FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

PTFS – GPU Programming

2022-07-25

Prof. Dr. Harald Köstler – Lehrstuhl für Informatik 10, Systemsimulation

and

Dominik Ernst - NHR@FAU



TOP 500 list

- List of current best supercomputers in the world (updated twice a year)
- https://www.top500.org/lists/top500/list/2022/06/
- Out of the Top10 of the list
  - 3x AMD MI250X GPUs. Including the current #1 and first Exaflop system, Frontier at ORNL
  - 2x NVIDIA V100 GPUs. Including Summit, former #1 system
  - 2x NVIDIA A100 GPUs
  - 2x Chinese domestic matrix/many core architectures. Including Tianhe-2a and Sunway TaihuLight, both former #1 systems
  - 1x A64FX CPUs. Fugaku, former #1 system

Where Does HPC Computer Architecture Go?

"Free Lunch is Over": automatic performance increases through

- architecture improvements using more transistors (complexity)
- increased clock rates (power considerations and physical limits)

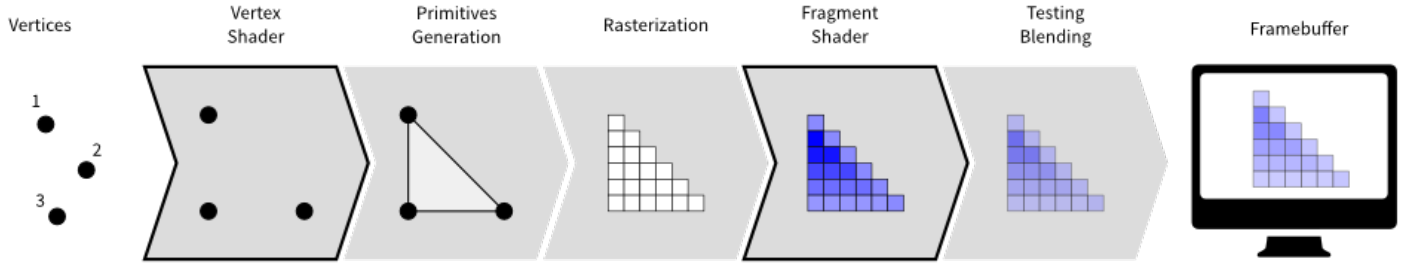
have stopped/slowed 15 years ago

Solution is to introduce parallelism on all hardware levels

- On-Node level
  - SIMD-like vectorization (on core)
  - Multi-core (with shared memory)
- Inter-node level
  - Clusters (with distributed memory and fast interconnects)

Why GPUs?

- Graphics Processing Units (GPUs) are massively parallel computer architectures for computations in real time 3D graphics (mostly: video games)



- Advanced graphics effects require increased programmability
- Feasbility for for non-graphics workloads or “General Purpose”
- GPUs offer high computational performance at relatively low costs
- Low level programming models (CUDA, OpenCL, HIP) still look like shaders



GPU vs CPU

CPUs are great for task parallelism

- Big and fast caches
- Branch prediction & speculative execution
- Prefetchers
- Low latency instructions
- Out-of-Order Execution

GPUs are great for data parallelism

- Wide SIMDs
- Large “core”-counts
- Extreme SMT

CUDA C++ Programming Guide (nvidia.com)

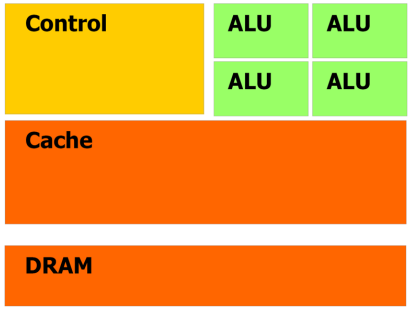
- Later Figures on slides are from this programming guide!
- **NVIDIA provides nice GPU tutorials, later examples are partly taken from these**

Trading single thread performance for parallelism: *GPGPUs vs. CPUs*

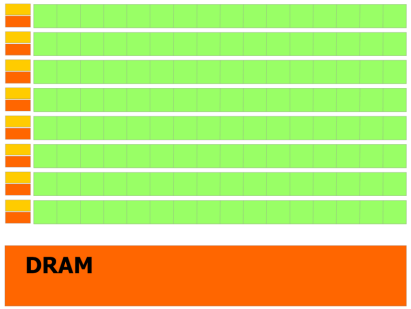
GPU/A64FX vs. CPU
light speed estimate
(per device)

MemBW ~ 2-5 x

PeakPerf (DP) ~ 3-4 x



CPU

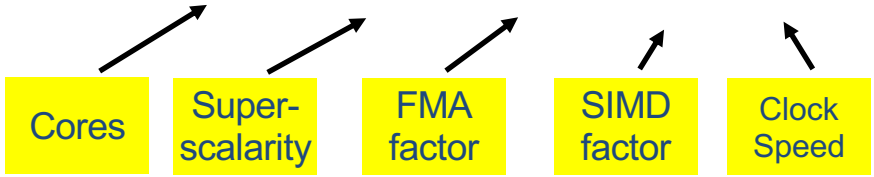


GPU

	2x Intel Xeon Platinum 8360Y	Fujitsu A64FX	NVidia A100 “Ampere”
Cores@Clock	2 x 36 @ ≥1.8 GHz	48 @ 2.2 GHz	108 SMs @ ~1.4 GHz
DP peak	4.2,…,6 TFlop/s	3.4 TFlop/s	19.5 TFlop/s
Stream BW (meas.)	2 x 170 GB/s	860 GB/s (HBM)	1400 GB/s
Transistors / TDP	~? Billion / 2x250 W	8 Billion / ~200W	54 Billion/400W
Threads to saturate bandwidth	~30	~20	~80.000

There is no single driving force for single core performance!

$$P_{chip} = n_{core} \cdot n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f$$



	n_{core}	n_{super}^{FP} inst./cy	n_{FMA}	n_{SIMD} ops/inst		Chip	f [GHz]	P_{chip} [TF/s]
Icelake	36	2	2	8	Q2/2021	Platinum 8360Y	1.8 / 2.6	2.1/3.0
AMD Rome	64	2	2	4	Q4/2019	EPYC 7742	2.25	2.3
Nvidia V100	80	1	2	32	Q2/2017	V100-PCIe-16GB	1.38	7.1
Nvidia A100	108	1	2	32	Q2/2020	A100-SXM4-40GB	1.41	9.7
AMD MI100	120	1	2	32	Q4/2020		1.51	11.5
AMD MI250X	2 x 110	1	2	64	Q2/2022		1.7	47.8

Nvidia A100 “Ampere” SXM4 40GB specs

Architecture

- 54.2 B Transistors
- ~ 1.4 GHz clock speed
- ~ 108 “SM” units (cores)
  - 64 SP / 32 DP FMA units each (SIMT)
  - 2:1 SP:DP performance
  - 4 “Tensor Cores” each

