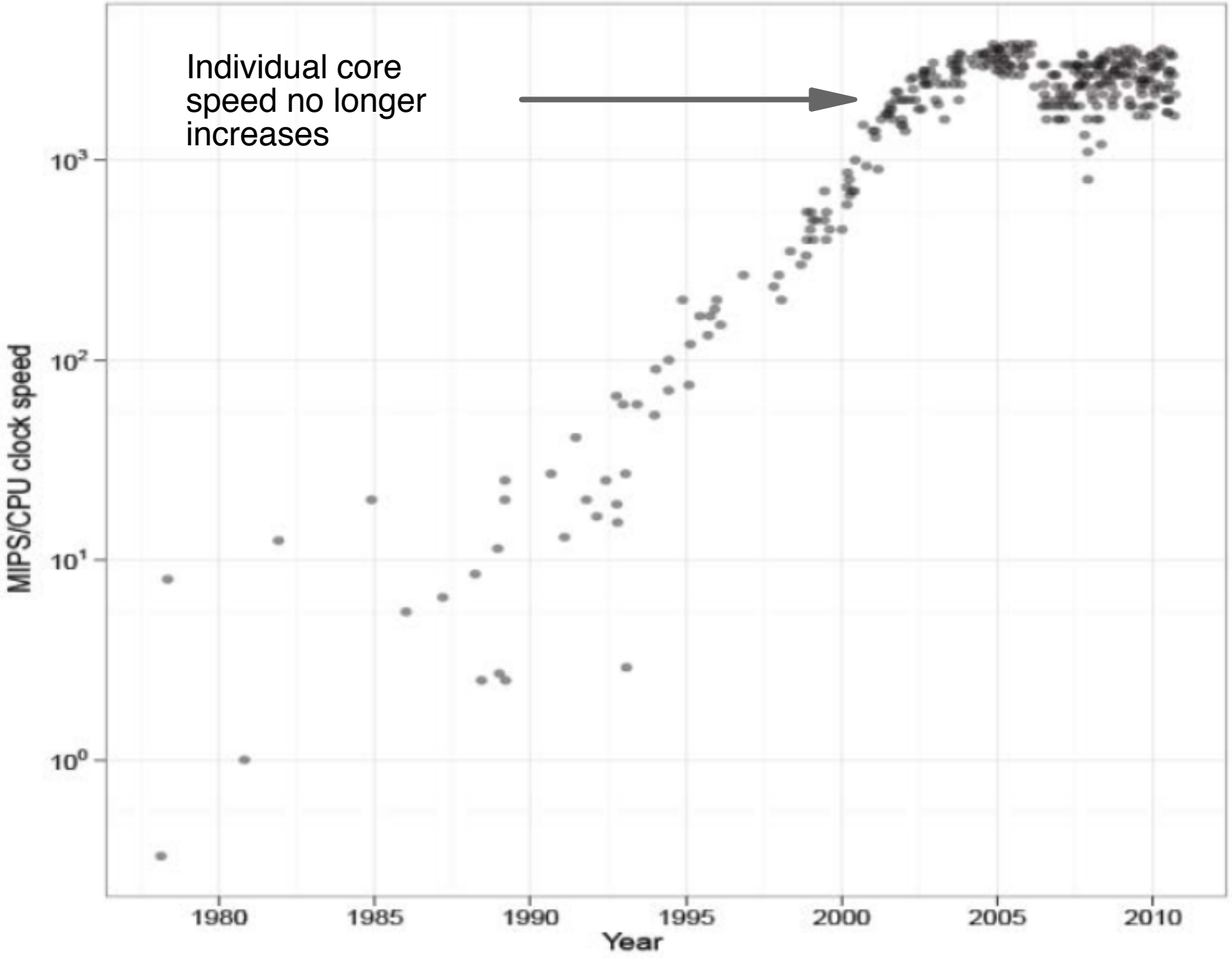
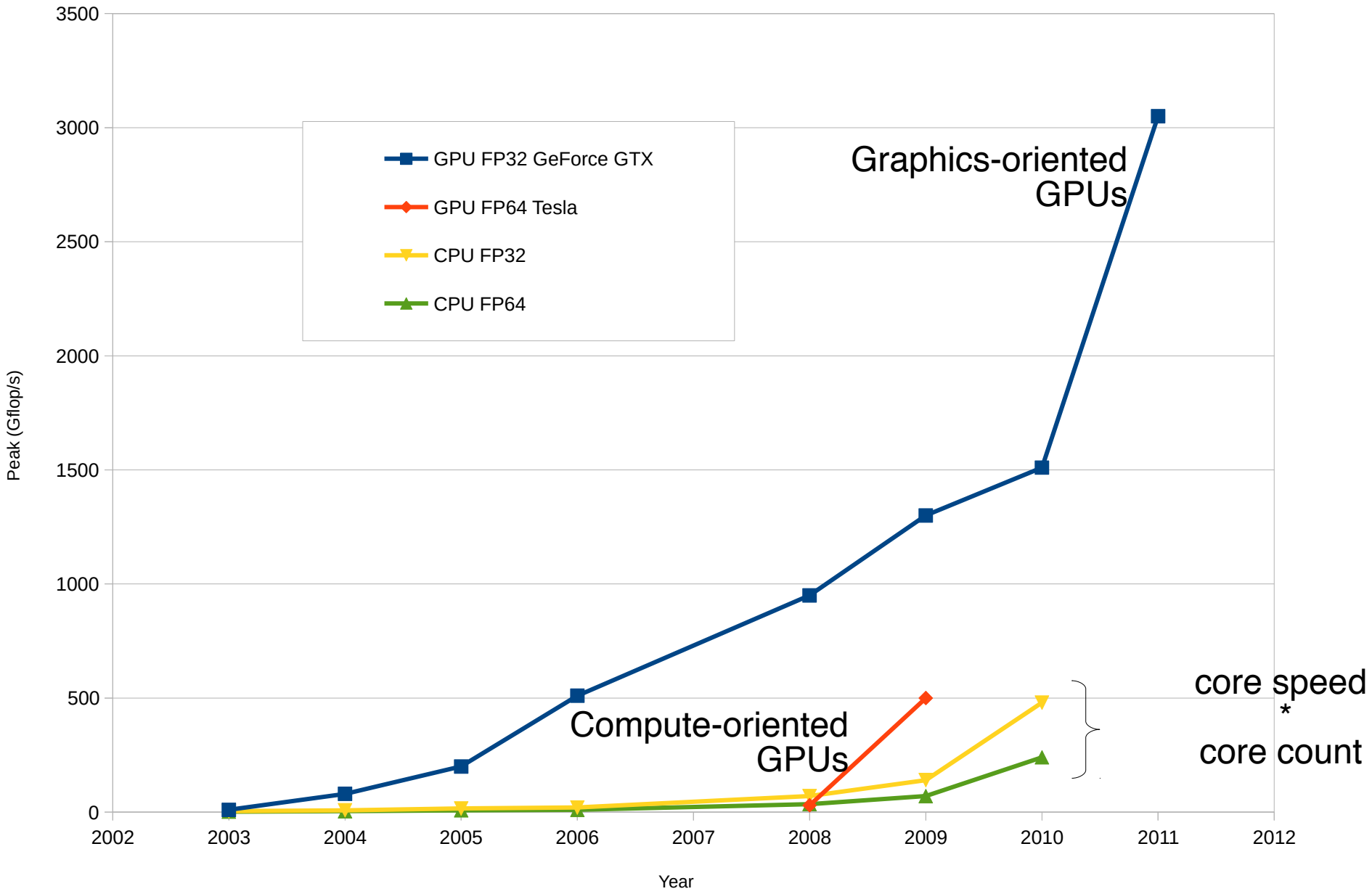
CUDA Programming

