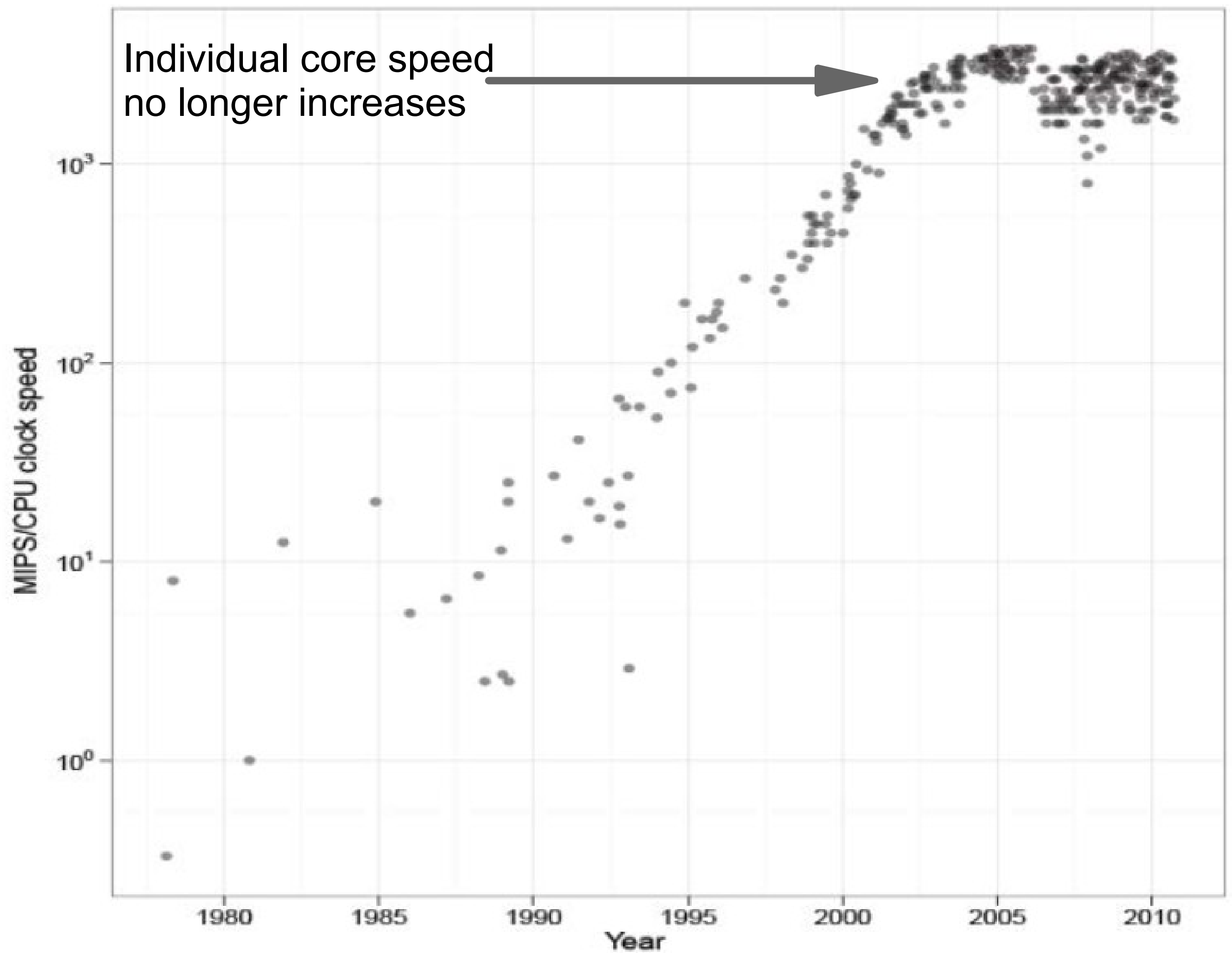
GPU Programming with CUDA

Piotr Luszczek

GPU Hardware Trends and Features

Why GPUs? Look at Per-Core CPU Performance



Decline of CPUs and Ascendents of GPUs

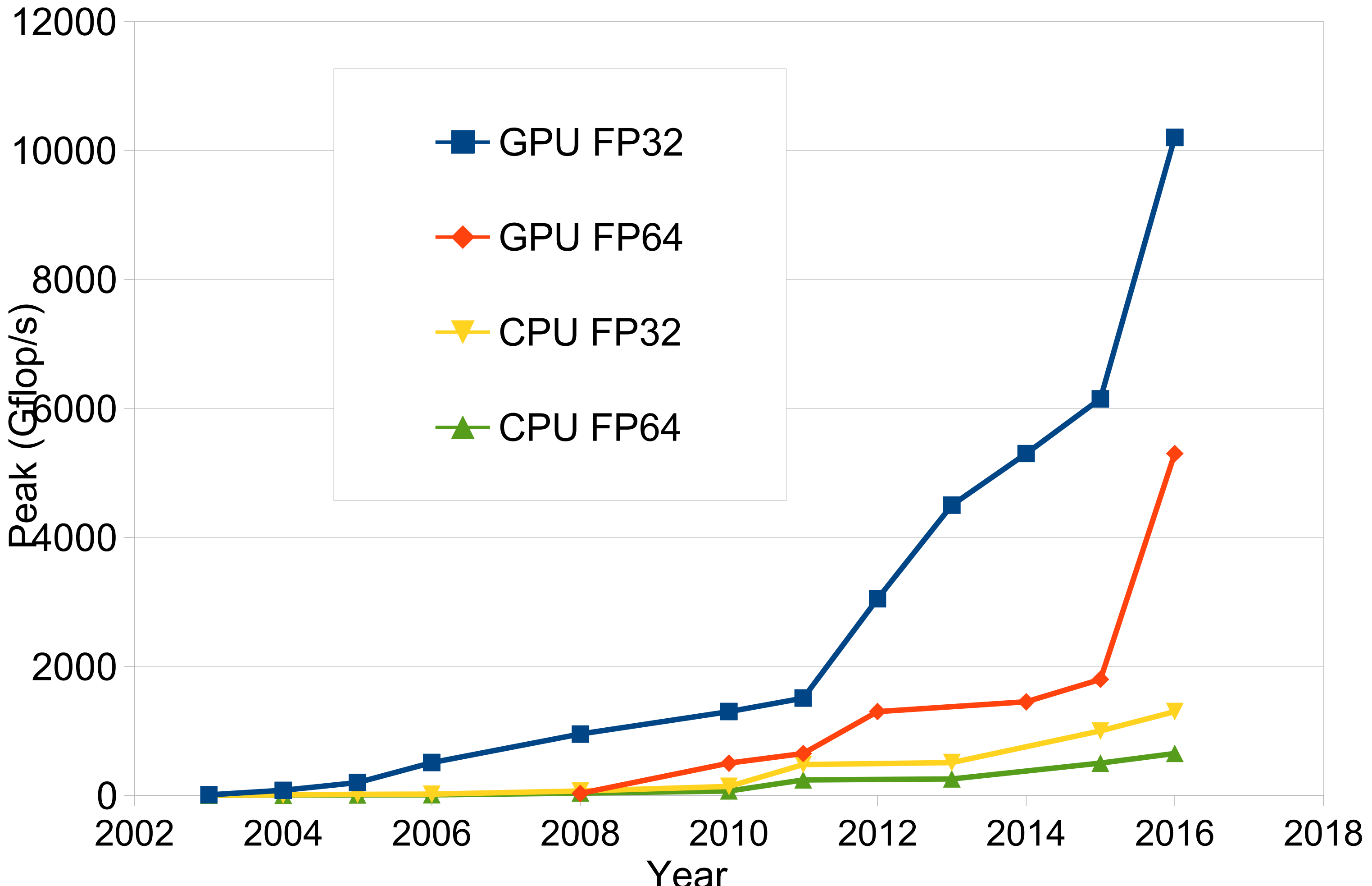
- CPUs were under assault since 2006 – the multicore year
- Some examples of declines
  - Moore's Law
    - Intel switched from tick-tock cycles to three-way process, architecture, optimization
  - Dennard Scaling
    - Shrinking scale, lowering voltage, and increasing frequency is no longer possible
  - Power wall
    - Modern processors are heavily throttled due to thermal issues
  - Frequency wall
    - Power consumption increases too fast with frequency
  - Additive core-count increases

- High Bandwidth Memory
  - Practical even for low-end gaming cards
  - Version 1 achieved 0.5 TB/s
  - Version 2 goes to 1 TB/s
- NVLink
  - Subsumes the PCI Express bottleneck
  - Allows practical shared-memory experience across GPUs
- New applications
  - AI
  - Self-driving
  - DNNs and CNNs
  - Big Data and ML

What Does it Take To Improve a CPU?

- Increase pipeline length
  - Instruction-Level Parallelism of serial code is fixed
- Improve data prefetcher
  - Even for perfect prediction, memory bandwidth is a fixed limit
- Improve out-of-order execution
  - Complexity usually grows quadratically with the number of outstanding instructions
- Improve branch predictor
  - Even straight line code (no branches) will be limited by frequency
- The best strategies are
  - Increase parallelism
  - Specialize hardware for common tasks

GPU vs. CPU Performance over Years



Similar trend holds for the main memory bandwidth.