- 9.7 TFlop/s DP peak (FP64)
- 19.5 TFlop/s DP peak (Tensor)
- 40 MiB L2 Cache

- 40 GB (5120-bit) HBM2
- MemBW ~ 1555 GB/s (theoretical)
- MemBW ~ 1400 GB/s (measured)



$$P_{chip} = n_{core} \cdot n_{super}^{FP} \cdot n_{FMA} \cdot n_{SIMD} \cdot f \cdot n_{tensor}$$

# SMs	1 inst./cy	FMA factor	#Flop/inst.

$$n_{core} = 108$$

$$n_{SIMD} = 32 \frac{Flop}{inst}$$

$$n_{tensor} = 8 \cdot 8 = 64 \frac{Flop}{inst}$$

$$n_{super} = 1 \frac{inst}{cy}$$

$$n_{FMA} = 2$$

$$f = 1.4 \frac{Gcy}{s}$$

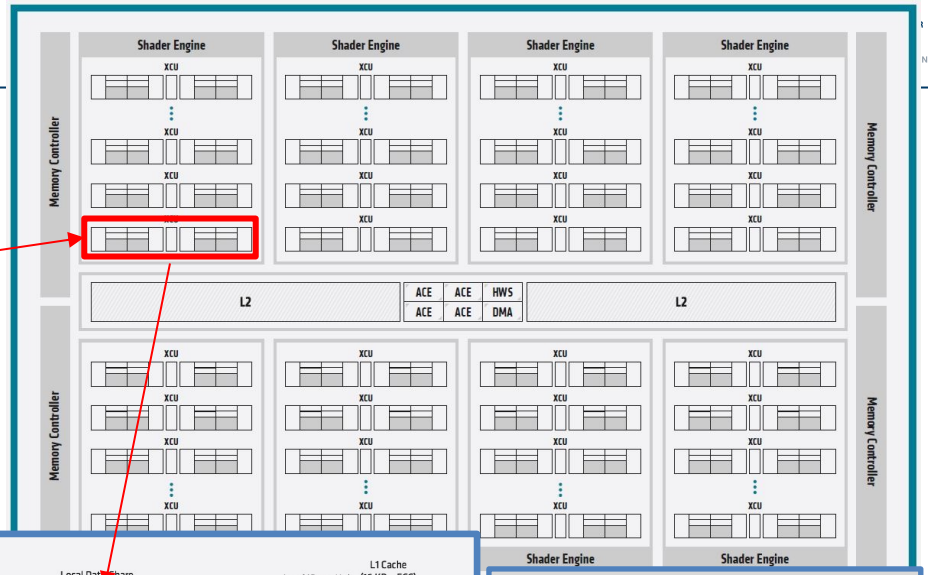
AMD MI250x specs

Architecture

- 2x 58 B Transistors
- ~ 1.7 GHz clock speed
- ~ 2 x 110 “CU” units (cores)
  - Multi chip module
  - 64 SP / 64 DP FMA units each (SIMT)
  - 1:1 SP:DP performance

- 47.9 TFlop/s DP peak (FP64)
- 95.8 TFlop/s DP peak (Tensor)
- 2 x 8 MiB L2 Cache

- 2x 64 GB (8192-bit) HBM2
- 3277 GB/s (theoretical)



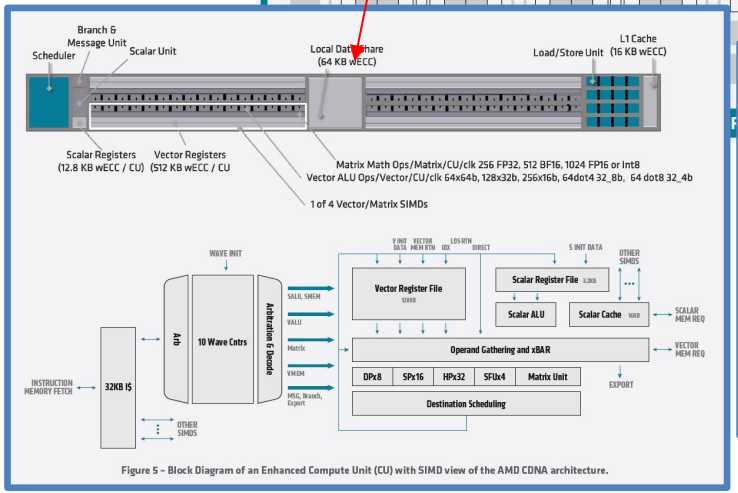


Figure 5 – Block Diagram of an Enhanced Compute Unit (CU) with SIMD view of the AMD CDNA architecture.