Piotr Luszczek

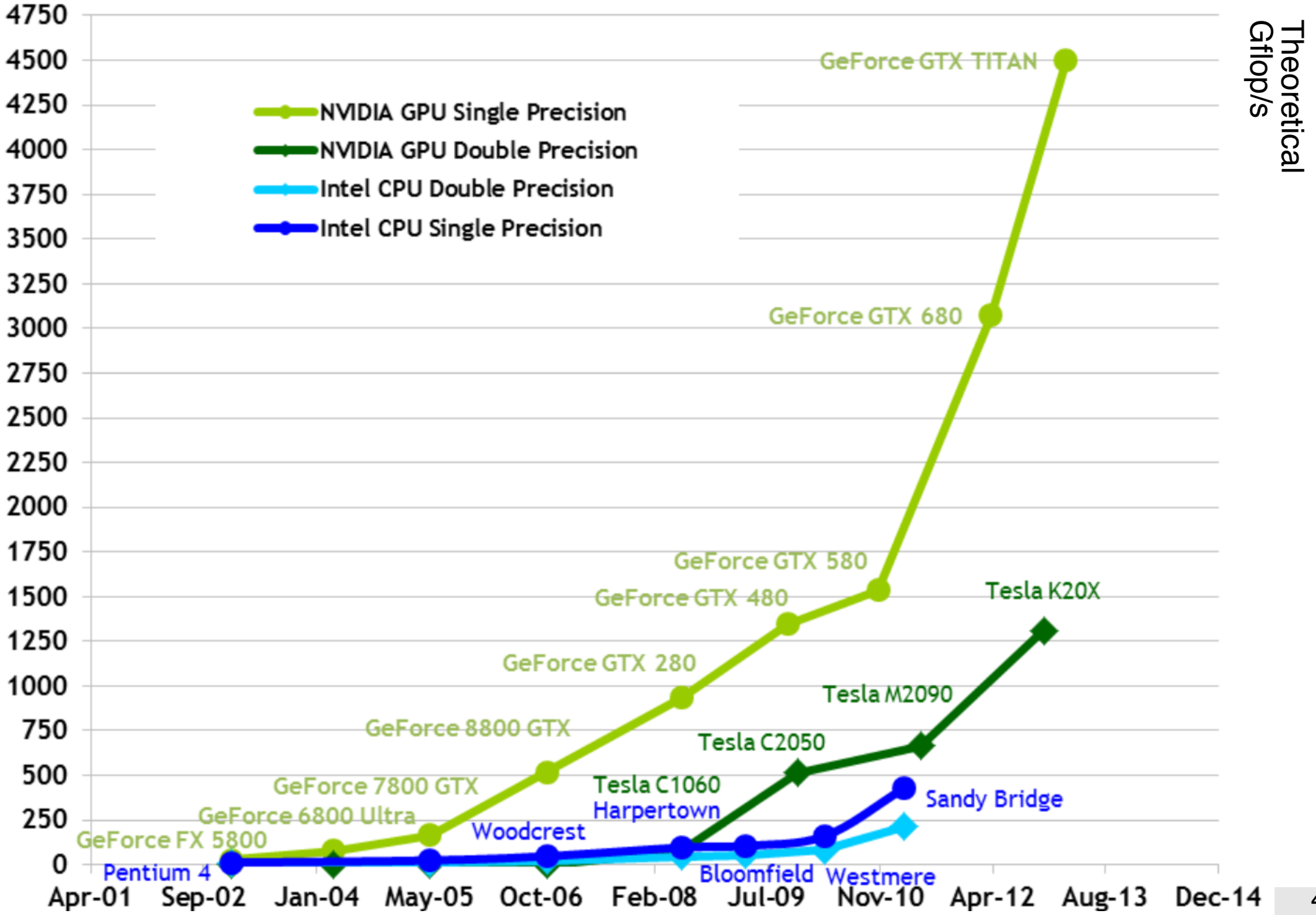
CPU and Their Per-Core Performance



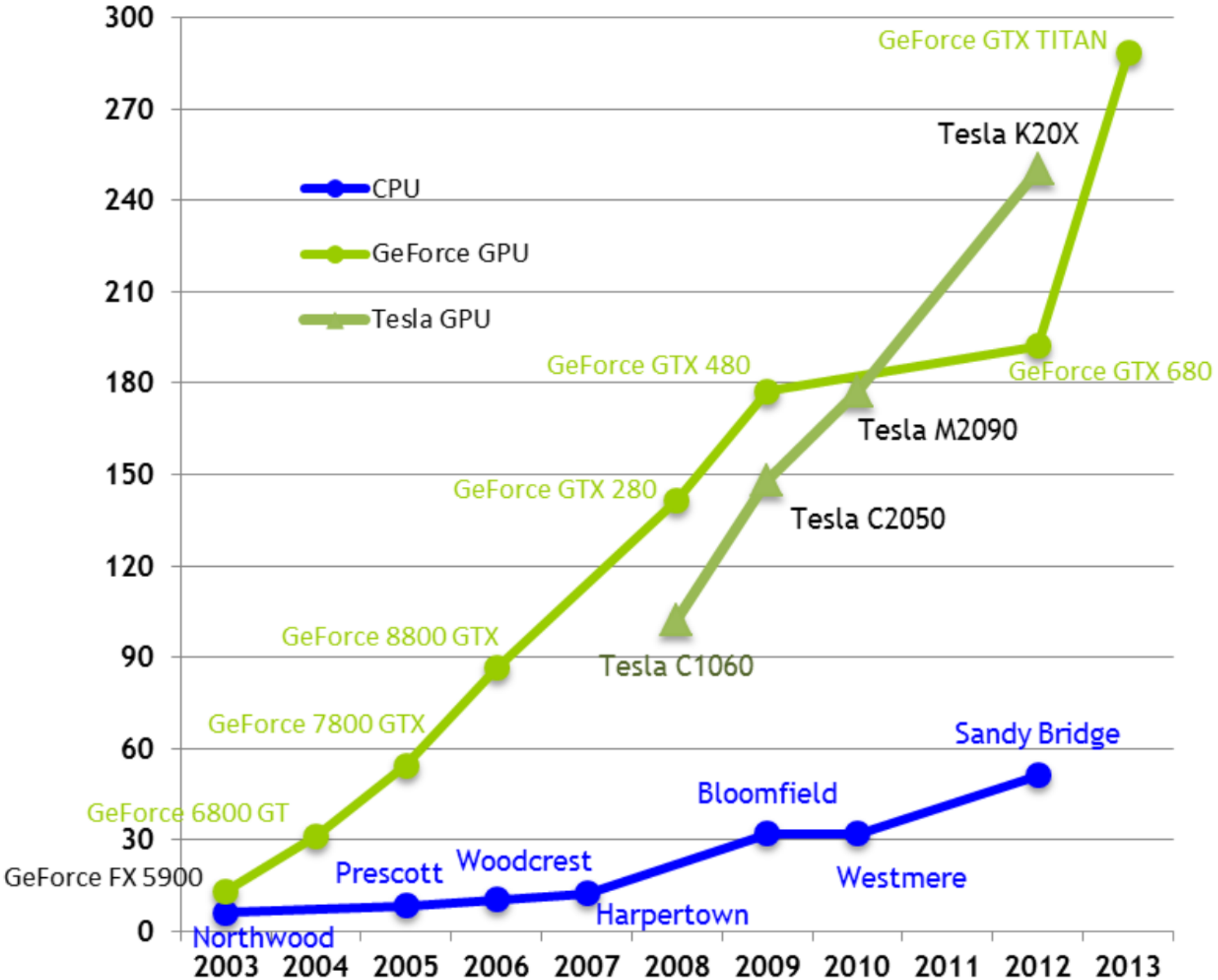