GPU and GPGPU: The Origin Story

- Programmable graphics pipeline
  - GLSL (shader language) for custom graphics effects in games
- Interpolation vs. dynamic range
  - Colors in graphics look better in floating-point in dynamic range
- Early attempts at programming
  - Cg, Brook, …
- Modern standards or de facto standards
  - CUDA (currently 8, 9.1.85)
    - Compute Unified Device Architecture
  - OpenCL (currently 2)
  - Vulkan and Metal
- High-level languages
  - OpenMP 4.5 with offload directives
  - OpenACC (like OpenMP but GPU-oriented)

Comparison of Early NVIDIA GPUs

Tesla Cards	G80	GT200	GF100
GPU		Tesla	Fermi
Transistors (billions)	.681	1.4	3.0
CUDA Cores	128	240	512
Double precision FP	None	30 FMA ops / clock	256 FMA ops / clock
Single precision FP	128 MAD ops / clock	240 ops / clock	512 FMA ops / clock
Special Function Units / SM	2	2	4
Warp schedulers / SM	1	1	2
Shared memory / SM (KiB)	16	16	16 or 48
L1 Cache / SM (KiB)	None	None	16 or 48
L2 Cache (KiB)	None	None	768
ECC Memory Support	No	No	Yes
Concurrent kernels	No	No	1..16
Load/Store Address Bits	32	32	64

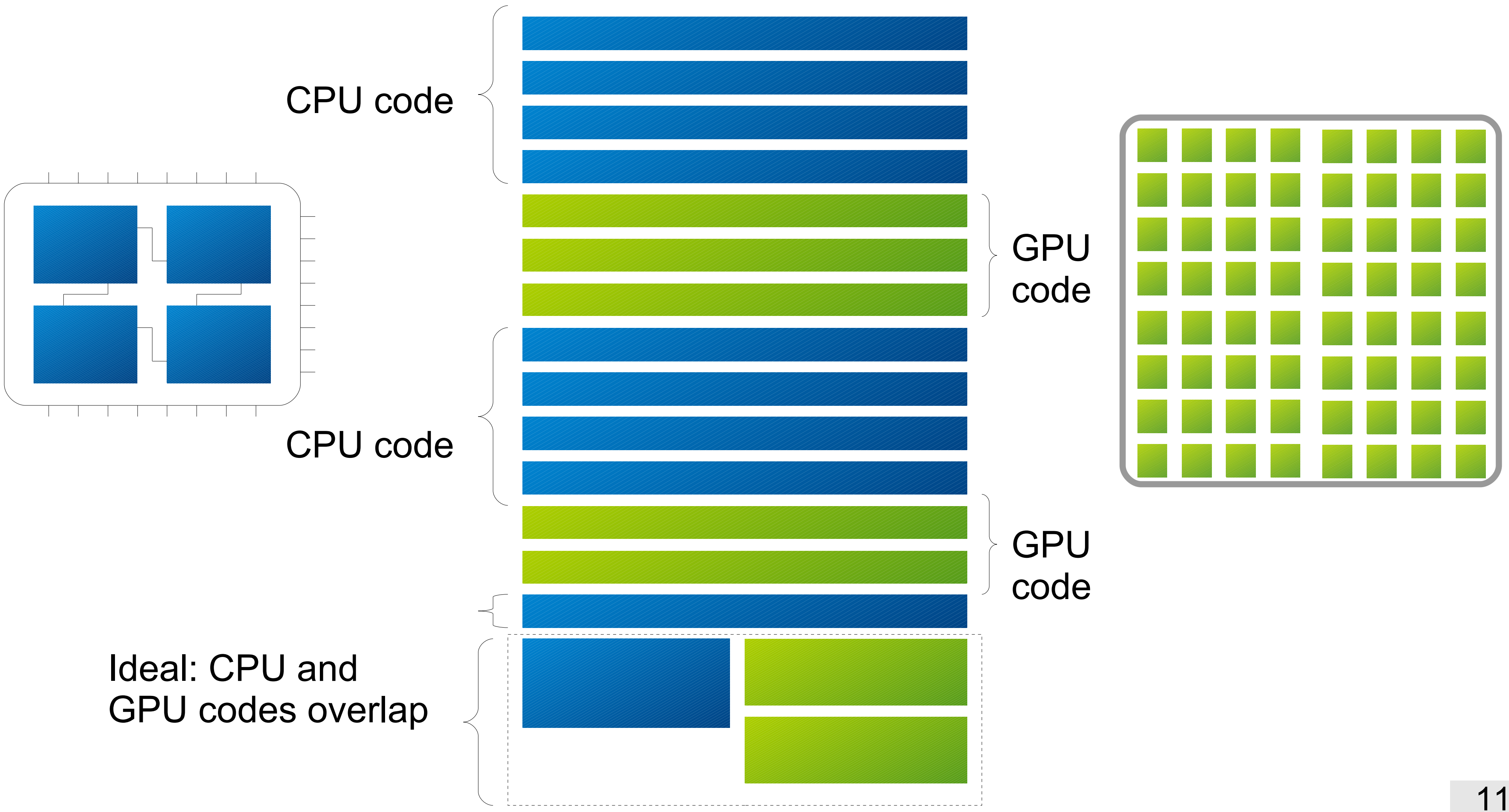
Comparison of Recent NVIDIA GPUs

Tesla cards	M40	K40	K80	P100	V100
GPU	GM200	GK110	2x GK210	GP100	GV100
Name	Maxwell	Kepler	Kepler	Pascal	Volta
SMs	24	15	13+13	56 (28 TPCs)	80 (40 TPCs)
CUDA cores	3072 (96 FP64 cores)	2880 (960 FP64 cores)	2x 2496	3584 (1792 FP64)	5120 (640 tensor cores)
Base Clock MHz	948	745	560	1328	
Boost Clock MHz	1114	810/875	875	1480	1455
Peak FP64 Tflop/s	0.213	1.680	2.91	5.304	7.5
Peak FP32 Tflop/s	7	5.04	8.73	10.6	15 (120 Tensor Core)
Texture Units	192	240	208+208	224	320
Register File KB	6144	3840	3328+3328	14336	20480
L2 Cache Size KB	3072	1536	3072	4096	6144
Memory type	GDDR5	GDDR5	GDDR5	HBM2	HBM2
Memory interface bits	384	384	384	4096	4096
Memory size GB	24	12	24 (12+12)	16	16 GB
Peak bandwidth GB/s	288	288	480 (240+240)	720	900
TDP W	250	235	300	300	300
Transistors billions	8	7.1	7.1+7.1	15.3	21.2
Die size mm^2	601	551	561+561	610	815
Manufacturing process	28 nm	28 nm	28 nm	16 nm	12 nm

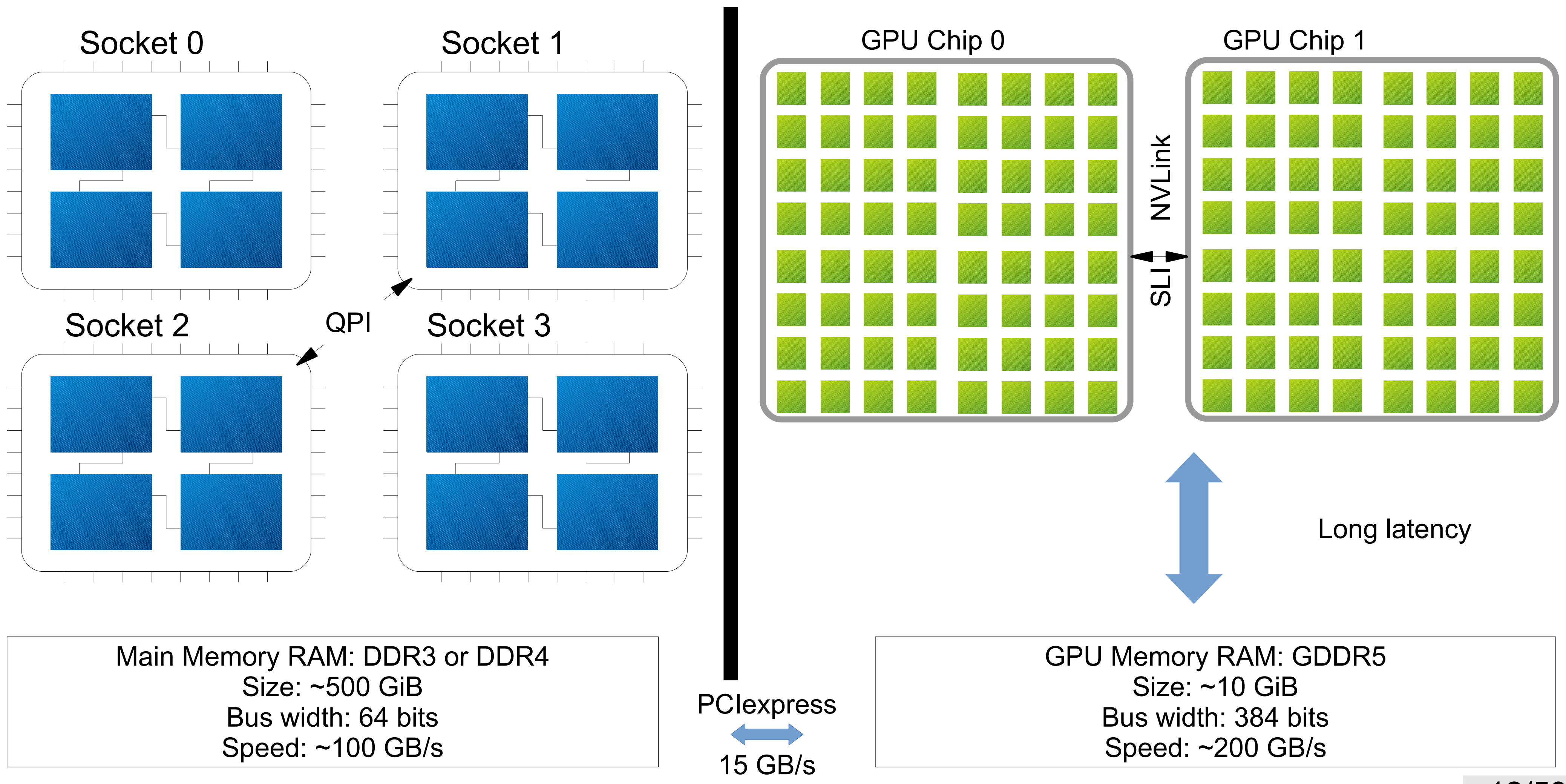
CUDA: Proprietary but Established

- Nine major releases
  - CUDA Toolkit 10.0 (Sept 2018)
  - CUDA Toolkit 9.2 (May 2018)
  - CUDA Toolkit 9.1 (Dec 2017)
  - CUDA Toolkit 9.0 (Sept 2017)
  - CUDA Toolkit 8.0 GA2 (Feb 2017)
  - CUDA Toolkit 8.0 GA1 (Sept 2016)
  - CUDA Toolkit 7.5 (Sept 2015)
  - CUDA Toolkit 7.0 (March 2015)
  - CUDA Toolkit 6.5 (August 2014)
  - CUDA Toolkit 6.0 (April 2014)
  - CUDA Toolkit 5.5 (July 2013)
  - CUDA Toolkit 5.0 (Oct 2012)
  - CUDA Toolkit 4.2 (April 2012)
- Older releases still available
  - CUDA Toolkit 4.1 (Jan 2012)
  - CUDA Toolkit 4.0 (May 2011)
  - CUDA Toolkit 3.2 (Nov 2010)
  - CUDA Toolkit 3.1 (June 2010)
  - CUDA Toolkit 3.0 (March 2010)
  - OpenCL 1.0 Release (Sept 2009)
  - CUDA Toolkit 2.3 (June 2009)
  - CUDA Toolkit 2.2 (May 2009)
  - CUDA Toolkit 2.1 (Jan 2009)
  - CUDA Toolkit 2.0 (Aug 2008)
  - CUDA Toolkit 1.1 (Dec 2007)
  - CUDA Toolkit 1.0 (June 2007)