$$n_{core} = 2x110$$

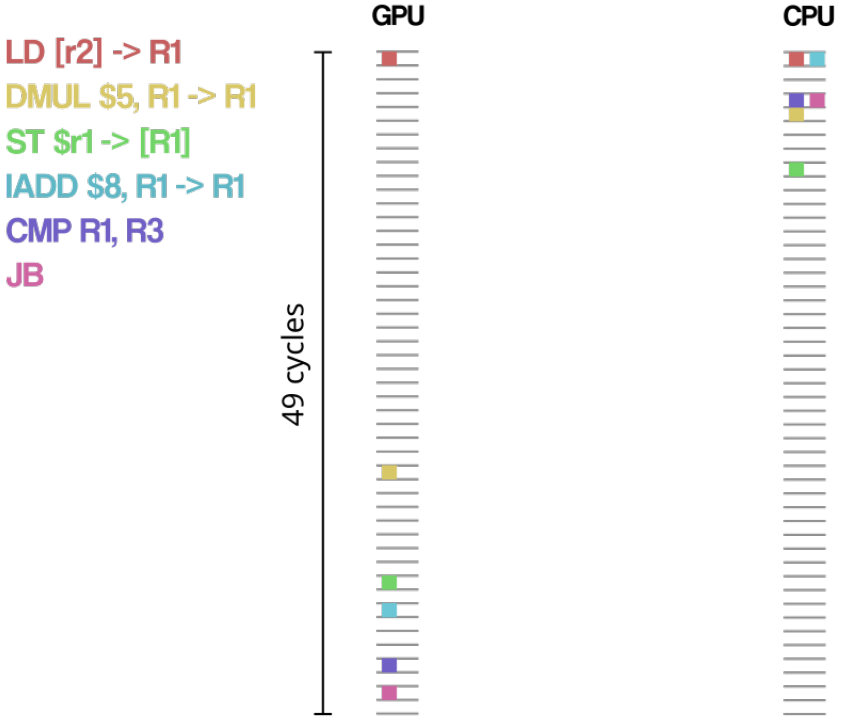
$$n_{SIMD} = 64\frac{Flop}{inst}$$

$$n_{super} = 1\frac{inst}{cy}$$

$$n_{FMA} = 2$$

$$f = 1.7\frac{\mathrm{Gcy}}{\mathrm{s}}$$

CPU vs GPU – Single Threaded

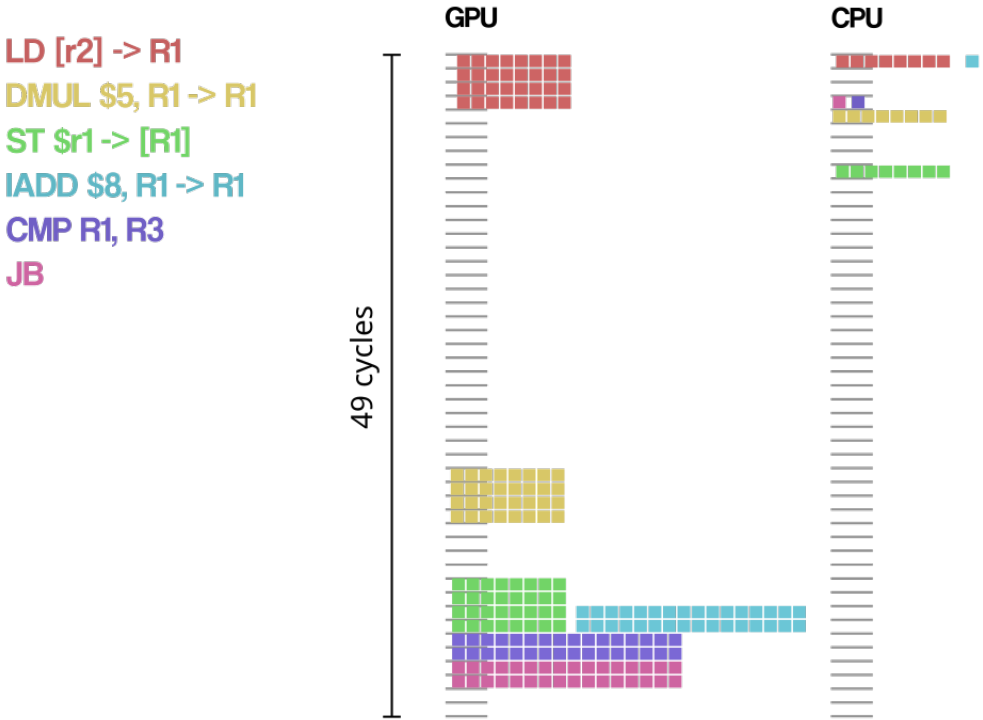


GPU has low **single thread throughput**:

- high cache latency
- higher instruction latency
- no Out-of-Order execution
- limited to single/dual issue

(The numbers in the illustration on the left side is representative for a V100 / A100 GPU and any modern Intel CPU)

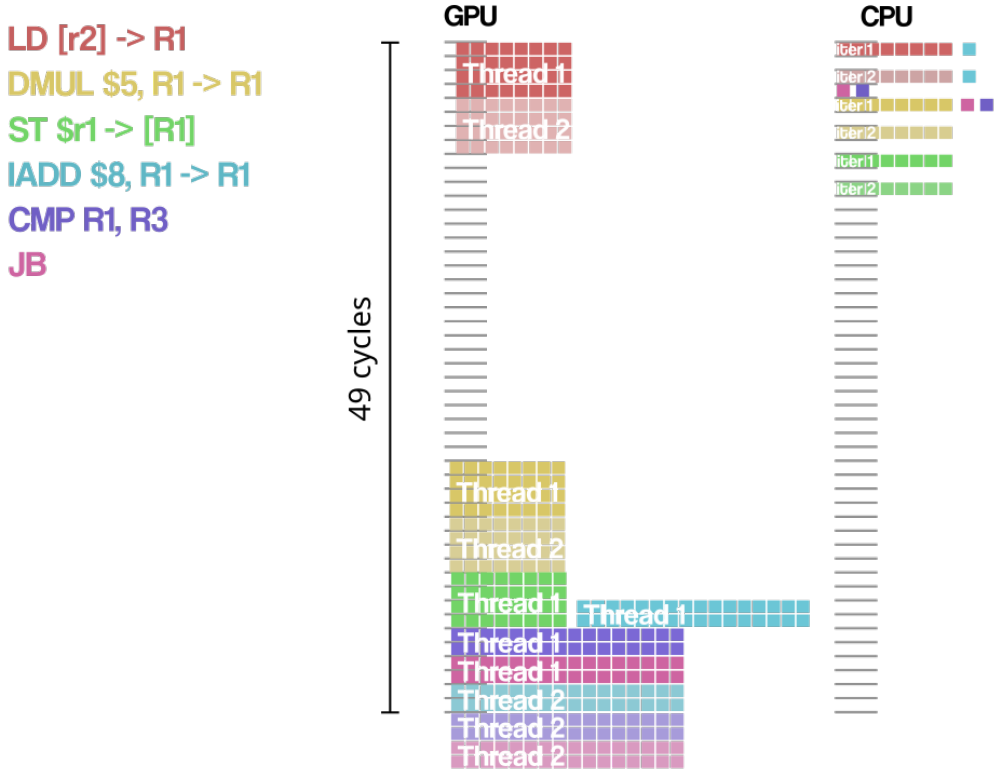
CPU vs GPU – Vectorization



Vector instructions

- CPU: 8 wide SIMD (AVX512/DP)
- GPU: 32 wide SIMD registers on 8/16 wide execution units
- On the GPU, every instruction is a vector instruction, vectorization by default!
- Better hardware support makes vectorization more flexible than on CPUs

CPU vs GPU – ILP vs TLP



Filling in the gaps:

- The CPU uses speculative execution and its Out-of-Order engine to fill gaps with instructions from following iterations

-> Instruction Level Parallelism (ILP)

- The GPU uses massive amounts of SMT (up to 16x) to fill gaps with instructions from other threads

-> Thread Level Parallelism (TLP)