GPU vs. CPU Performance over Years



GPUs vs CPUs: Computing Power



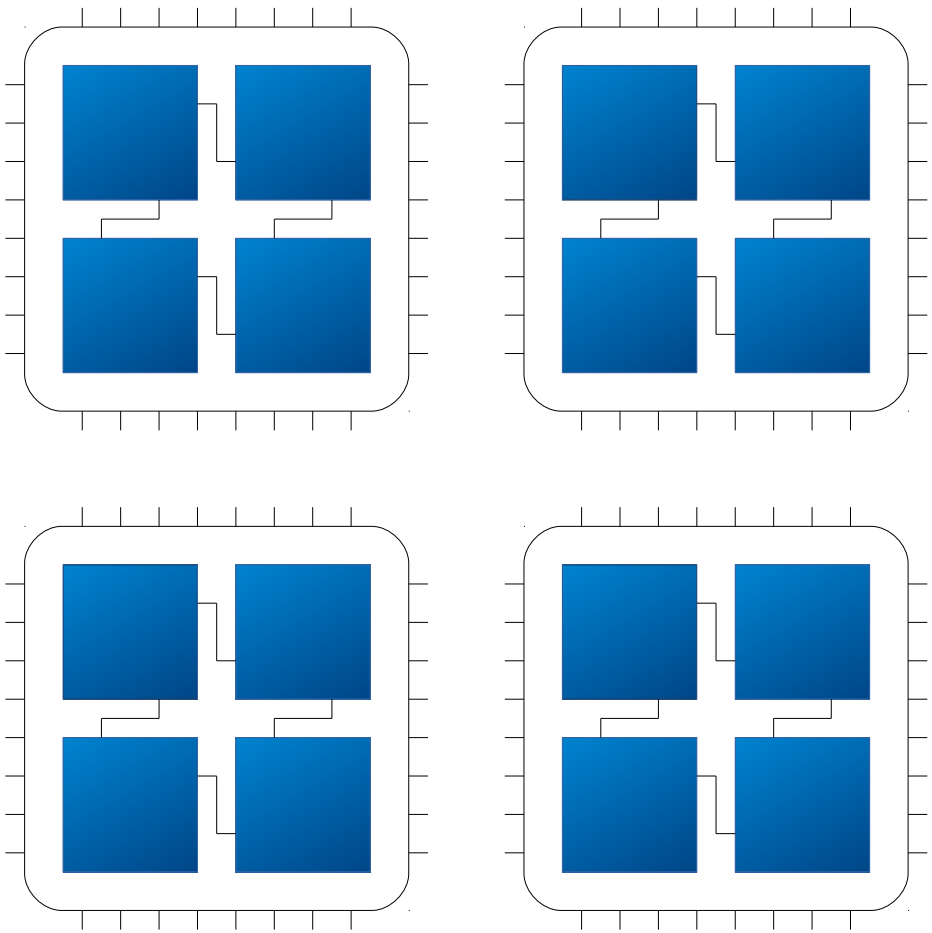
GPUs vs. CPUs: Memory Bandwidth (GB/s)

