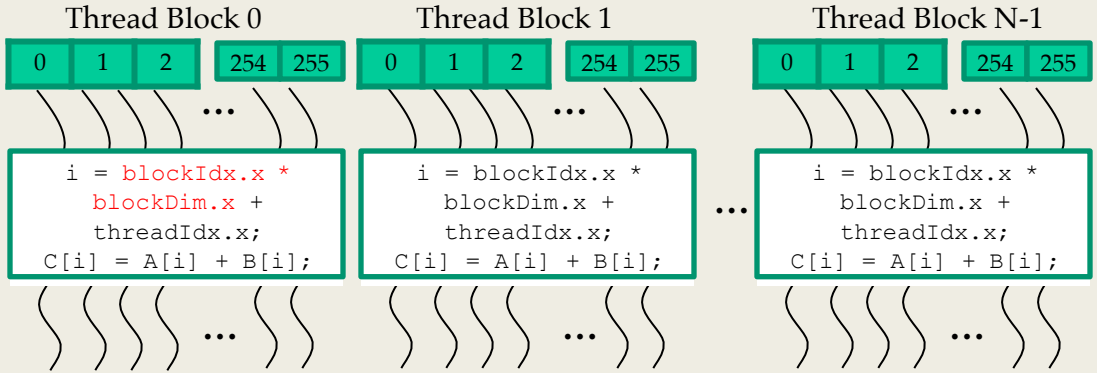


threads

threadIdx tuple is unique to each thread
WITHIN A BLOCK.

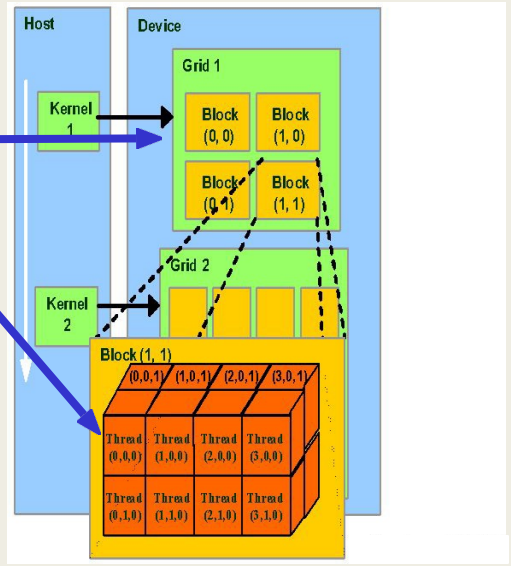
Thread Blocks: Scalable Cooperation

- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization** (to be covered later)
  - Threads in different blocks cooperate less (later)

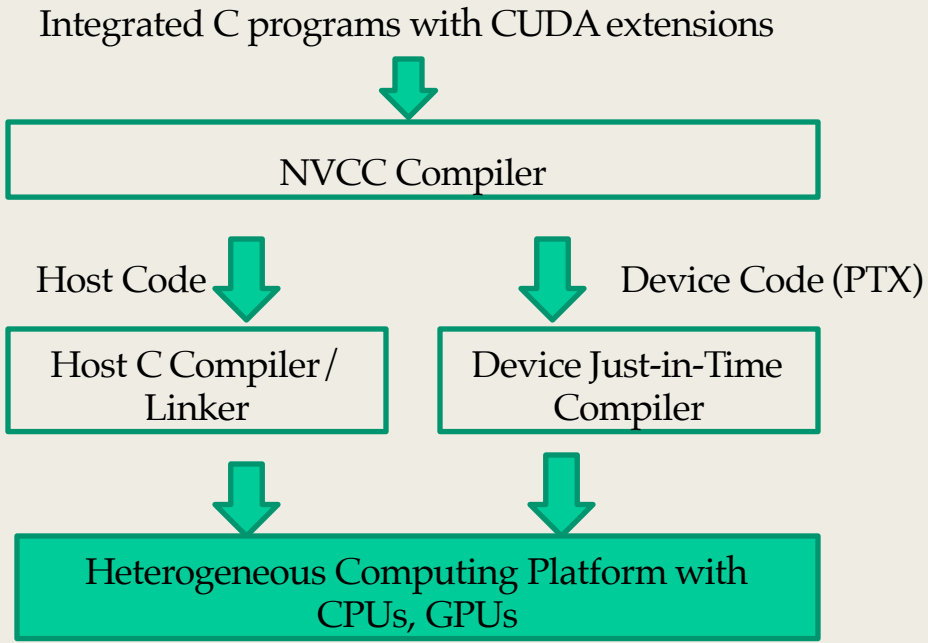


blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D
    threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Vectors, matrices, tensors
  - Solving PDEs on volumes
  - …