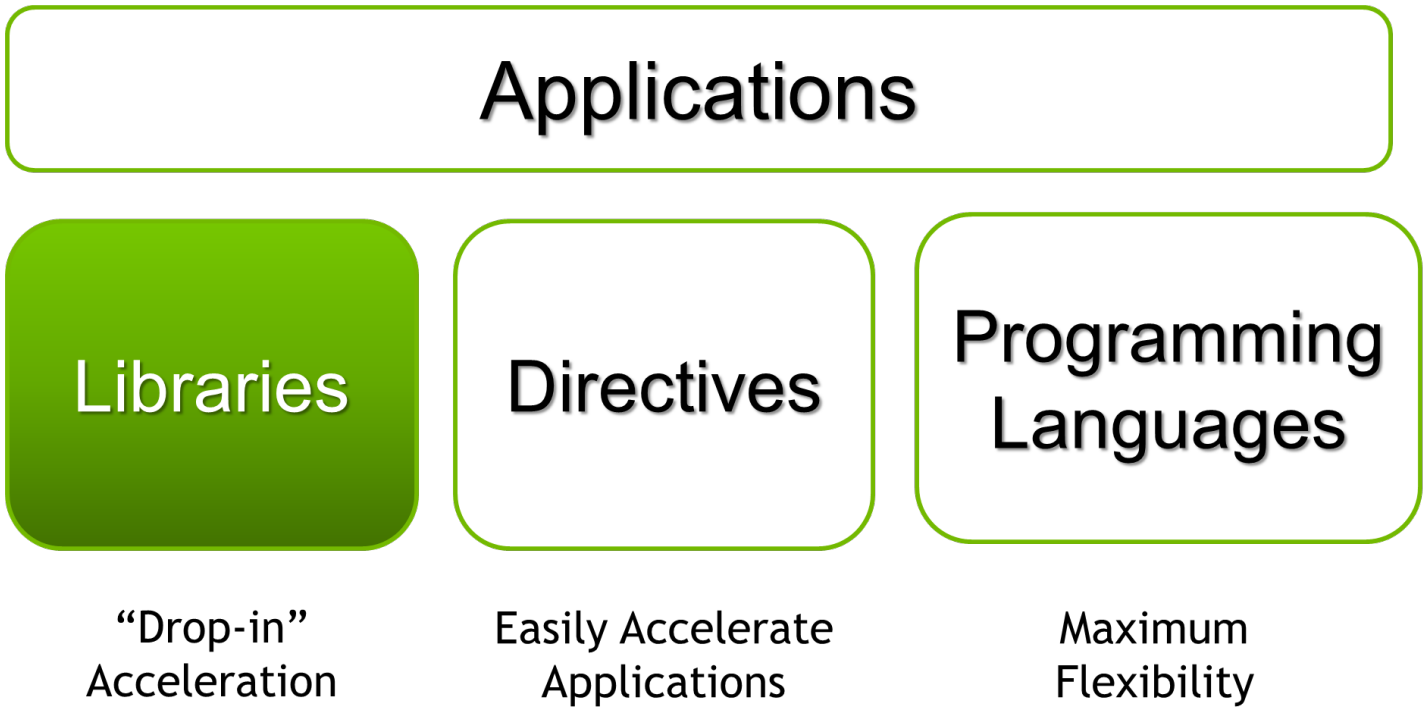
FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Basic Concepts of GPU Programming

CUDA and OpenCL



3 Ways to Accelerate Applications on GPU



extracted from Nvidia slides found at developer.download.nvidia.com

Data Parallel Programming Models: CUDA, OpenCL, HIP, SyCL

- Program it like a pixel shader: write a small shader program (compute kernel) that computes just the color (result) for a single pixel (work item)
- Evaluate program for a whole image (iteration space)
- General program still runs on the CPU, dispatches or offloads computational kernels to the GPU

```
double A[1000];
double B[1000];

[...]

// copy A from host to device
call_kernel("scale", 1000, A, B);
// copy B from device to host
```

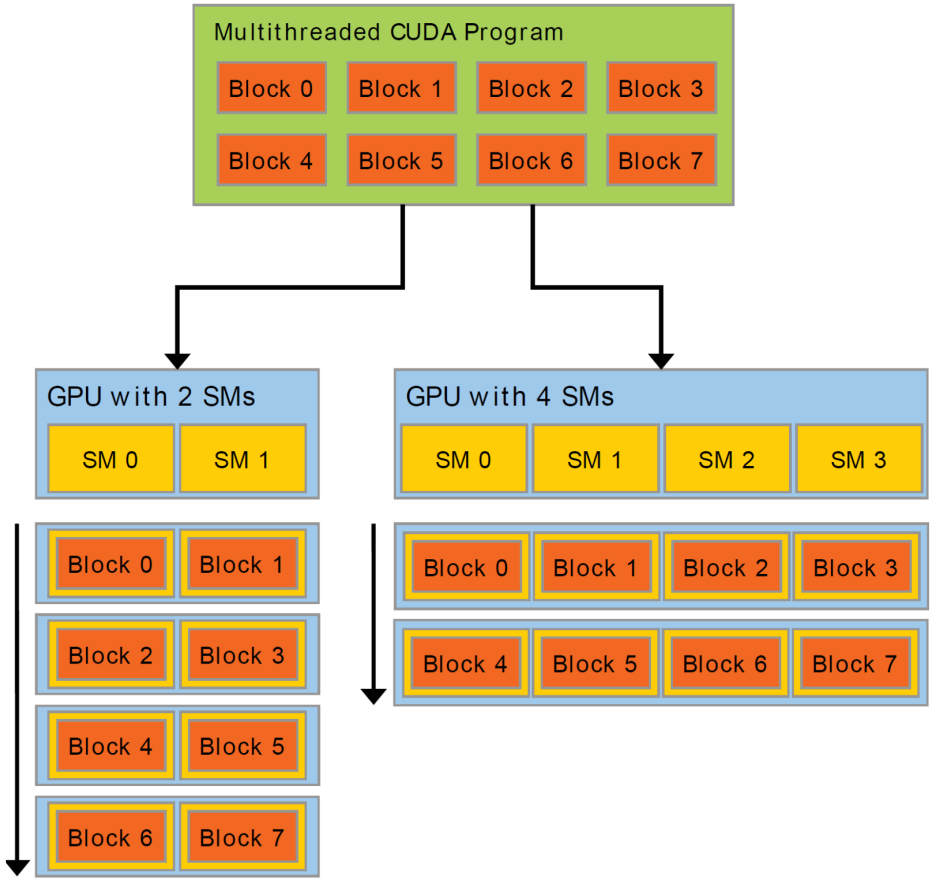
```
scale( double* in,
       double* out,
       int threadID ) {
    out[threadID] = in[threadID] * 2.0;
}
```