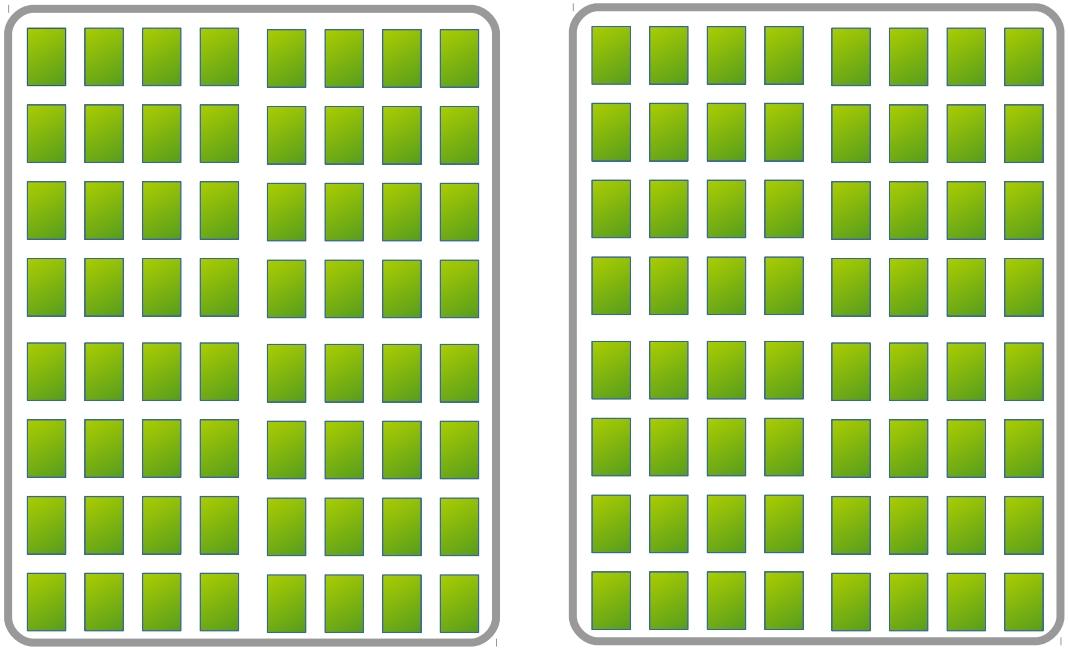


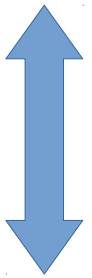
GPU and GPGPU: Origin Story

- Programmable graphics pipeline
  - GLSL
- Interpolation vs. dynamic range
  - Colors in graphics look better in floating-point
- Early attempts at programming
  - Cg, Brook, …
- Modern standards or de facto standards
  - CUDA (currently 8)
    - Compute Unified Device Architecture
  - OpenCL (currently 2)
- High-level languages
  - OpenMP 4
  - OpenACC

Hardware: CPU vs. GPU



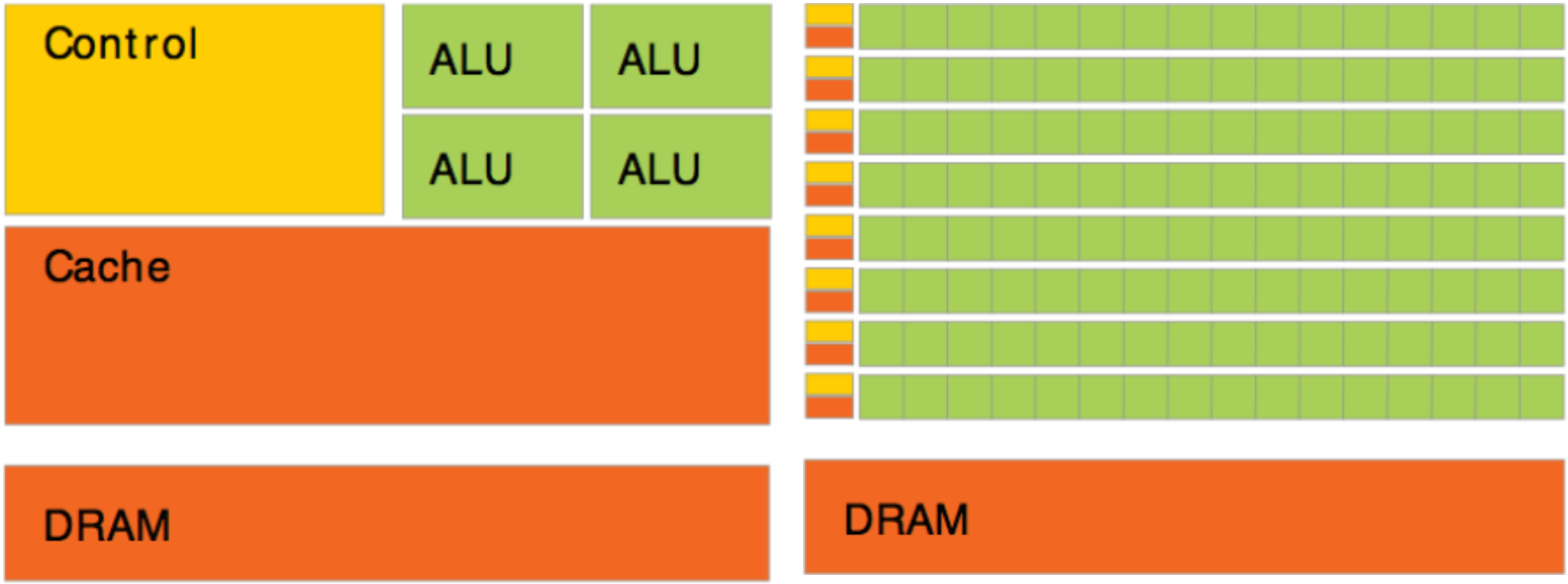




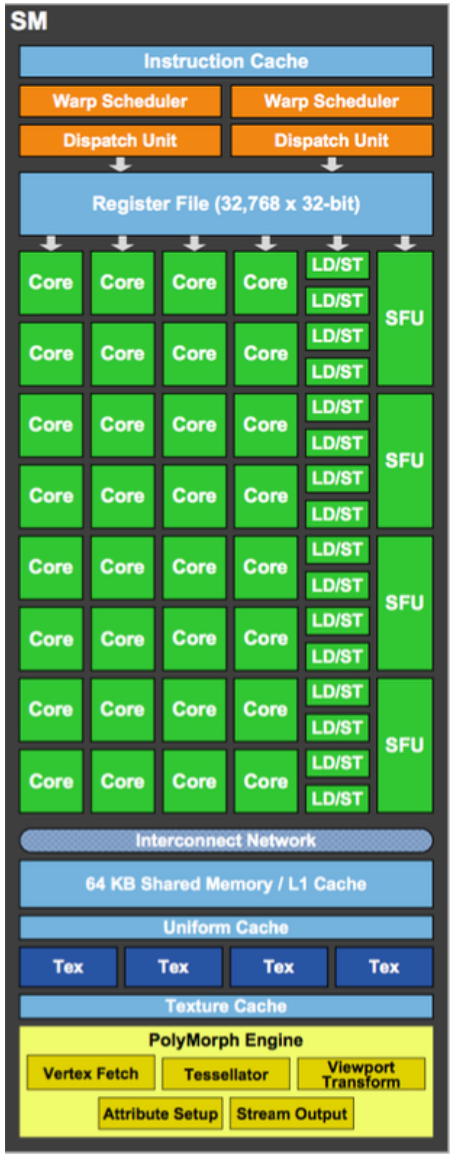
Long latency

Main Memory RAM: DDR3 or DDR4 Size: ~100 GiB Speed: ~50 GB/s	PCIexpress	GPU Memory RAM: GDDR5 Size: ~10 GiB Speed: ~200 GB/s