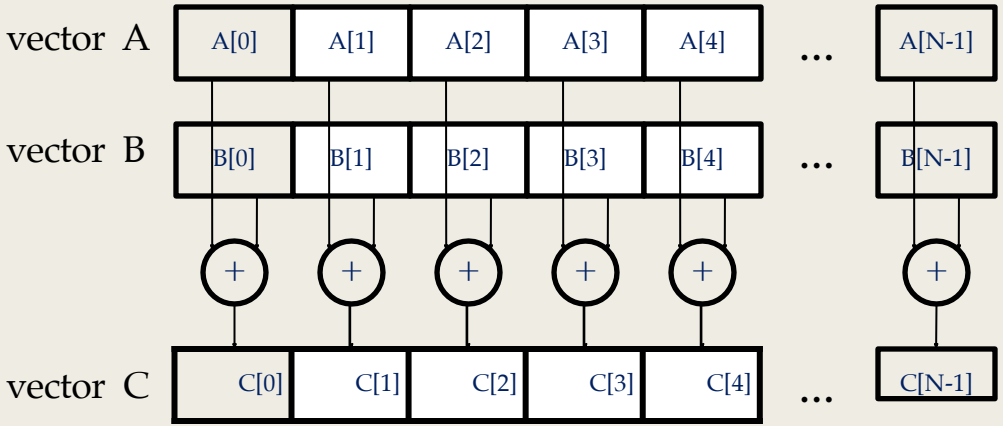
Compiling A CUDA Program



Hello world on CHPC GPU nodes

- You can now use the following partitions/allocations:
  - *kingspeak-gpu on kingspeak*
  - *owner-gpu-guest on kingspeak*
  - *notchpeak-gpu on notchpeak*
  - *owner-gpu-guest on notchpeak*
- For fast testing use: *notchpeak-shared-short* each node has 2 Nvidia Tesla k80
- Interactive allocation of 1 GPU for 15minutes
  ```
  salloc -n 1 -N 1 -t 0:15:00 -p notchpeak-shared-short -A notchpeak-shared-short --gres=gpu:k80:1
  ```
- You can use the same parameters in a batch scripts (i.e., #SBATCH <param>)
- Run **always** using **srun: `srun ./executable_name`**
- More info about CHPC accelerators: https://www.chpc.utah.edu/documentation/guides/gpus-accelerators.php

Vector Addition – Conceptual View



Vector Addition – Traditional C Code

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

int main()
{
    // Memory allocation for A_h, B_h, and C_h
     // I/O to read A_h and B_h, N elements
     …
    vecAdd(A_h, B_h, C_h, N);
}
```

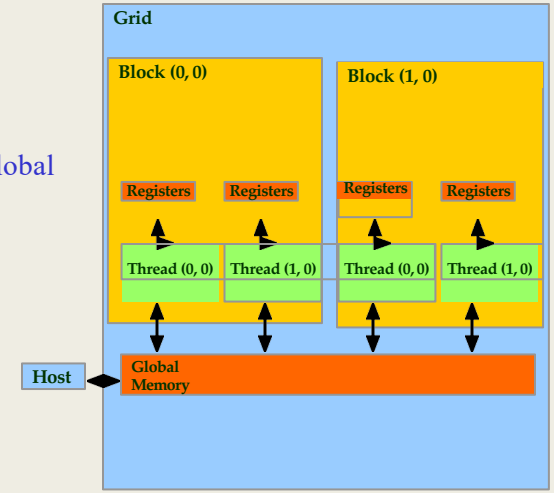
CUDA vecAdd Host Code