Software: CPU + GPU



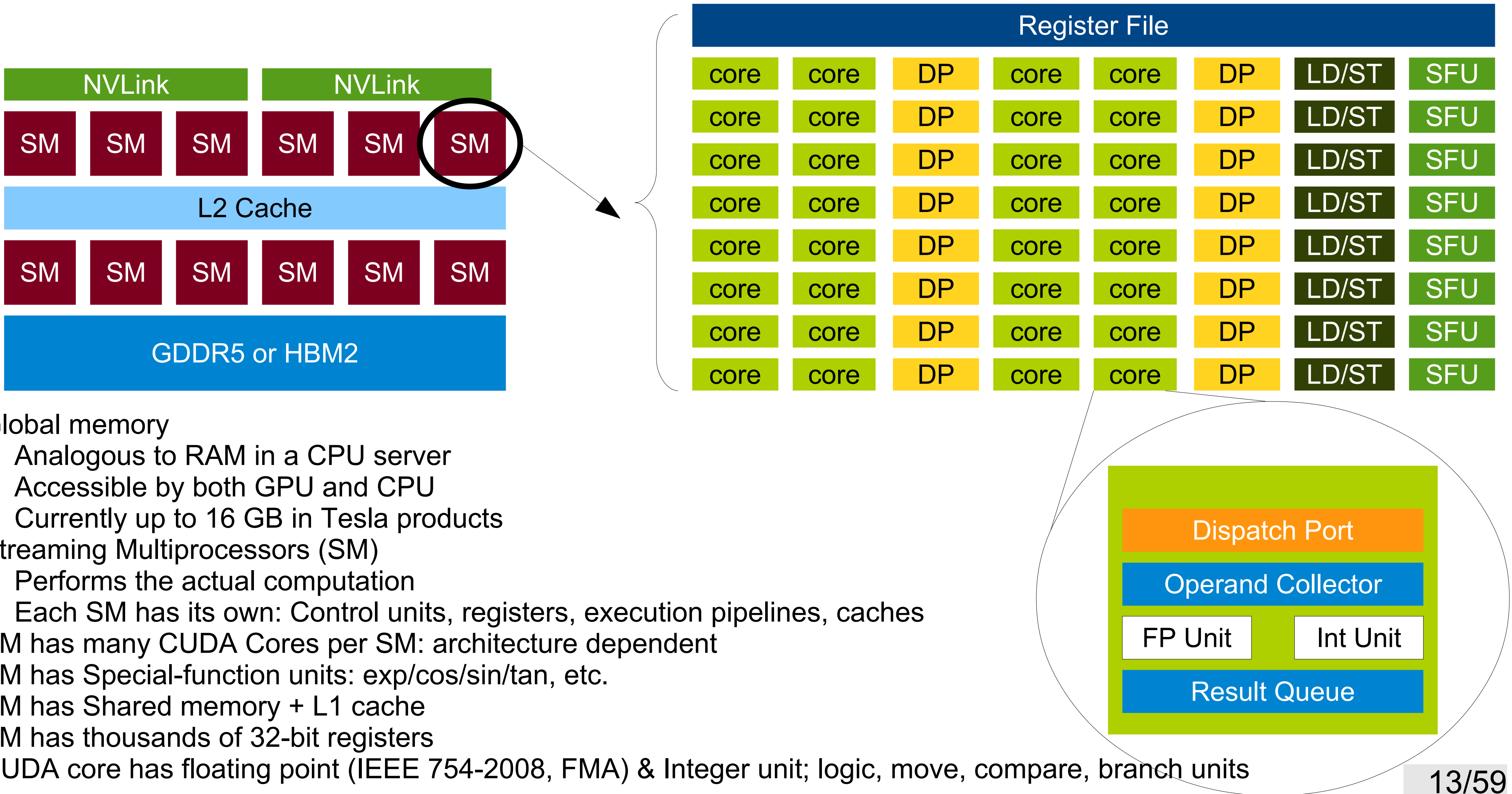
Hardware: CPU vs. GPU



Main Memory RAM: DDR3 or DDR4
Size: ~500 GiB
Bus width: 64 bits
Speed: ~100 GB/s

GPU Memory RAM: GDDR5
Size: ~10 GiB
Bus width: 384 bits
Speed: ~200 GB/s

GPU Device Structure



- Global memory
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 16 GB in Tesla products
- Streaming Multiprocessors (SM)
  - Performs the actual computation
  - Each SM has its own: Control units, registers, execution pipelines, caches
- SM has many CUDA Cores per SM: architecture dependent
- SM has Special-function units: exp/cos/sin/tan, etc.
- SM has Shared memory + L1 cache
- SM has thousands of 32-bit registers
- CUDA core has floating point (IEEE 754-2008, FMA) & Integer unit; logic, move, compare, branch units

GPU Execution Model: Abstraction above Device

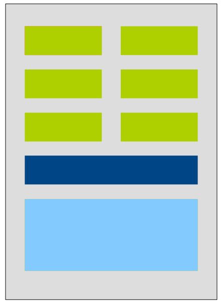
Hardware

Software

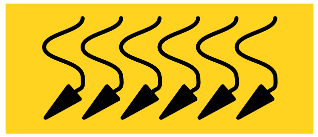
Scalar Processor

Thread

Threads are executed by scalar processors



Multiprocessor

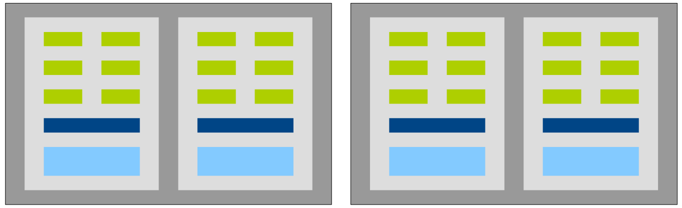


Thread Block

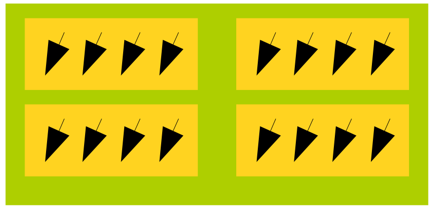
Thread blocks are executed on multiprocessor

Thread blocks do not migrate

Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)



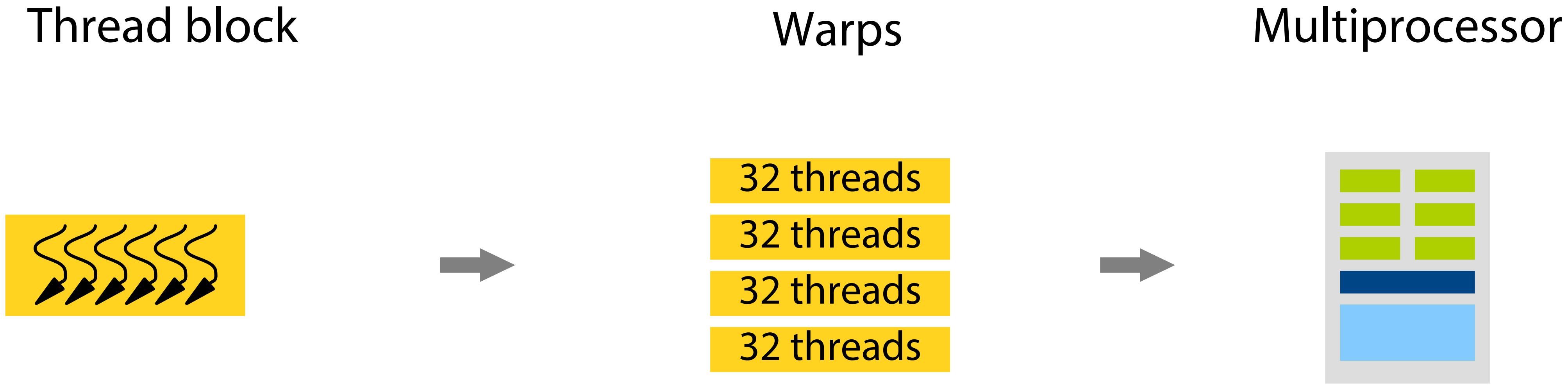
Device



Grid

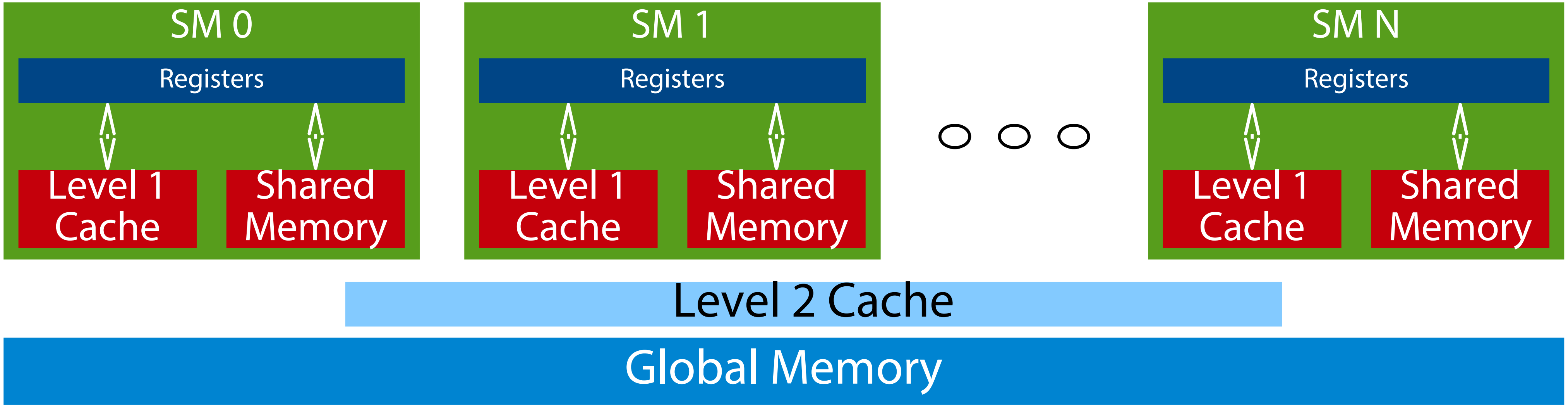
A kernel is launched as a grid of thread blocks