CPU vs. GPU: Simplified



NVIDIA Fermi

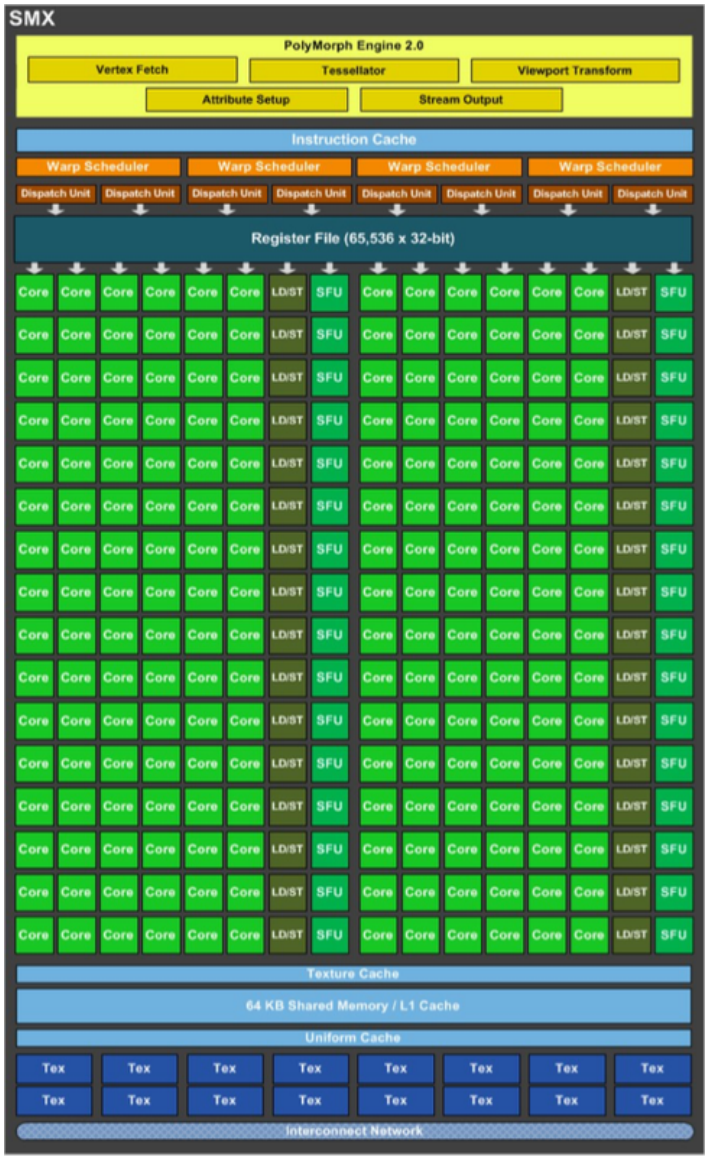


- 16 multiprocessors
- 32 cores each
- 512 total cores

NVIDIA Kepler

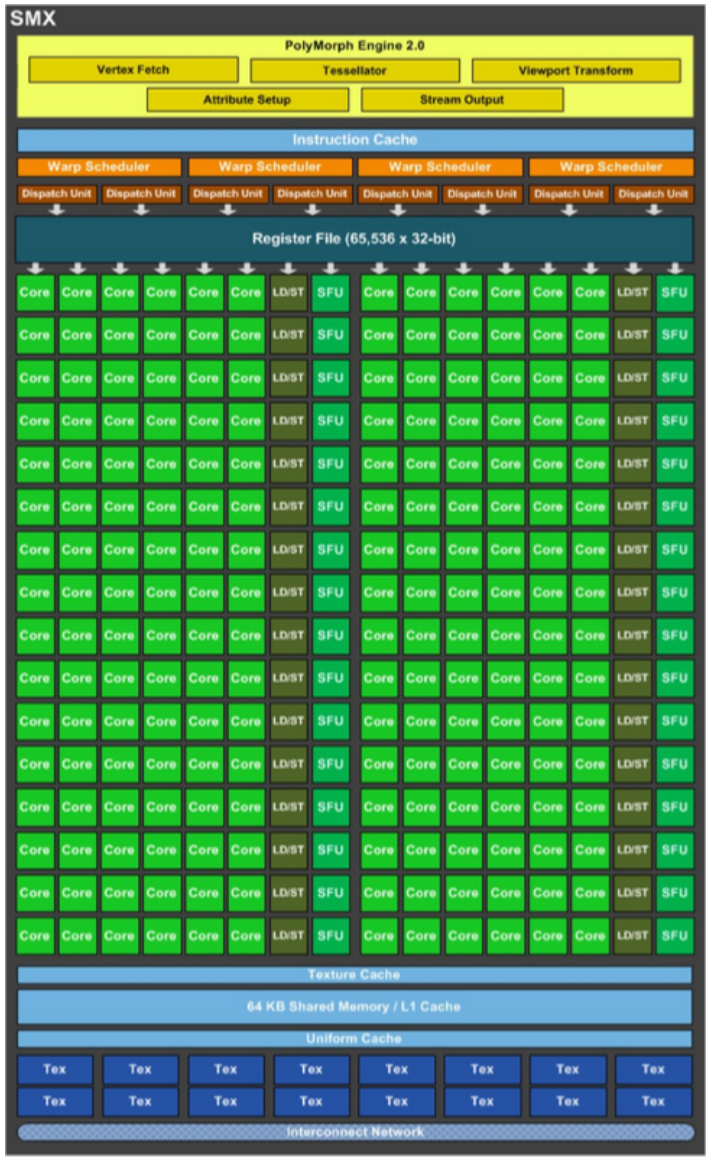
- 15 multiprocessors
- 192 cores each
- 2 880 total cores