GPU programming APIs

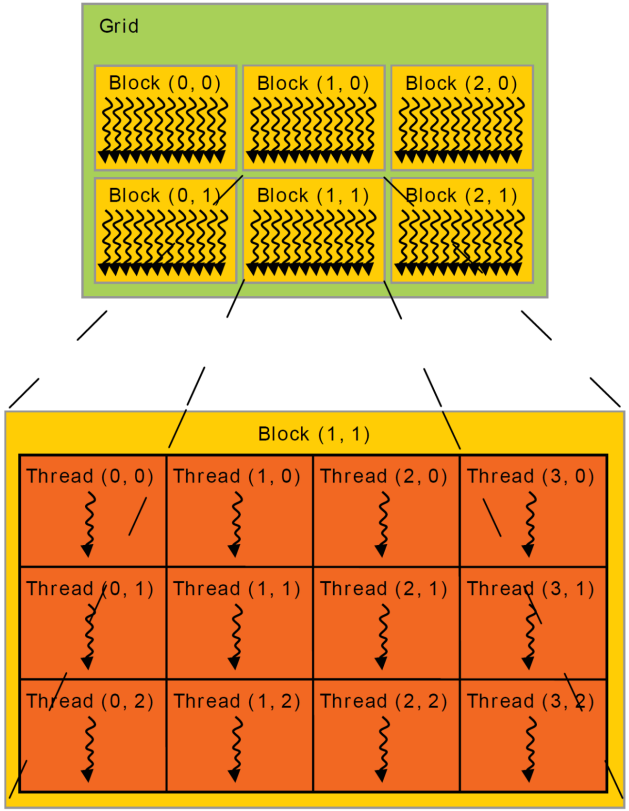
- **CUDA** (Compute Unified Device Architecture)
  - Proprietary by NVIDIA, works only on NVIDIA products
  - Largest Tool / Library / Software ecosystem
  - Always supports the latest hardware features, best choice for NVIDIA hardware
- **OpenCL** (Open Computing Language)
  - Managed by non-profit technology consortium Khronos group: www.khronos.org/opencl
  - Supported by many vendors, including support for FPGA / DSP
  - „runs everywhere“, but platform specific optimization is necessary
  - Evolution of OpenCL is glacial compared to CUDA, many implementations have partial and spotty support
- **HIP** (Heterogeneous Interface for Portability)
  - Straight up CUDA clone by AMD
  - HIP and CUDA programs are very transferable between each other (perl CUDA->HIP porting skript)
- **SYCL**
  - Managed by Khronos Group
  - C++ oriented, standard algorithm like language
  - Mostly championed by Intel, 3rd party support for all GPU platforms

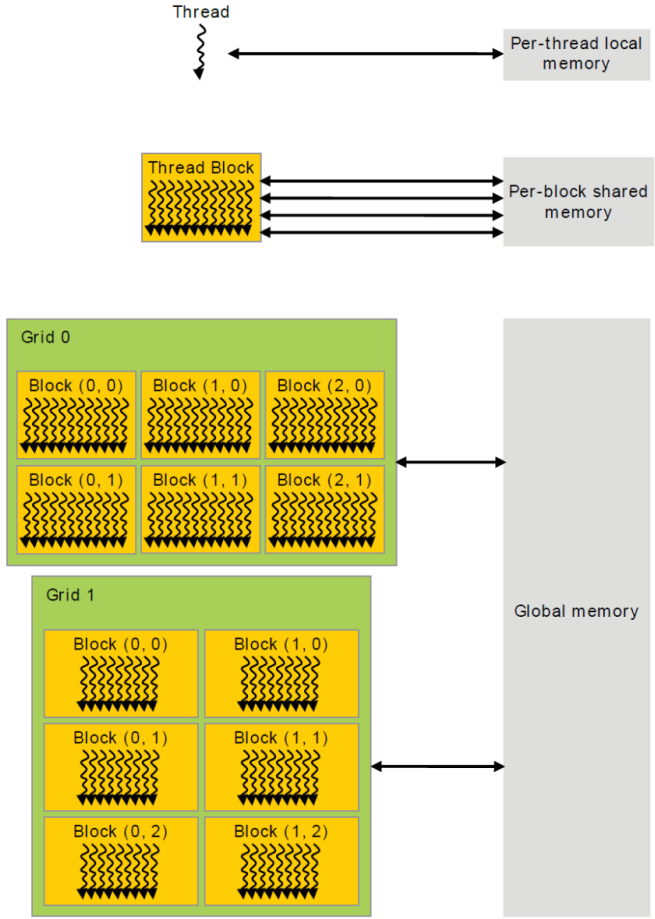
Automatic Scalability

GPU Features	NVIDIA Tesla P100	NVIDIA Tesla V100	NVIDIA A100
GPU Codename	GP100	GV100	GA100
GPU Architecture	NVIDIA Pascal	NVIDIA Volta	NVIDIA Ampere
Compute Capability	6.0	7.0	8.0
Threads / Warp	32	32	32
Max Warps / SM	64	64	64
Max Threads / SM	2048	2048	2048
Max Thread Blocks / SM	32	32	32
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Block	65536	65536	65536
Max Registers / Thread	255	255	255
Max Thread Block Size	1024	1024	1024
FP32 Cores / SM	64	64	64
Ratio of SM Registers to FP32 Cores	1024	1024	1024
Shared Memory Size / SM	64 KB	Configurable up to 96 KB	Configurable up to 164 KB



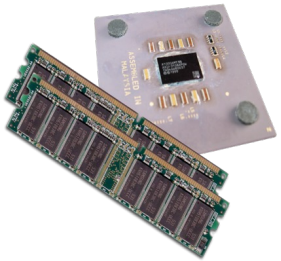
Grid of Thread Blocks





Heterogeneous Computing

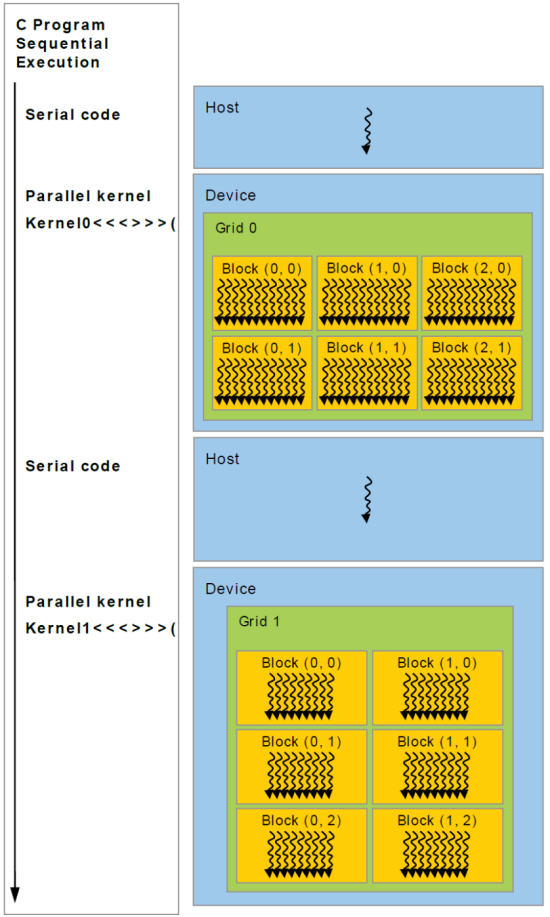
- Terminology:
  - *Host* The CPU and its memory (host memory)
  - *Device* The GPU and its memory (device memory)