GPU Warp: Basic Unit of Hardware Execution



- A thread block consists of 32-thread warps
- A warp is executed physically in parallel (SIMT) on a multiprocessor

GPU Memory Hierarchy



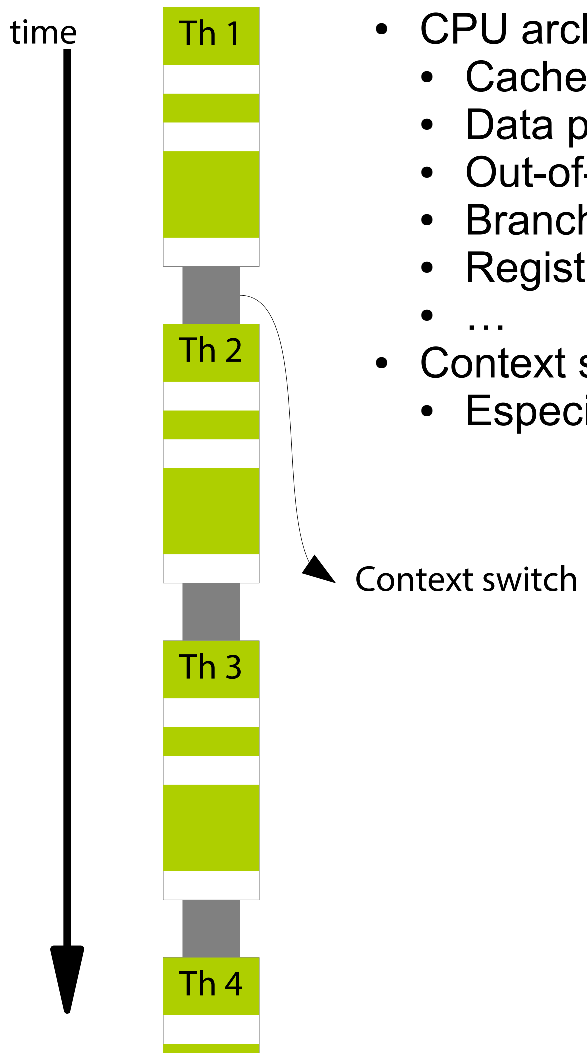
Memory System on each SM

- Extremely fast, but small, i.e., 10s of kB
- Programmer chooses whether to use cache as L1 or Shared Memory
- L1
  - Hardware-managed—used for things like register spilling
  - Should NOT attempt to utilize like CPU caches
- Shared Memory—programmer MUST synchronize data accesses!!!
  - User-managed scratch pad
  - Repeated access to same data or multiple threads with same data

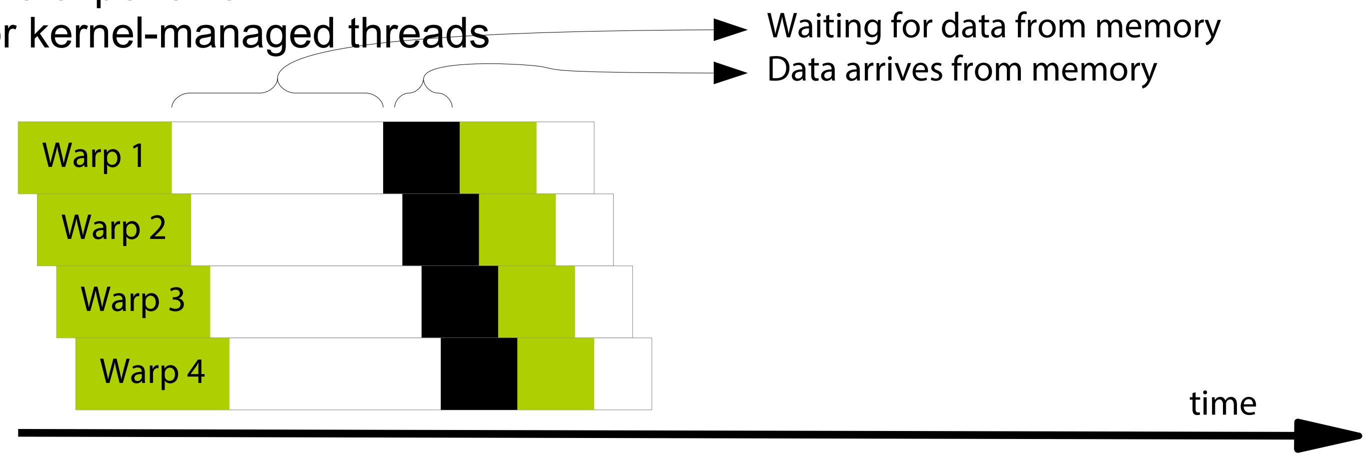
Memory system on each GPU board

- Unified L2 cache (100s of kB)
  - Fast, coherent data sharing across all cores in the GPU
- Unified/Managed Memory
  - Since CUDA 6 it's possible to allocate 1 pointer (virtual address) whose physical location will be managed by the runtime.
  - Pre-Pascal GPUs —managed by software, limited to GPU memory size
  - Pascal & Beyond —Hardware can page fault to manage location, can oversubscribe GPU memory

Low-Latency vs. Throughput-Oriented Processing



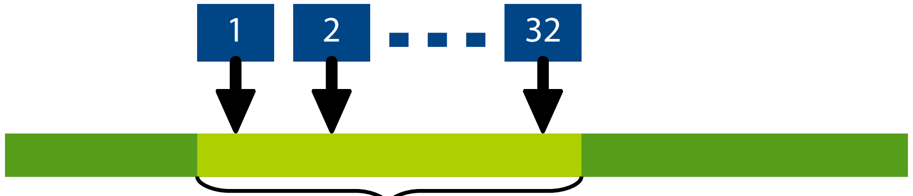
- CPU architecture minimizes latency within each thread with:
  - Cache hierarchy: L1, L2, L3, L4
  - Data prefetcher
  - Out-of-order scheduler
  - Branch predictor
  - Register renaming
  - …
- Context switch is expensive
  - Especially for kernel-managed threads



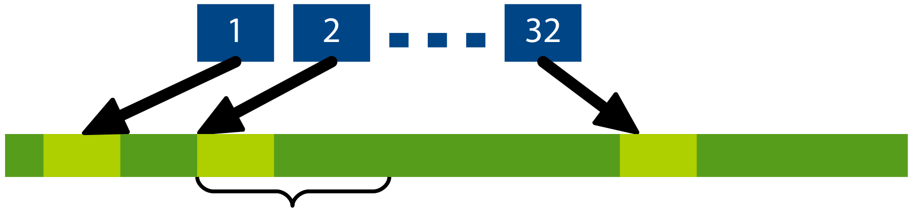
- GPU architecture hides latency with computation from other thread warps
- GPU scheduler can switch between warps in a few cycles because all resources are already allocated on SM

Memory Coalescing for Reads and Writes

- Global memory access happens in transactions of 32 or 128 bytes only
- The hardware will try to reduce requests to as few transactions as possible
- Coalesced access:
  - A group of 32 contiguous threads (“warp”) accessing adjacent words in global memory
  - Fewer transactions and high utilization of GDDR5/HBM2 bandwidth
- Uncoalesced access:
  - A warp of 32 threads accessing scattered words
  - Many transactions and low utilization of bandwidth
- Optimization tip:
  - If the data must be scattered, try to gather the data into shared memory explicitly and load/store from there

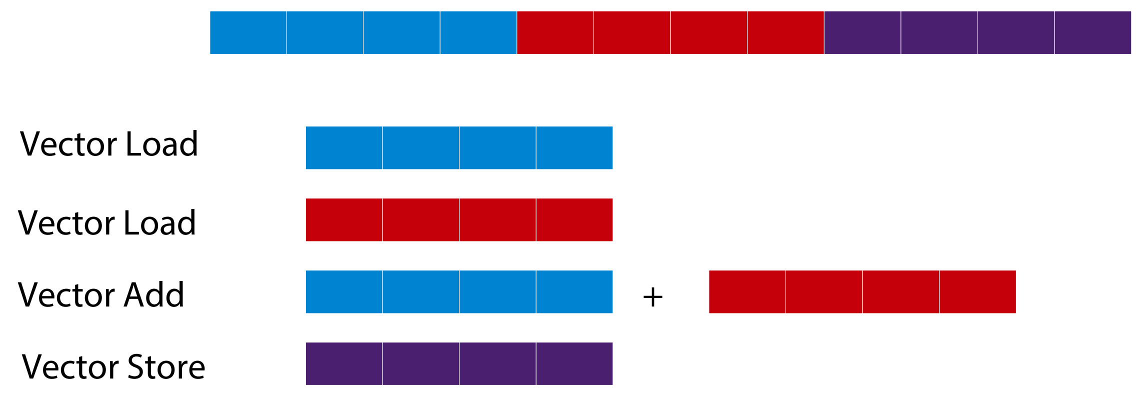


Single transaction of 128 bytes = 32 x 4 bytes



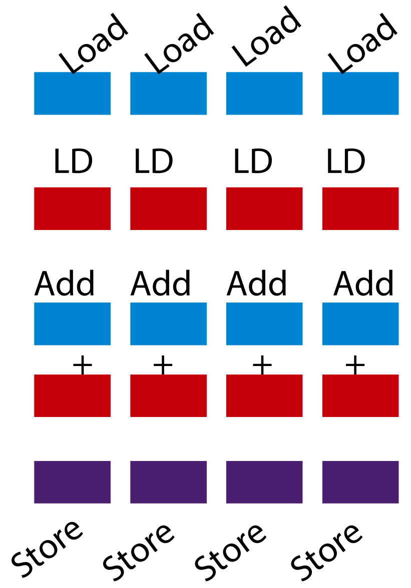
32 bytes in transaction but only 4 bytes used:
4 / 32 = 12.5% bandwidth used