NVIDIA Kepler: Game and HPC Editions

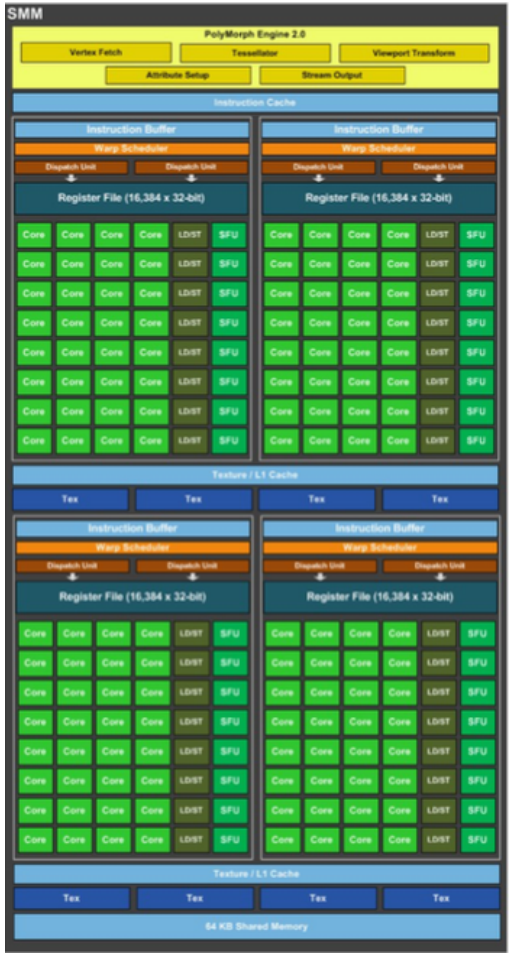


Fermi SM, Kepler SMX, Maxwell SMM

32 cores

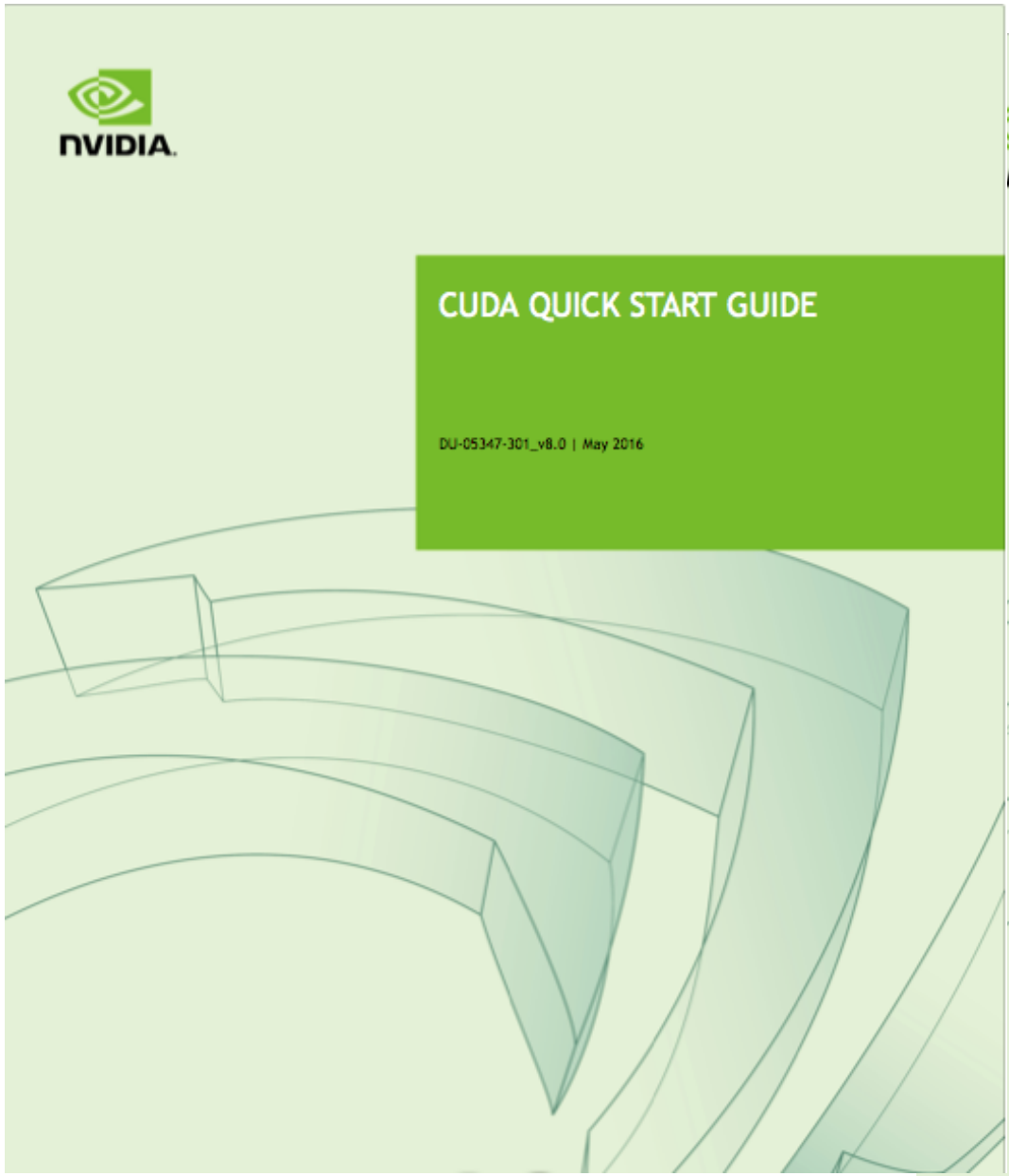


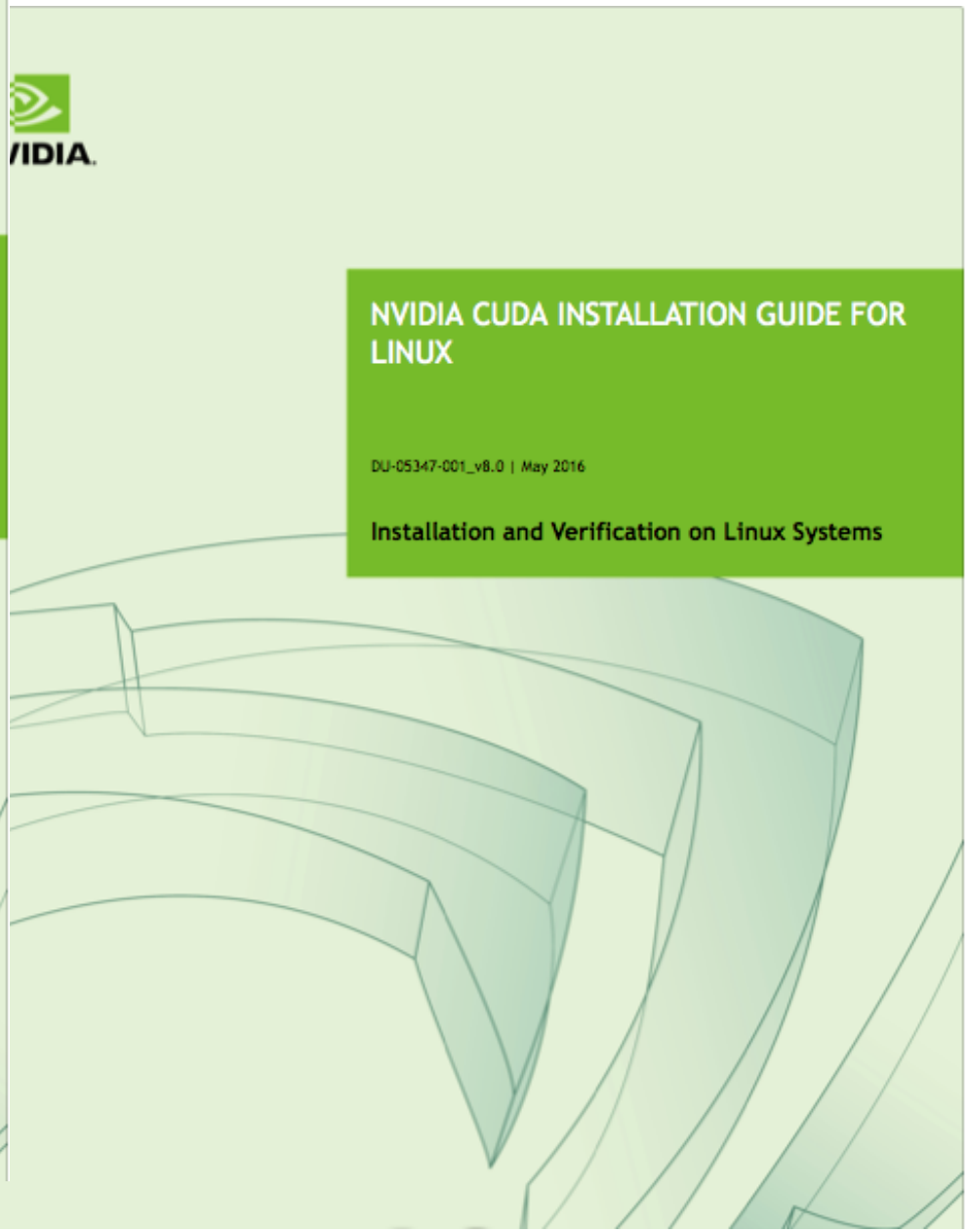
192 cores



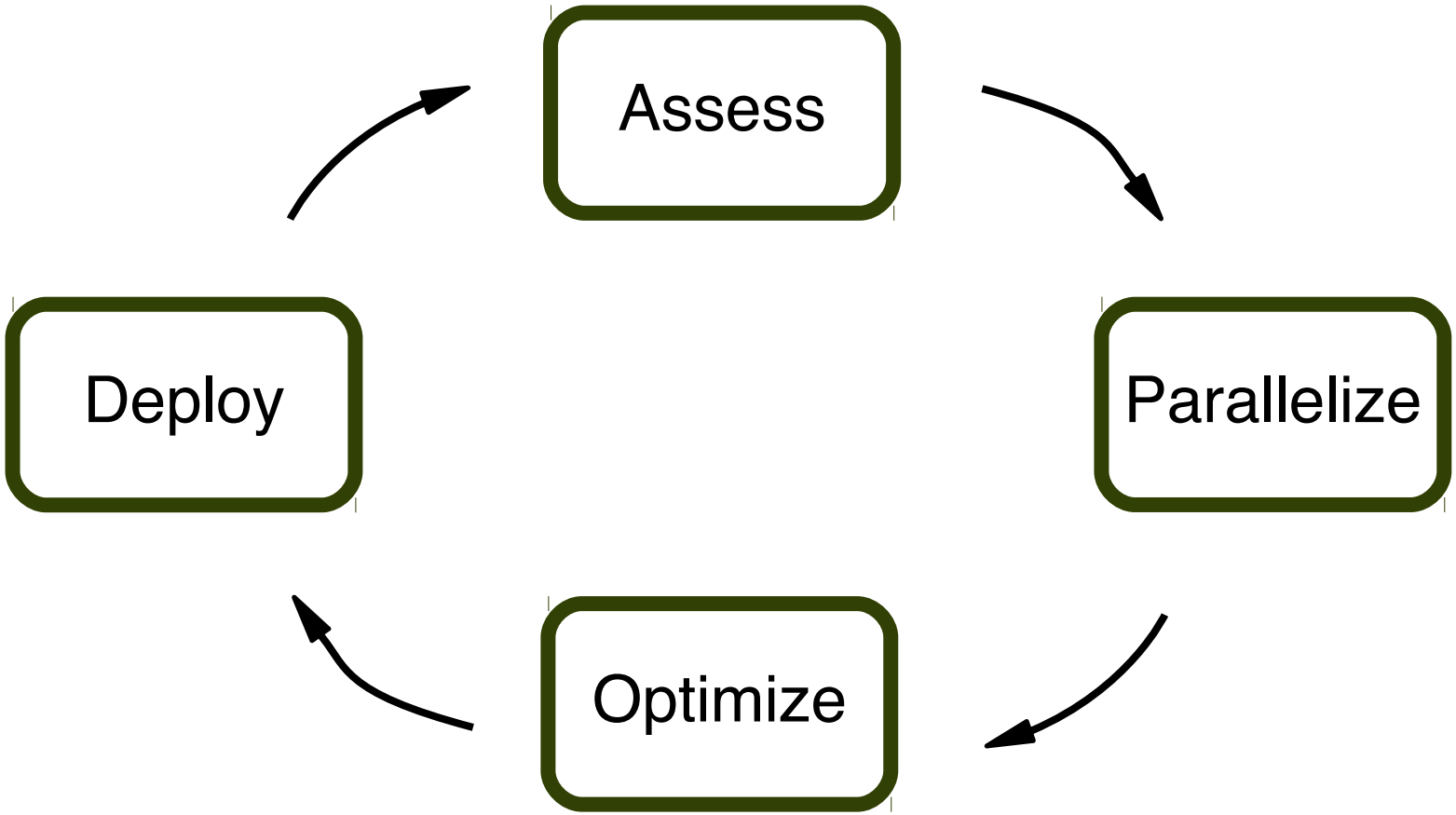
128 cores

Main Sources of CUDA Documentation