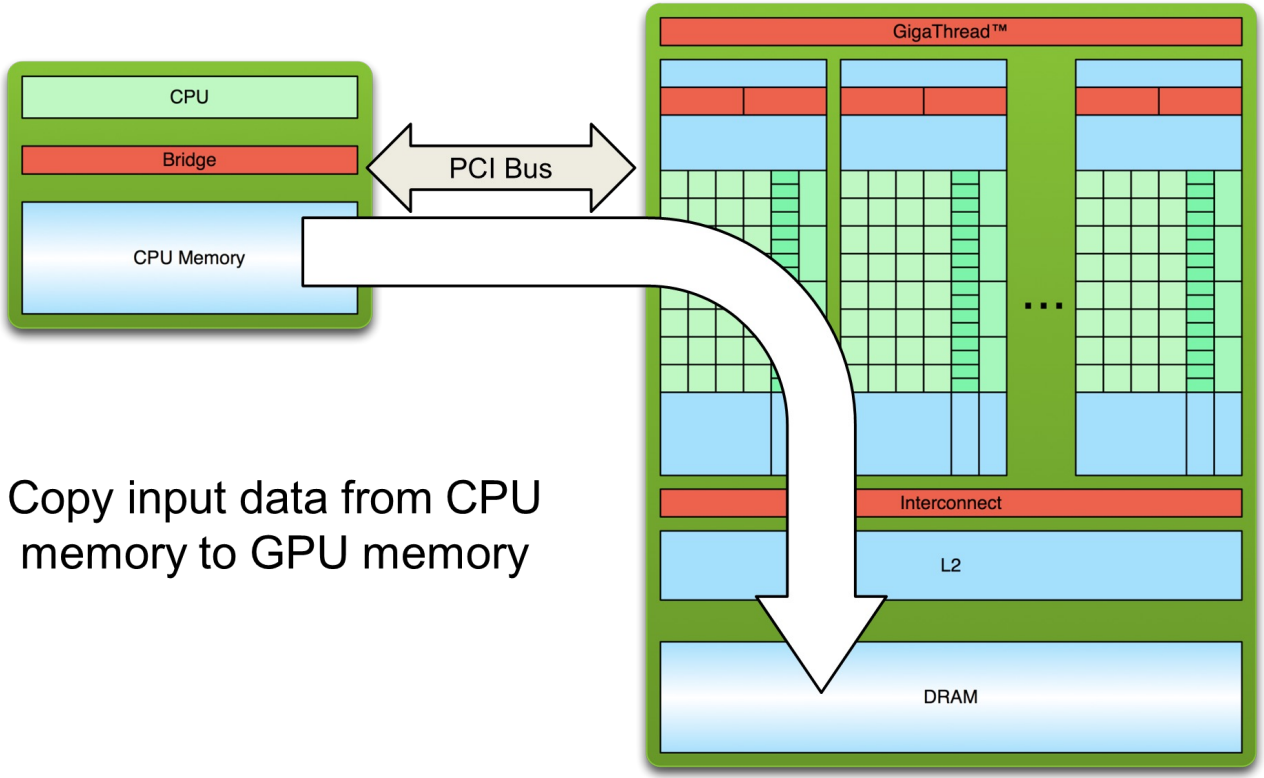
Host



Device

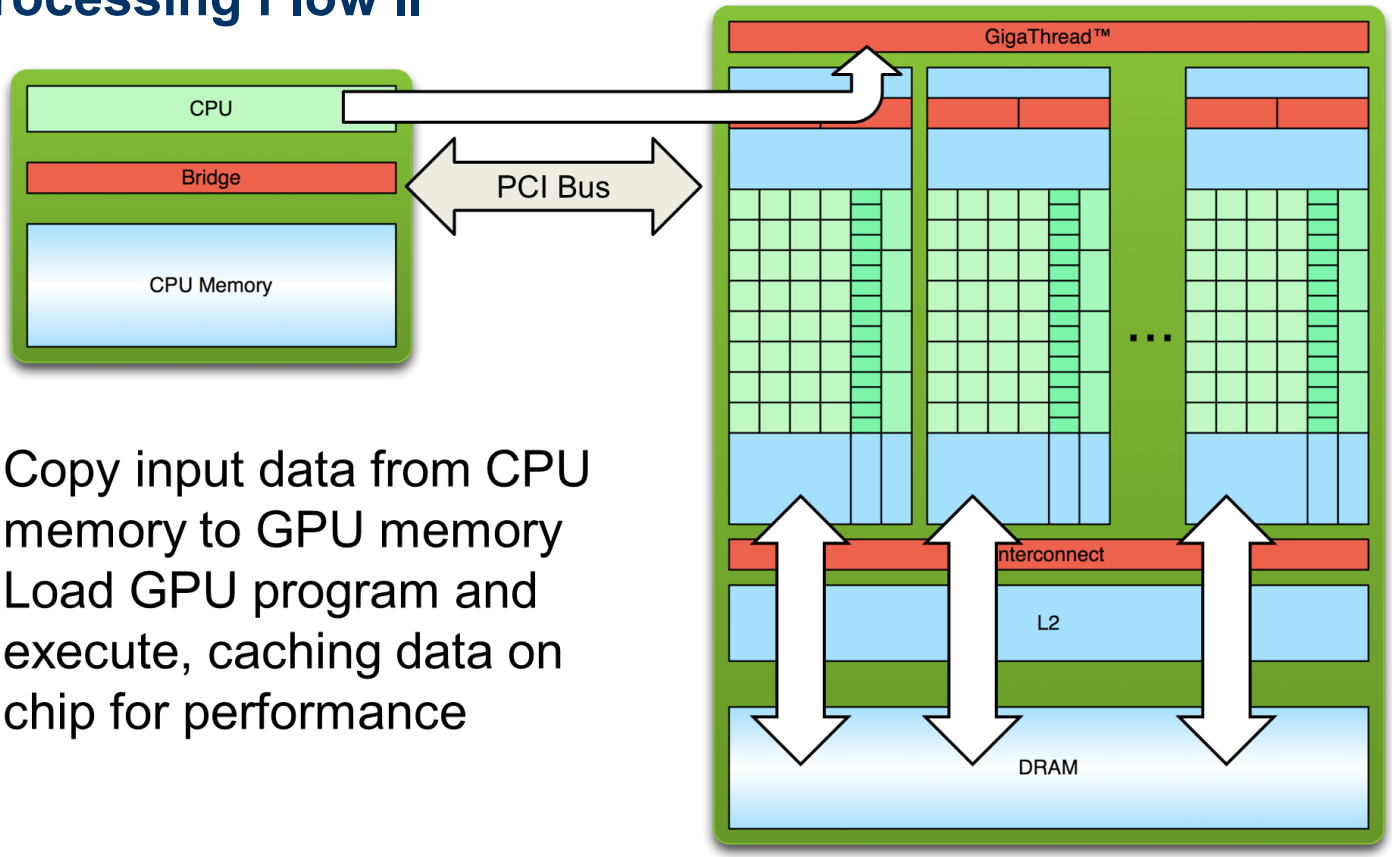


Simple Processing Flow I



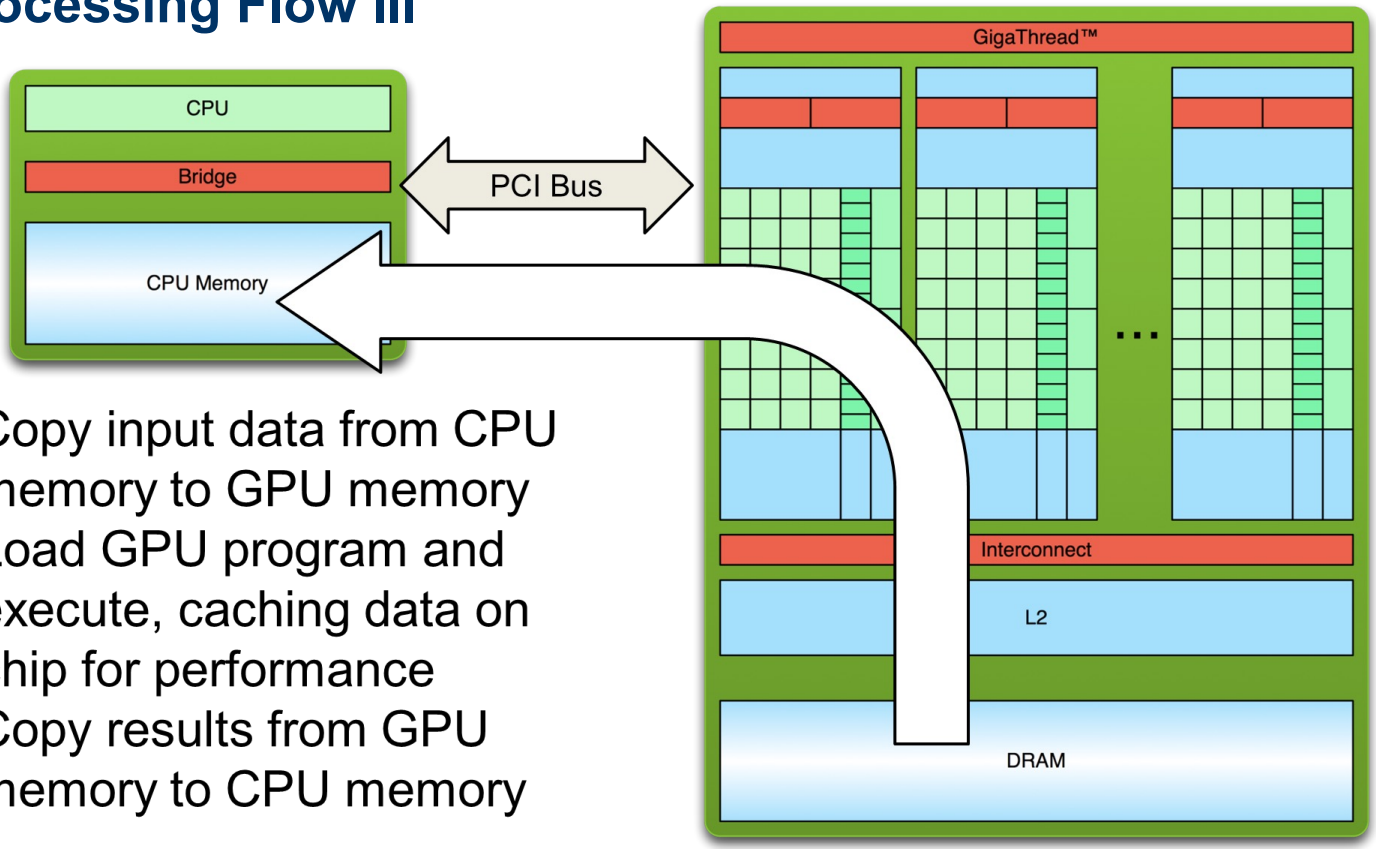
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow II



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow III



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

CUDA Hello World!

```
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

* Standard C that runs on the host
* NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:
$ nvcc hello_world.cu
$ a.out
Hello World!

CUDA Hello World! with Device Code

```
__global__ void mykernel(void) {
}

int main(void) {
    mykernel<<<1,1>>>(); // grid dim, block dim
    printf("Hello World!\n");
    return 0;
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

Vector addition on GPU

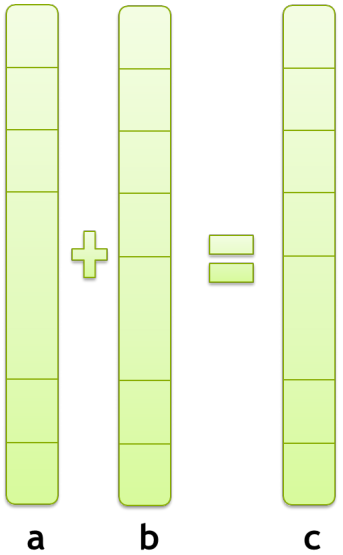


Parallel Programming in CUDA C/C++

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {
  *c = *a + *b;
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory
- We need to allocate memory on the GPU



Addition on the Device: main()

```
int main(void) {
        int a, b, c;              // host copies of a, b, c