CPU SIMD (SSE, AVX, AltiVec, NEON) vs GPU SIMT



Single Instruction Multiple Data (SIMD)

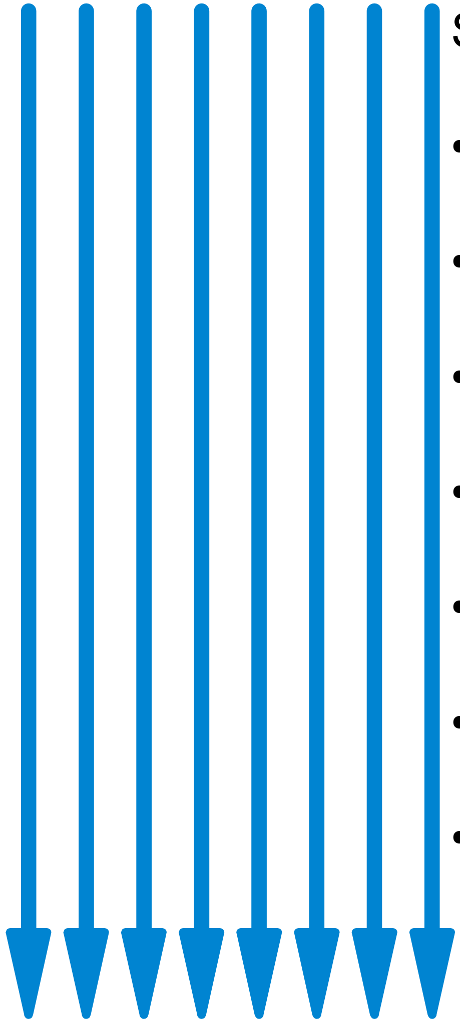
- Vector instructions perform the same operation on multiple data elements.
- Data must be loaded and stored in contiguous buffers
- Either the programmer or the compiler must generate vector instructions



Single Instruction Multiple Thread (SIMT)

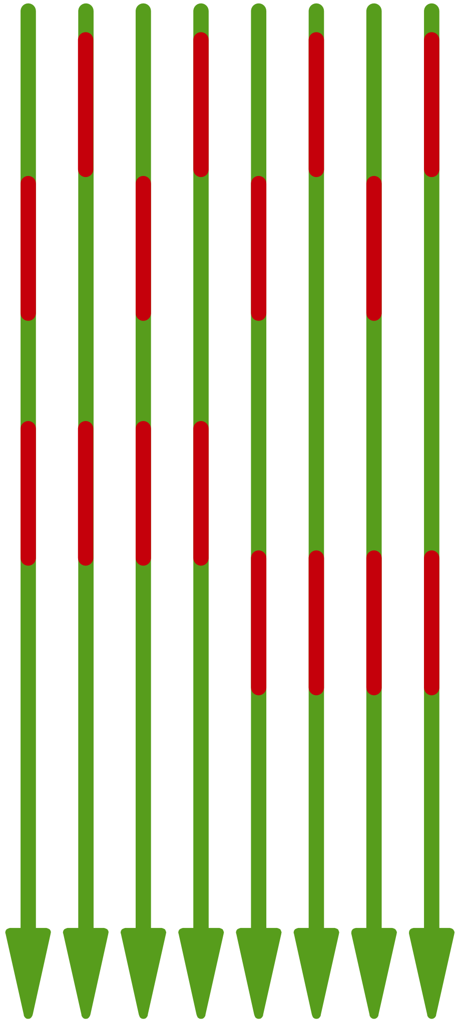
- Scalar instructions execute simultaneously by multiple hardware threads
- Contiguous data not required.
- So if something can run in SIMD, it can run in SIMT, but not necessarily the reverse.
- SIMT can better handle indirection
- The hardware enables parallel execution of scalar instructions

SIMD and SIMT Branching and Converged Execution



SIMD

- Execute converged instructions
- Generate vector mask for true
- Execute masked vector instruction
- Generate vector mask for false
- Execute masked vector instruction
- Continue to execute converged instructions
- Divergence (hopefully) handled by compiler through masks and/or gather/scatter operations.



SIMT

- Execute converged instructions
- Executed true branch
- Execute false branch
- Continue to execute converged instructions
- Divergence handle by hardware through predicated instructions.

CUDA Software Stack

Typical Release Content

- Kernel module
  - Linux and Windows
  - Mac OS X no longer supported
    - AMD cards used in Apple server, desktop, and laptops
    - Some report success with eGPU
      - External GPU through Lightning cable
- Compiler, linker, assembler, ...
- Libraries
  - Numerical: cuBlas, cuFFT, cuRand, cuDNN
  - Algorithm: NPP, Thrust
  - Profiling: CUPTI, …
- Tools
  - Profilers, tracers, GUI

CUDA Compiler, Assembler, and Linker

- Typical workflow
  - nvcc -c gpu-code.cu -o gpu-code.o
    - Accepts .cu source code and produces object file .o
  - nvcc gpu-code.o -o gpu-code
    - Accepts .o object file and produces executable binary
- Frontend compatible with GNU compiler chain
  - Many command line arguments are compatible
- Backend based on LLVM with custom optimization passes
- Assembler works in two stages
  - PTX
    - Sometimes it is called a portable assembler because it may include support for multiple devices
  - Binary
    - Binary code for a specific GPU that will trigger runtime error when run on incompatible devices
- Linker may exchange object files between GNU, LLVM, and NVCC
  - The linker must be NVCC to link in NVIDIA-specific objects

CUDA as a Programming Language

- Accepted syntax is C/C++
  - Do not use the latest C++ features
    - C++14 and C++17 are not fully supported
- Additional features
  - Function and type decorators
    - For example: __global__
  - Triple chevron notation
    - For example:
  - Built-in types

CUDA Blocks, Grids, and Threads

Piotr Luszczek

Minimal CUDA Code Example

```
__global__ void sum(double x, double y, double *z) {
  *z = x + y;
}
int main(void) {
  double *device_z, host_z;

  cudaMalloc( &device_z, sizeof(double) );

  sum<<<1,1>>>(2.0, 3.0, device_z);

  cudaMemcpy( &host_z, device_z, sizeof(double),
              cudaMemcpyDeviceToHost );

  printf(“%g\n”, host_z);

  cudaFree(device_z);

  return 0; }
```

```
$ nvcc sum.cu -o sum
$ ./sum
5
```

Structure of CUDA Code

```
// parallel function (GPU)
__global__ void sum(double x, double y, double *z) { *z = x + y; }

// sequential function (CPU)
void sum_cpu(double x, double y, double *z) { *z = x + y; }

// sequential function (CPU)
int main(void) {
  double *dev_z, hst_z;

  cudaMalloc( &dev_z, sizeof(double) );

  // launch parallel code (CPU    GPU)
  sum<<<1,1>>>(2.0, 3.0, dev_z);

  cudaMemcpy( &hst_z, dev_z, sizeof(double), cudaMemcpyDeviceToHost );

  printf("%g\n", hst_z[i]);

  cudaFree(dev_z);

  return 0;
}
```

Introducing Parallelism to CUDA Code

- Two points where parallelism enters the code
  - Kernel invocation
    - `sum<<< 1,1>>>( a, b, c )`
    - `sum<<<10,1>>>( a, b, c )`
  - Kernel execution
    - `__global__ void sum(double *a, double *b, double*c)`
    - `c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x]`
- CUDA makes the connection between:
  - invocation “sum<<<10,1>>>” with
  - execution and its index ”blockIdx.x”
- Recall GPU massive parallelism
  - Many CUDA cores
  - Many CUDA threads
  - Many GPU SM (or SMX) units

CUDA Parallelism with Blocks

```
int N = 100, SN = N * sizeof(double);
__global__ void sum(double *a, double *b, double *c) {
  c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x]; // no loop!
}

int main(void) {
  double *dev_a, *dev_b, *dev_c, *hst_a, *hst_b, *hst_c;

  cudaMalloc( &dev_a, SN ); hst_a = calloc(N, sizeof(double));
  cudaMalloc( &dev_b, SN ); hst_b = calloc(N, sizeof(double));
  cudaMalloc( &dev_c, SN ); hst_c = malloc(N, sizeof(double));

  cudaMemcpy( dev_a, hst_a, SN, cudaMemcpyHostToDevice );
  cudaMemcpy( dev_b, hst_b, SN, cudaMemcpyHostToDevice );

  sum<<<10,1>>>(dev_a, dev_b, dev_c); // only 10 elements will be used

  cudaMemcpy( &hst_c, dev_c, SN, cudaMemcpyDeviceToHost );

  for (int i=0; i<10; ++i) printf("%g\n", hst_c[i]);

  cudaFree(dev_a); free(hst_a);
  cudaFree(dev_b); free(hst_b);
  cudaFree(dev_c); free(hst_c);
  return 0; }
```

Details on Execution of Blocks on GPU

```
// BLOCK 0               // BLOCK 1

c[0]=a[0]+b[0];          c[1]=a[1]+b[1];
  // BLOCK 2             // BLOCK 3

  c[2]=a[2]+b[2];        c[3]=a[3]+b[3];
  // BLOCK 4             // BLOCK 5

  c[4]=a[4]+b[4];        c[5]=a[5]+b[5];
  // BLOCK 6             // BLOCK 7

  c[6]=a[6]+b[6];        c[7]=a[7]+b[7];
  // BLOCK 8             // BLOCK 9

  c[8]=a[8]+b[8];        c[9]=a[9]+b[9];
```

- Blocks is a level of parallelism
  - There are other levels
- Blocks execute in parallel
  - Synchronization is
    - Explicit (special function calls, etc.)
    - Implicit (memory access, etc.)
    - Mixed (atomics, etc.)
- Total number of available blocks is hardware specific
  - CUDA offers inquiry functions to get the maximum block count