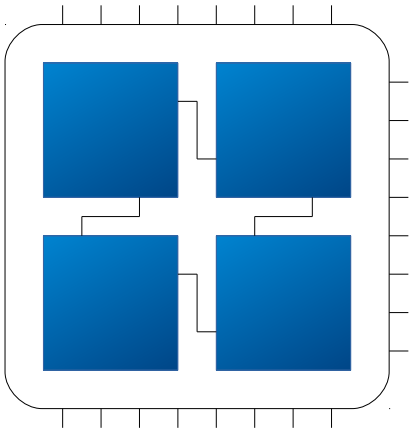


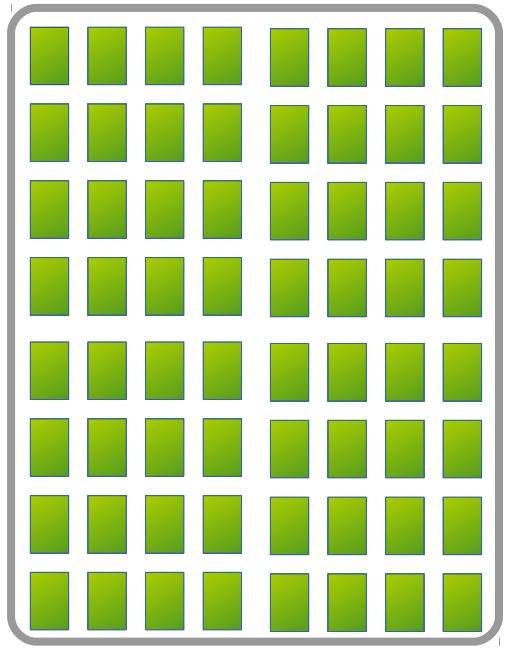
http://docs.nvidia.com/cuda/



- Trace
- Profile
- Disassemble

Software: CPU + GPU





Minimal 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, 3, 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, 3, 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