        int *d_a, *d_b, *d_c;          // device copies of a, b, c
        int size = sizeof(int);

        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Setup input values
        a = 2;
        b = 7;
```

Addition on the Device: main()

```
        // Copy inputs to device
        cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

        // Launch add() kernel on GPU
        add<<<1,1>>>(d_a, d_b, d_c);

        // Copy result back to host
        cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

        // Cleanup
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

Moving to parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();
```



```
add<<< N, 1 >>>();
```

- Instead of executing `add()` once, execute N times in parallel

Vector Addition on the Device

- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using `blockIdx.x`

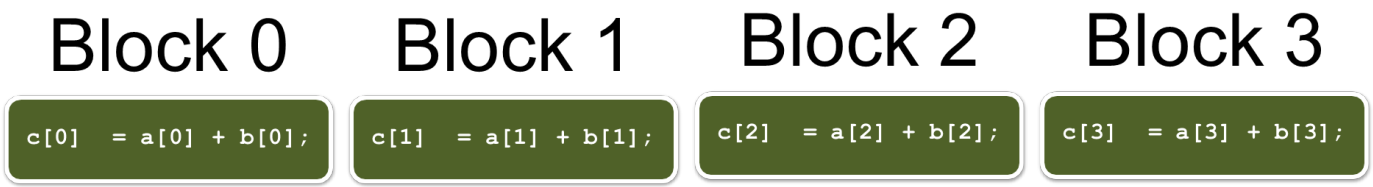
```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index

Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- On the device, each block can execute in parallel:



Vector Addition on the Device: add()

- Returning to our parallelized `add()` kernel