Adding Thread Parallelism to CUDA Code

- Kernel invocation
  - `sum<<<10, 1>>>( x, y, z )  // block-parallel`
  - `sum<<< 1,10>>>( x, y, z )  // thread-parallel`
- Kernel execution
  - `z[threadIdx.x] = x[threadIdx.x] + y[threadIdx.x]`
- Consistency of syntax
  - Minimum changes to switch from blocks to threads
  - Similar naming for blocks and threads

CUDA Parallelism with Threads

```
int N = 100, SN = N * sizeof(double);
__global__ void sum(double *a, double *b, double *c) {
  c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x]; // no loop!
}

int main(void) { // sequential function (CPU)
  double *dev_a, *dev_b, *dev_c, *hst_a, *hst_b, *hst_c;

  cudaMalloc( &dev_a, SN ); hst_a = calloc(SN);
  cudaMalloc( &dev_b, SN ); hst_b = calloc(SN);
  cudaMalloc( &dev_c, SN ); hst_c = malloc(SN);

  cudaMemcpy( dev_a, hst_a, SN, cudaMemcpyHostToDevice );
  cudaMemcpy( dev_b, hst_b, SN, cudaMemcpyHostToDevice );

  sum<<<1,10>>>(dev_a, dev_b, dev_c);

  cudaMemcpy( &hst_c, dev_c, SN, cudaMemcpyDeviceToHost );

  for (int i=0; i<10; ++i) printf(“%g\n”, hst_c[i]);

  cudaFree(dev_a); free(hst_a);
  cudaFree(dev_b); free(hst_b);
  cudaFree(dev_c); free(hst_c);
  return 0; }
```

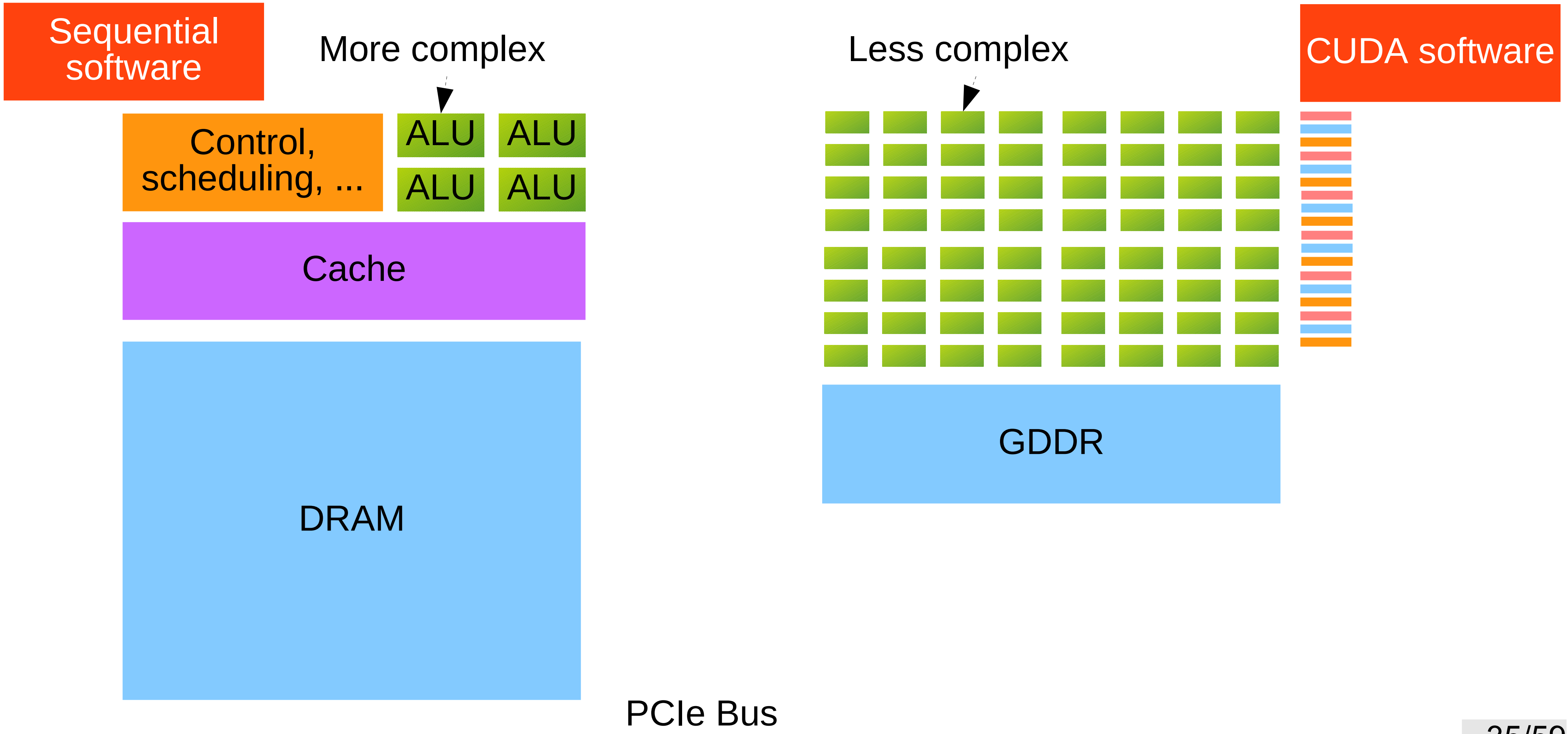
More on Block and Thread Parallelism

- When to use blocks and when to use threads?
  - Synchronization between threads is cheaper
  - Blocks have higher scheduling overhead
- Block and thread parallelism can be combined
  - Often it is hard to get good balance between both
  - Exact combination depends on
    - GPU generation
      - Tesla, Fermi, Kepler, Maxwell, Pascal, Volta, ...
    - SM/SMX configuration
    - Memory size

Thread Identification Across APIs

- POSIX threads
  - `pthread_t tid = pthread_self();`
- MPI
  - `MPI_Comm_rank(comm, &rank);`
  - `MPI_Comm_size(comm, &size);`
- OpenMP
  - `int tid = omp_get_thread_num();`
  - `int all = omp_get_num_threads();`
- CUDA
  - `int blkid = blockIdx.x + (blockIdx.y + blockIdx.z * gridDim.y) * gridDim.x`
  - `int inside_blk_tid = threadIdx.x + (threadIdx.y + threadIdx.z * blockDim.y) * blockDim.x`

GPU Optimization: Target the Hardware



Matrix Summation: CPU

```
float A[n][n], B[n][n], C[n][n];

/* note the order of the loops */
for (int i=0; i<n; ++i)
  for (int j=0; j<n; ++j)
    C[i][j] = A[i][j] + B[i][j];
    // row-major order
```

Matrix Summation: GPU Kernel

```
__global__ void matrix_sum(float *A, float *B, float *C, int m,
int n) {
  int x = threadIdx.x + blockIdx.x * blockDim.x;
  int y = threadIdx.y + blockIdx.y * blockDim.y;

  if (x < m && y < n) {
    int ij = x + y*m; // column-major order
    C[ij] = A[ij] + B[ij];
  }
}
```

Matrix Summation: GPU Launching (Slow!)

```
// optimization: copy data outside of the loop
cudaMemcpy(dA,...);
cudaMemcpy(dB,...);
for (int i=0; i<n; ++i)
  for (int j=0; j<n; ++j) {
    int ij = i + j*n; // column-major order
    matrix_sum<<<1,1>>>(dA+ij, dB+ij, dC+ij, 1,1);
    // problem: kernel launch overhead 1-10 ms
  }
cudaMemcpy(hC,dC,...);
```

Matrix Summation: Faster Launching

- Kernel launch for every row

```
for (int i=0; i<n; ++i)
  matrix_sum<<1,n>>(dA+i, dB+i, dC+i, 1,n );
```

- Kernel launch for every column

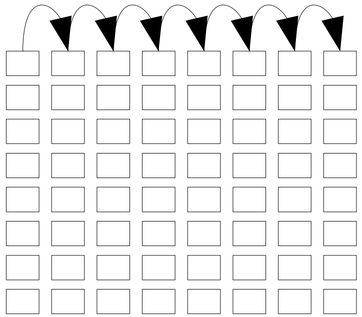
```
for (int j=0; j<n; ++j)
  int k = j*n;
  matrix_sum<<n,1>>(dA+k, dB+k, dC+k, n,1 );
```

- Kernel launch for all rows and columns at once

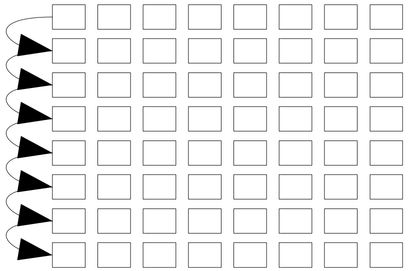
```
matrix_sum<<n,n>>(dA, dB, dC, n,n );
```

  - Single point to incur kernel-launch overhead
  - Might run into hardware limits on threads, thread blocks, and their dimensions