```
#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


}
```

Partial Overview of CUDA Memories

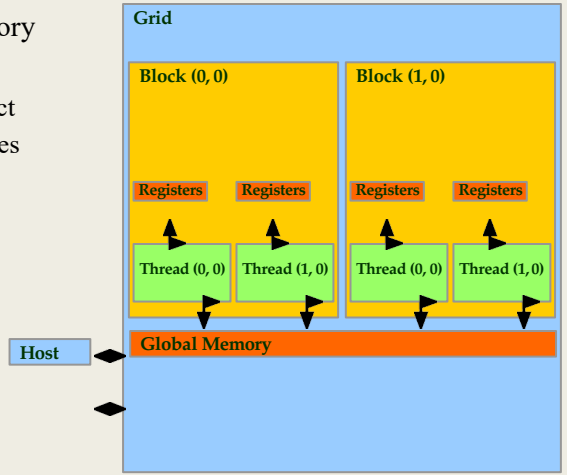
- Device code can:
  - R/W per-thread registers
  - R/W per-grid global memory
- Host code can
  - Transfer data to/from per grid global memory

We will cover more later.



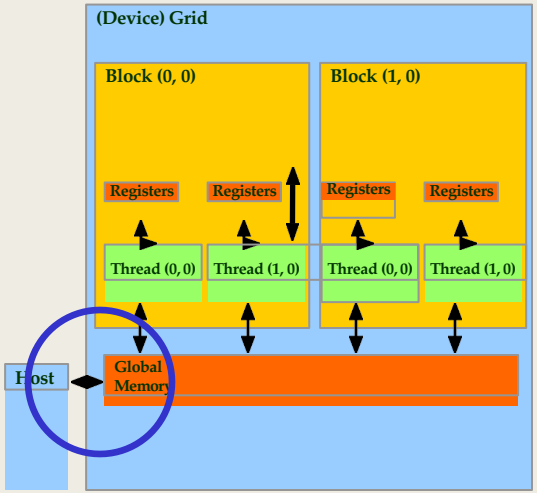
CUDA Device Memory Management API

- `cudaMalloc()`
  - Allocates object in the device global memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** the allocated object in terms of bytes
- `cudaFree()`
  - Frees object from device global memory
    - **Pointer** to freed object



Host-Device Data Transfer API

- `cudaMemcpy()`
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer



```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);  float* A_d, B_d, C_d;

    1.     // Transfer A and B to device memory  cudaMalloc((void **)
           &A_d, size);
           cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
           cudaMalloc((void **) &B_d, size);
           cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

       // Allocate device memory for  cudaMalloc((void **) &C_d, size);

    2. // Kernel invocation code – to be shown later

    3. // Transfer C from device to host
           cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
       // Free device memory for A, B, C
           cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x ;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}

int vectAdd(float* A, float* B, float* C, int n)
{
    // A_d, B_d, C_d allocations and copies omitted
    // Run ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
}
```

Example: Vector Addition Kernel

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A_d, float* B_d, float* C_d, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```

Host Code

```
int vecAdd(float* A, float* B, float* C, int n)
{
  // A_d, B_d, C_d allocations and copies omitted
  // Run ceil(n/256) blocks of 256 threads each
  vecAddKernel<<<ceil(n/256.0),256>>>(A_d, B_d, C_d, n);
}
```

More on Kernel Launch

Equivalent Host Code

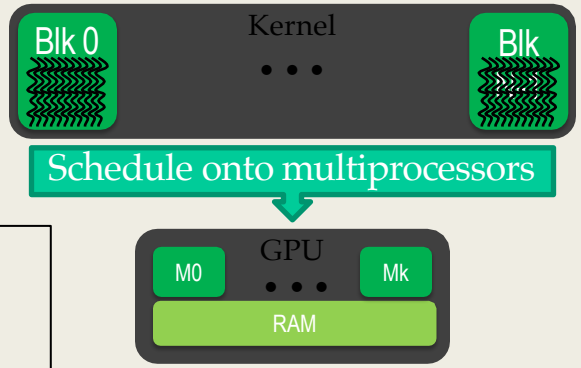
```
int vecAdd(float* A, float* B, float* C, int n)
{
 // A_d, B_d, C_d allocations and copies omitted
 // Run ceil(n/256) blocks of 256 threads each
  dim3 DimGrid(ceil(n/256), 1, 1);
  dim3 DimBlock(256, 1, 1);

  vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

Kernel execution in a nutshell

```
__host__
Void vecAdd()
{
  dim3 DimGrid(ceil(n/256.0),1,1);
  dim3 DimBlock(256,1,1);

  vecAddKernel<<<DimGrid,DimBlock>>>(A_d, B_d, C_d, n);
}
```

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

   if( i<n ) C_d[i] = A_d[i]+B_d[i];
}
```



More on CUDA Function Declarations

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

- `__global__` defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return `void`
- `__device__` and `__host__` can be used together