```
__global void add(int *a, int *b, int *c) {
      c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- Let's take a look at main()…

Vector Addition on the Device: main()

```
#define N 512
int main(void) {
    int *a  *b  *c          // host copies of a, b, c
    int *d_a, *d_b, *d_c;  // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition on the Device: main()

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

CUDA threads

- Terminology: a block can be split into parallel threads

- Let's change `add()` to use parallel *threads* instead of parallel *blocks*

```
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- We use `threadIdx.x` instead of `blockIdx.x`

- Need to make one change in `main()`…

Vector Addition Using Threads: main()

```
#define N 512
int main(void) {
    int *a, *b, *c;                 // host copies of a, b, c
    int *d_a, *d_b, *d_c;           // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition Using Threads: main()

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

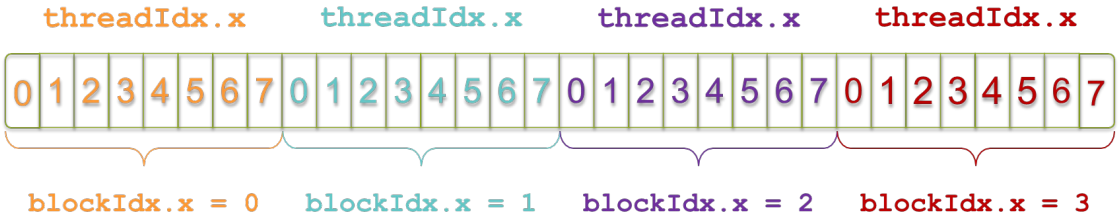
    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Combining Blocks and Threads

- We've seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- Let's adapt vector addition to use both blocks and threads
- Why? We'll come to that…
- First let's discuss data indexing…

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

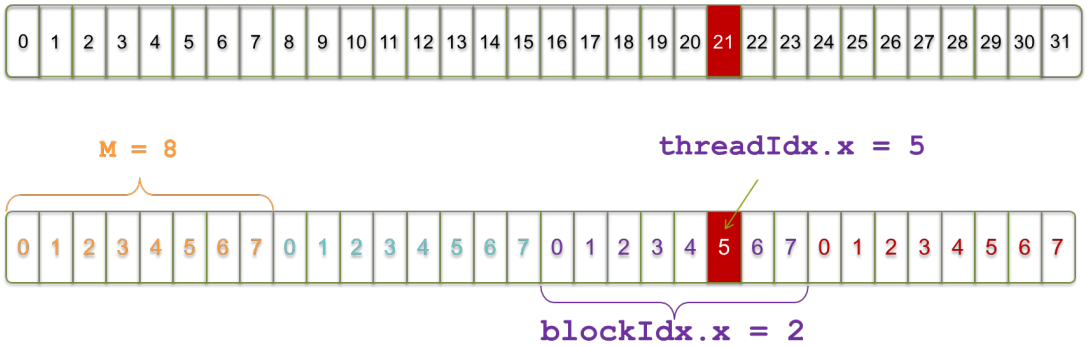


  - With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

Indexing Arrays: Example

- Which thread will operate on the red element?




```
int index = threadIdx.x + blockIdx.x * M;
          =      5      +     2      * 8;
          = 21;
```

Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- Combined version of `add()` to use parallel threads *and* parallel blocks

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

- What changes need to be made in `main()`?

Addition with Blocks and Threads: main()

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;               // host copies of a, b, c
    int *d_a, *d_b, *d_c;         // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Addition with Blocks and Threads: main()

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`

- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
                        if (index < n)
                    c[index] = a[index] + b[index];
                                }
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- (Unlike parallel blocks) threads have fast mechanisms to:
  - Communicate
  - Synchronize
- To look closer, we need a new example…

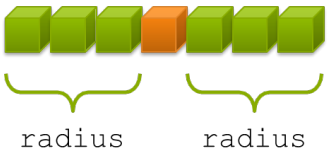
1D Stencil on GPU



1D Stencil

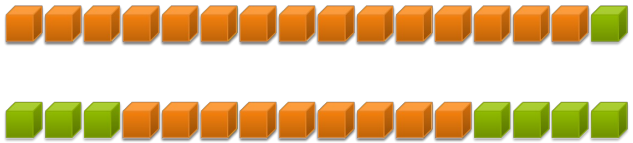
- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

- If radius is 3, then each output element is the sum of 7 input elements:



Implementing Within a Block

- Each thread processes one output element
  - blockDim.x elements per block
- Input elements are read several times
  - With radius 3, each input element is read seven times

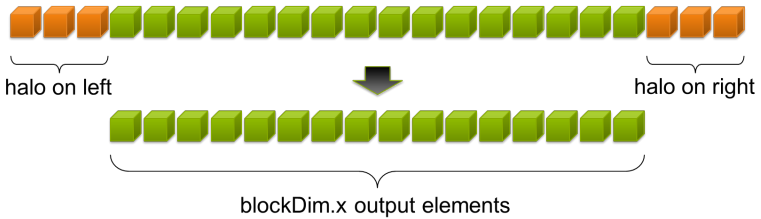


Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using `__shared__`, allocated per block
- Data is not visible to threads in other blocks

Implementing With Shared Memory

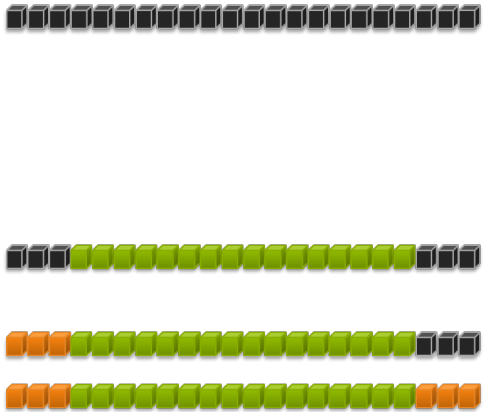
- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory
  - Each block needs a halo of radius elements at each boundary



Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
  __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
  int gindex = threadIdx.x + blockIdx.x * blockDim.x;
  int lindex = threadIdx.x + RADIUS;

  // Read input elements into shared memory
  temp[lindex] = in[gindex];
  if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] =
      in[gindex + BLOCK_SIZE];
  }
```



Stencil Kernel

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

Data race!

- The stencil example will not work…
- Suppose thread 15 reads the halo before thread 0 has fetched it…

Store at temp[18]



```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex - RADIUS = in[gindex - RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];
```

Skipped, threadIdx > RADIUS

Load from temp[19]



__syncthreads()

- `void __syncthreads();`
- Synchronizes all threads within a block
  - Used to prevent RAW / WAR / WAW hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + radius;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();
```

Stencil Kernel

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

Next time…

* Performance Model and Analysis for Stencils in 2D and 3D on GPU