Ordering Matrix Elements in Memory

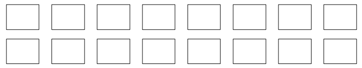


Row-major order in C/C++

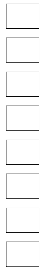


Column-major order in Fortran

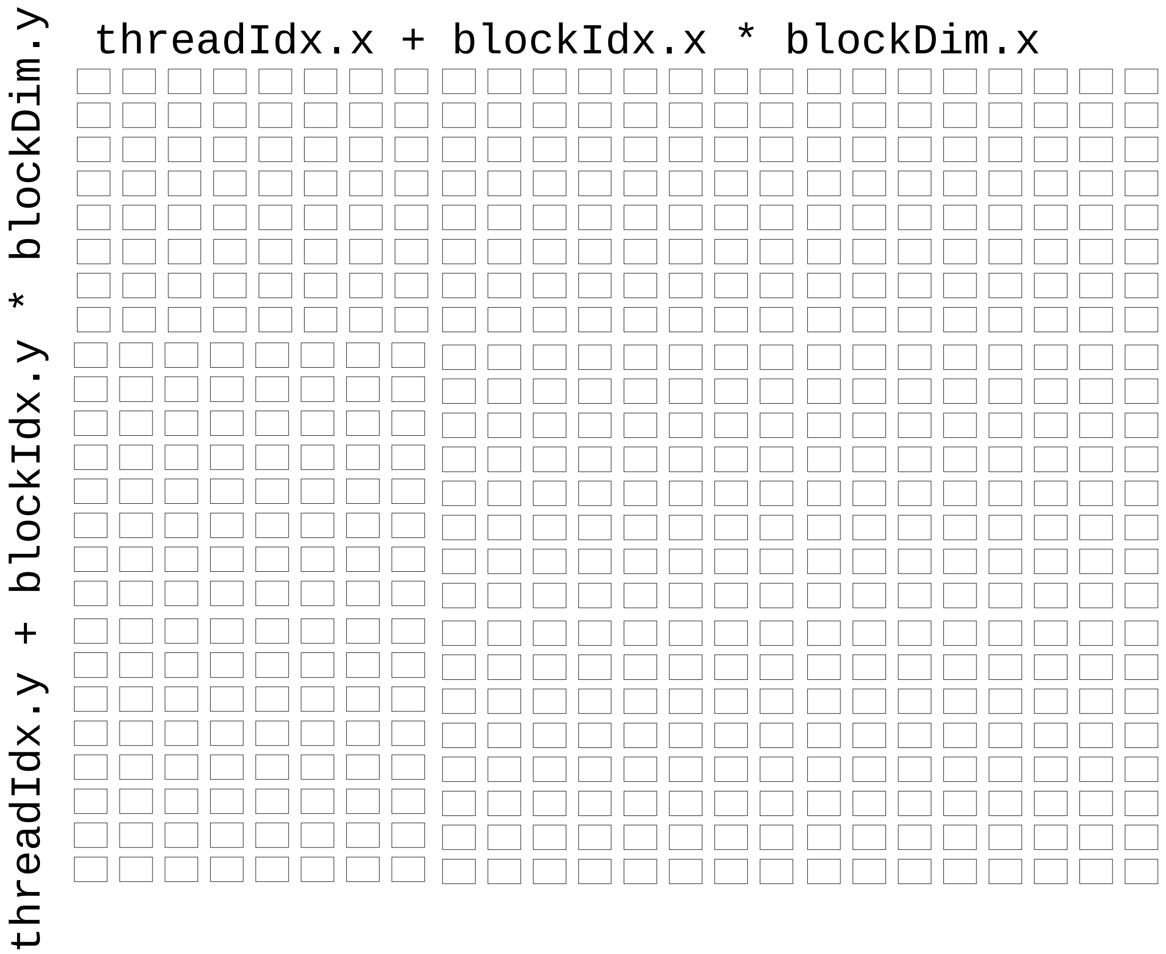
```
A + 0x0
A + 0x40
A + 0x80
A + 0xC0
A + 0x100
```



```
A + 0x0
A + 0x4
A + 0x8
A + 0xC
A + 0x10
A + 0x14
```



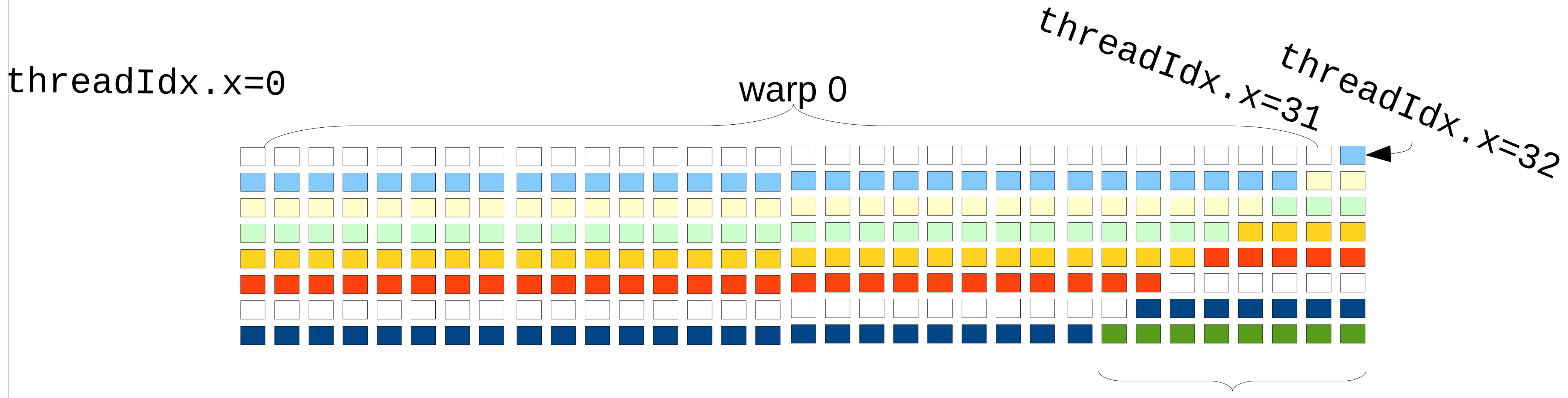
Mapping Threads to Matrix Elements


threadIdx.x + blockIdx.x * blockDim.x
threadIdx.y + blockIdx.y * blockDim.y

Scheduling Thread on a GPU

- Programming model for GPUs is SIMT
  - Many threads (ideally) execute the same instruction on different data
  - Performance drops quickly if threads don’t execute the same instruction
- Basic unit of scheduling is called a warp
  - The size of warp is (and has been) 32 threads
  - If one of the threads in a warp stalls then entire warp is de-scheduled and another warp is picked
  - Threads are assigned to warp with x-dimension changing fastest
- Some operations can only be performed on half-warp
  - Some GPU cards only have 16 load/store units per SM
  - Each half-warp in a full warp will be scheduled to issue a load one after the other

Mapping Threads In a Block to Warps



Remaining threads in the block will be mapped to an incomplete warp. This is inefficient and incomplete warps should be avoided.

GPU Warp Scheduling Details

- GPU has at least one warp scheduler per SM
  - With newer GPU hardware cards, this number increases
- The scheduler picks an eligible warp and executes all threads in the warp
- If any of the threads in the executing warp stalls (uncached memory read, etc.) the schedule makes it inactive
- If there are no eligible warps left then GPU idles
- Context switch between warps is fast
  - About 1 or 2 cycles (1 nano-second on 1 GHz GPU)
  - The whole thread block has resources allocated on an SM (by the compiler) ahead of time

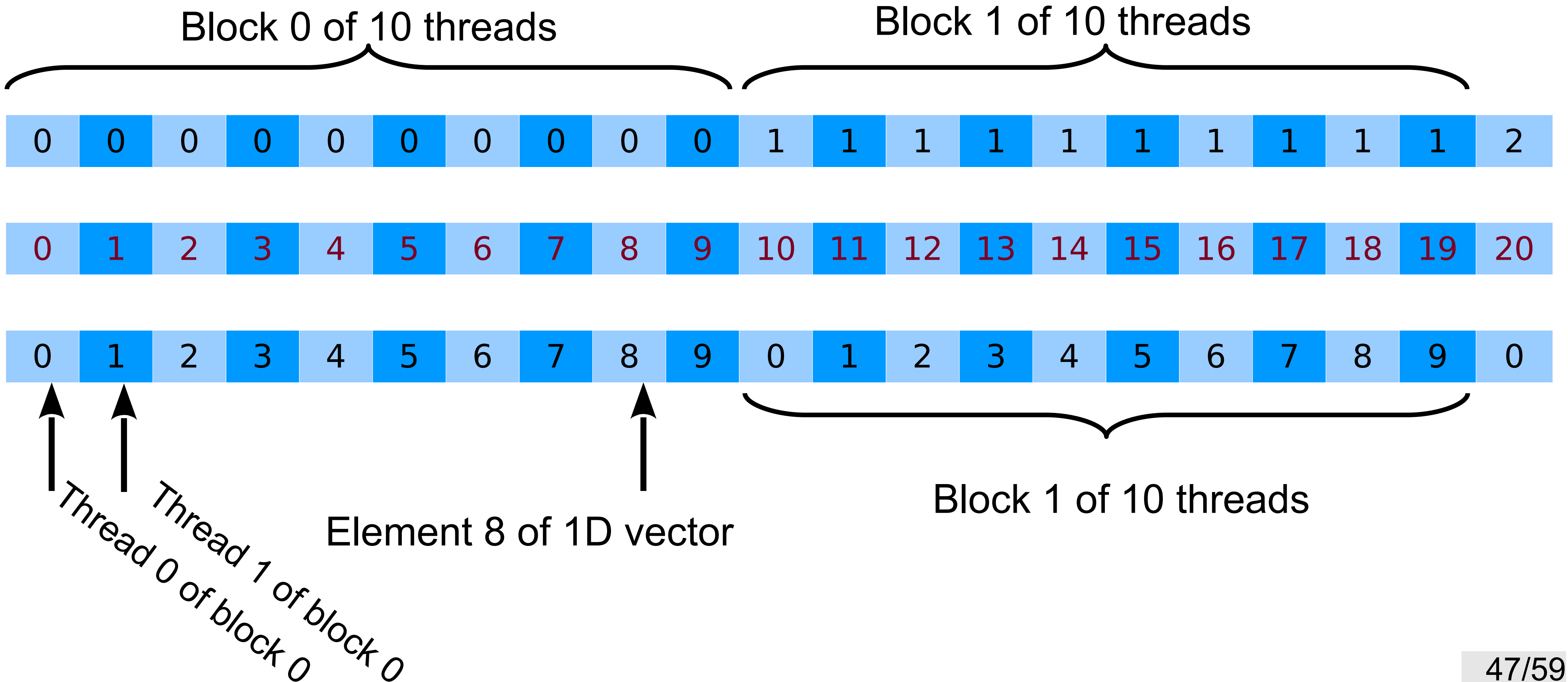
CUDA – Beyond Basics

Piotr Luszczek

Mixing Blocks and Threads

```
int N = 100, SN = N * sizeof(double);
__global__ void sum(double *a, double *b, double *c) {
  int idx = threadIdx.x + blockIdx.x * blockDim.x;
  c[idx] = a[idx] + b[idx]; // no loop!
}
int main(void) {  double *dev_a, *dev_b, *dev_c, *hst_a, *hst_b, *hst_c;

  cudaMalloc( &dev_a, SN ); hst_a = calloc(N, sizeof(double));
  cudaMalloc( &dev_b, SN ); hst_b = calloc(N, sizeof(double));
  cudaMalloc( &dev_c, SN ); hst_c = malloc(N, sizeof(double));

  cudaMemcpy( dev_a, hst_a, SN, cudaMemcpyHostToDevice );
  cudaMemcpy( dev_b, hst_b, SN, cudaMemcpyHostToDevice );

  sum<<<10,10>>>(dev_a, dev_b, dev_c); // all 100 elements will be used

  cudaMemcpy( &hst_c, dev_c, SN, cudaMemcpyDeviceToHost );

  for (int i=0; i<10; ++i) printf(“%g\n”, hst_c[i]);

  cudaFree(dev_a); free(hst_a);
  cudaFree(dev_b); free(hst_b);
  cudaFree(dev_c); free(hst_c);

  return 0;
}
```

Block/Thread Indexing with <<<10,10>>>



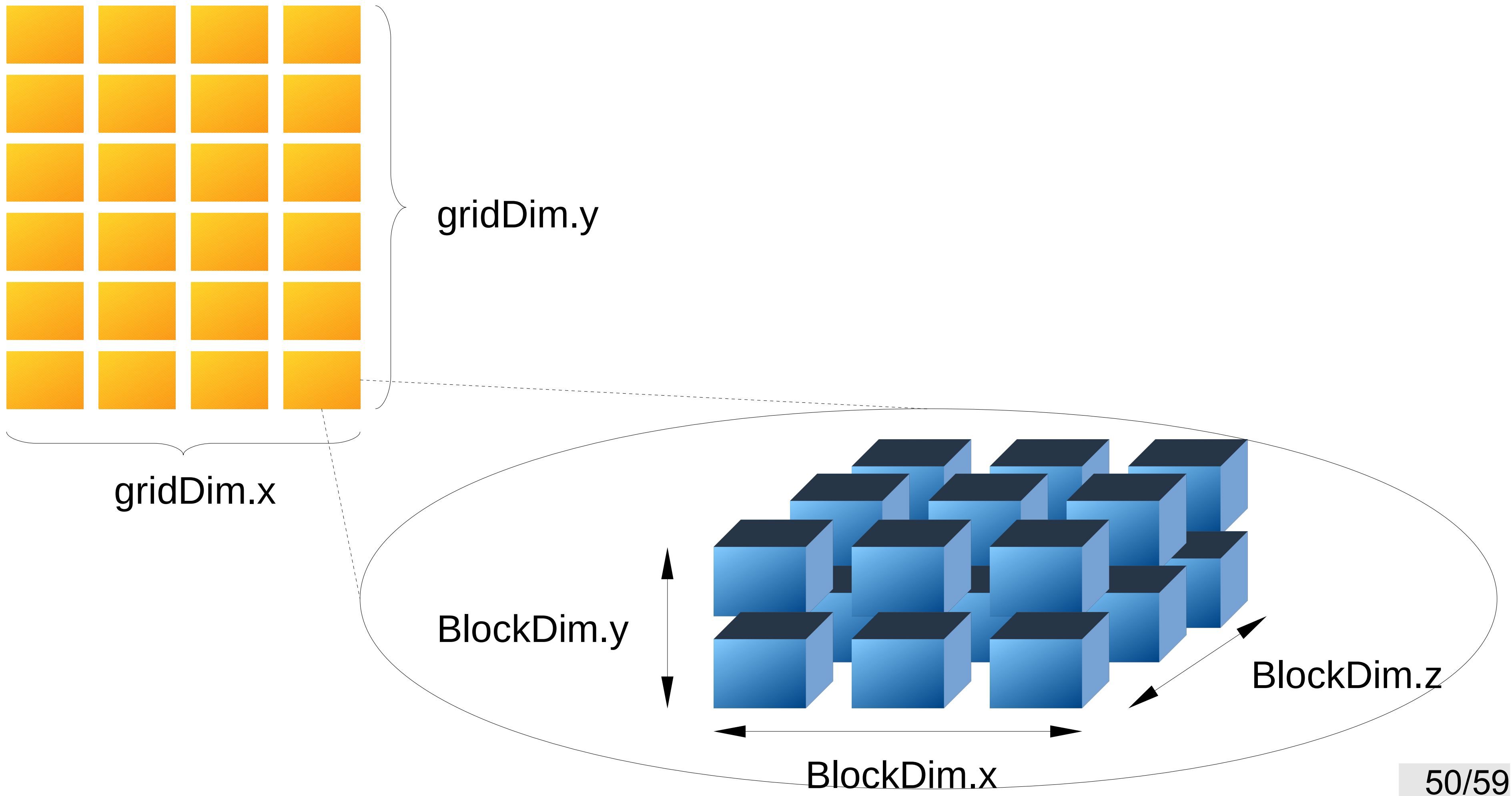
Mixing Blocks and Threads and Loop

```
int N = 1000, SN = N * sizeof(double);
__global__ void sum(double *a, double *b, double *c) {
  int idx = (threadIdx.x + blockIdx.x * blockDim.x) * 10;
  for (int i=0; i<10; ++i) c[idx+i] = a[idx+i] + b[idx+i]; // use loop
}
int main(void) {  double *dev_a, *dev_b, *dev_c, *hst_a, *hst_b, *hst_c;

  cudaMalloc( &dev_a, SN ); hst_a = calloc(N, sizeof(double));
  cudaMalloc( &dev_b, SN ); hst_b = calloc(N, sizeof(double));
  cudaMalloc( &dev_c, SN ); hst_c = malloc(N, sizeof(double));

  cudaMemcpy( dev_a, hst_a, SN, cudaMemcpyHostToDevice );
  cudaMemcpy( dev_b, hst_b, SN, cudaMemcpyHostToDevice );

  sum<<<10,10>>>(dev_a, dev_b, dev_c);

  cudaMemcpy( &hst_c, dev_c, SN, cudaMemcpyDeviceToHost );

  for (int i=0; i<10; ++i) printf(“%g\n”, hst_c[i]);

  cudaFree(dev_a); free(hst_a);
  cudaFree(dev_b); free(hst_b);
  cudaFree(dev_c); free(hst_c);

  return 0;
}
```

Grid, Blocks, Threads

- Complete syntax (almost):
  - kernel<<<gridDim, blockDim>>>
- CUDA API provides a data type: dim3
  - Grid of blocks:
    dim3 gridDim(grid_X_dimension, grid_Y_dimension)
  - Block of threads:
    dim3 blockDim(blk_X_d, blk_Y_d, blk_Z_d)

CUDA Grid of Blocks and Blocks of Threads



Limitations for and Early NVIDIA GPU

- GT200 was an early programmable GPU, It had the following limits
  - 512 threads / block
  - 1024 threads / SM
  - 8 blocks / SM
  - 32 threads / warp
- Modern GPUs, such as Volta, have limits an order of magnitude higher
- To obtain the limits of the device use a call to `cudaGetDeviceProperties`()
  - It takes device ID as a parameter so it works for multiple GPUs

Brief Notes on Function Calls in Kernels

Calling Functions Inside Kernels

```
__device__ int fibonacci(int n) { // don’t do this!!!
  if (n > 2)
    return fibonacci(n-2) + fibonacci(n-1);
  return n;
}

__device__ int gpu_function(int ix) {
  return ix + 1;
}

__global__ void kernel(int *a, int *b, int *c) {
  c[0] = a[0] + gpu_function(b[0]);
}

int main(void) {
  kernel<<<10,10>>>(dev_a, dev_b, dev_c);

  return 0;
}
```

Function Calling: Behind the Scenes

- On CPU, function calls are achieved with
  - Stack register
  - Instructions to call and return from a function
  - Main memory devoted to stack
    - Contains return address and variable values
- On GPU, these features are missing to simplify the design of GPU cores
- The CUDA compiler has to do the work to make calls to device functions possible
  - This is mostly done with code inlining:

```
__device__ f() {
  g(); // code of function “g” is inserted here
}
```

- Recursive calls pose problems if the depth of recursion is unknown
  - If the compiler can determine the depth of recursion then the inlining is possible
    - fibonacci(10) is OK

Calling Host Functions

- Since Fermi generation, it is OK to call printf() from kernel code
  - It is highly discouraged, especially for debugging
  - But it may be handy on occasion
- Dynamic Parallelism was introduced in Kepler
  - It allows launching kernels directly from other kernels

Synchronization and the Use of the Asynchronous Interface

Synchronization Between Threads

- Threads within a block execute together and share:
  - L1 Cache
  - Shared memory
  - Split between the two is decided by kernel launch parameter
- Threads often do not execute all at the same time
  - But most of the time there are multiple threads executing
- They must synchronize
  - Synchronization is for all threads inside a single block
  - Blocks of threads are executed in arbitrary order
    - Gives CUDA runtime scheduling flexibility
- The most often used synchronization method
  - __syncthreads()

Sample Usage of Thread Synchronization

- Synchronization is invoked in kernel functions

  -
    ```
    __global__ kernel() {
      // parallel code section 0

      // wait for all threads to finish section 0
      __syncthreads();

      // parallel code section 1
    }
    ```
- Common mistake: not all threads reach synchronization

  -
    ```
    __global__ error_kernel() {
      if (threadIdx.x == 13) {
        // only some threads synchronize
        __syncthreads();
    }  else
        // empty code path
    }
    ```

Asynchronous CUDA Calls

- Recall the following
  - The speed of the PCIexpress bus is slow
    - Use NVLink if your hardware supports it
  - Overlapping computation and communication
  - CPUs and GPUs are independent and work in parallel
  - There may be more than one CPU/GPU installed
- There are asynchronous equivalents of many CUDA calls
  - cudaMemcpy() has asynchronous equivalent cudaMemcpyAsync()
    - cudaMemcpy() will block CPU until the copy finishes
    - cudaMemcpyAsync() returns immediately and proceeds in the background
      - Keep in mind that CPU resources are needed to make progress
  - cudaDeviceSynchronize() blocks CPU until all past asynchronous calls complete