Introducing 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!
}

// sequential function (CPU)
int main(void) {
  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;
}
```

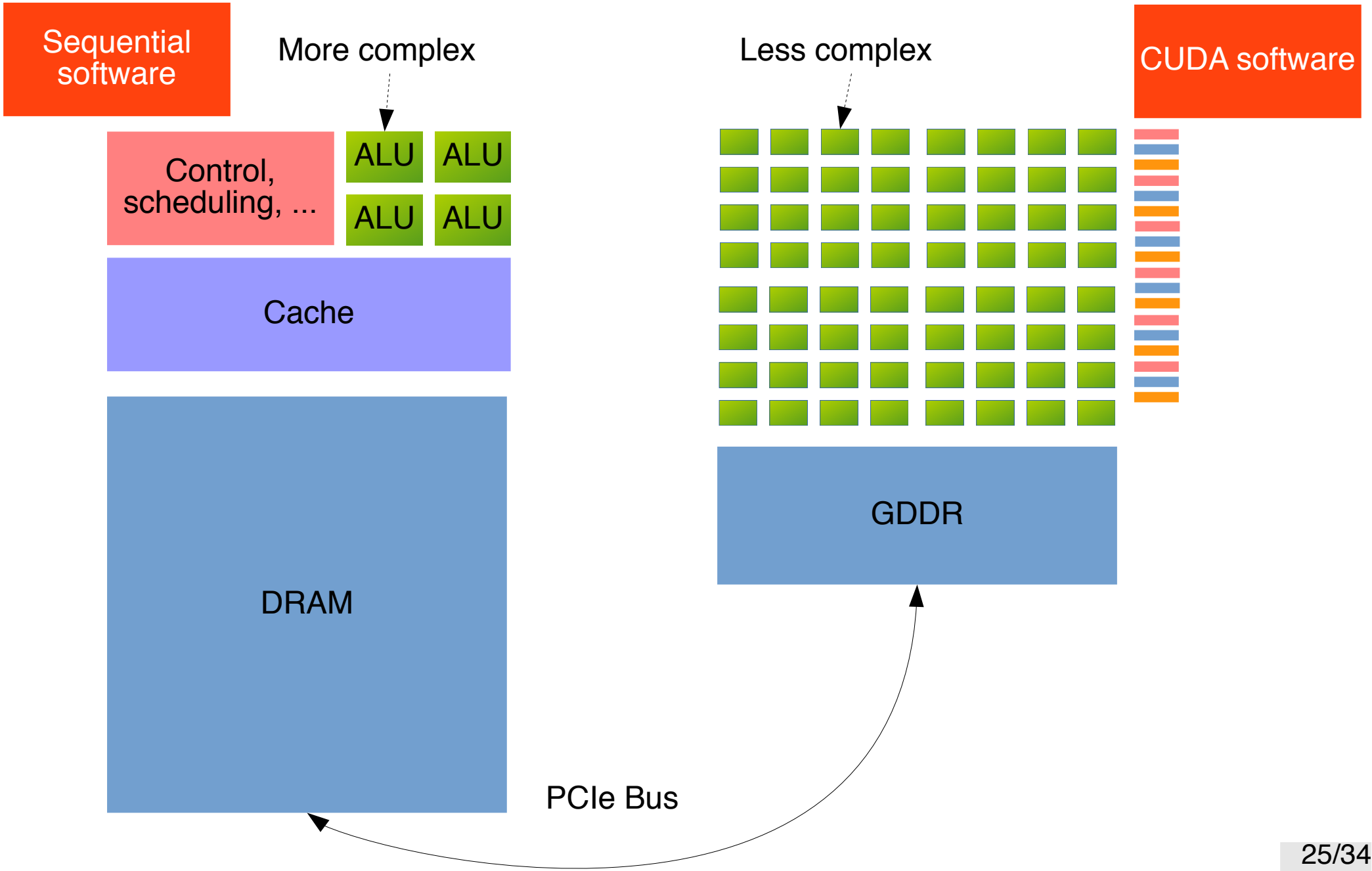
More on Block and Thread Parallelism

- 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

- 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 Goal: 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(...);
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(...);
```

Matrix Summation: GPU Launching (Faster!)

- 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

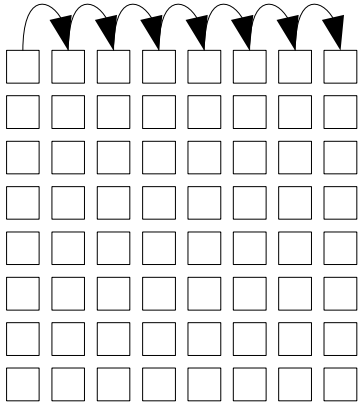
```
for (int j=0; j<n; ++j)
  int k = j*n;
  matrix_sum<<n,1>>(dA+k, dB+k, dC+k, n,1 );
```

- Kernel launch for all rows and columns at once

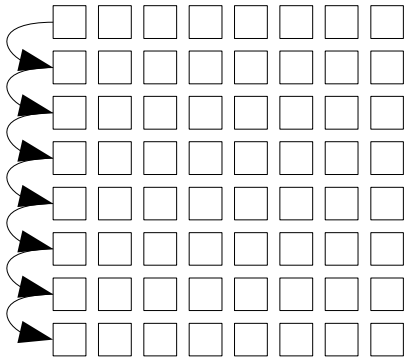
```
matrix_sum<<n,n>>(dA, dB, dC, n,n );
```

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

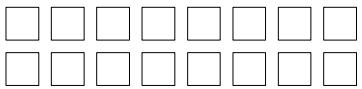
Ordering Matrix Elements in Memory



Row-major order in C/C++

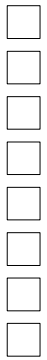


Column-major order in Fortran

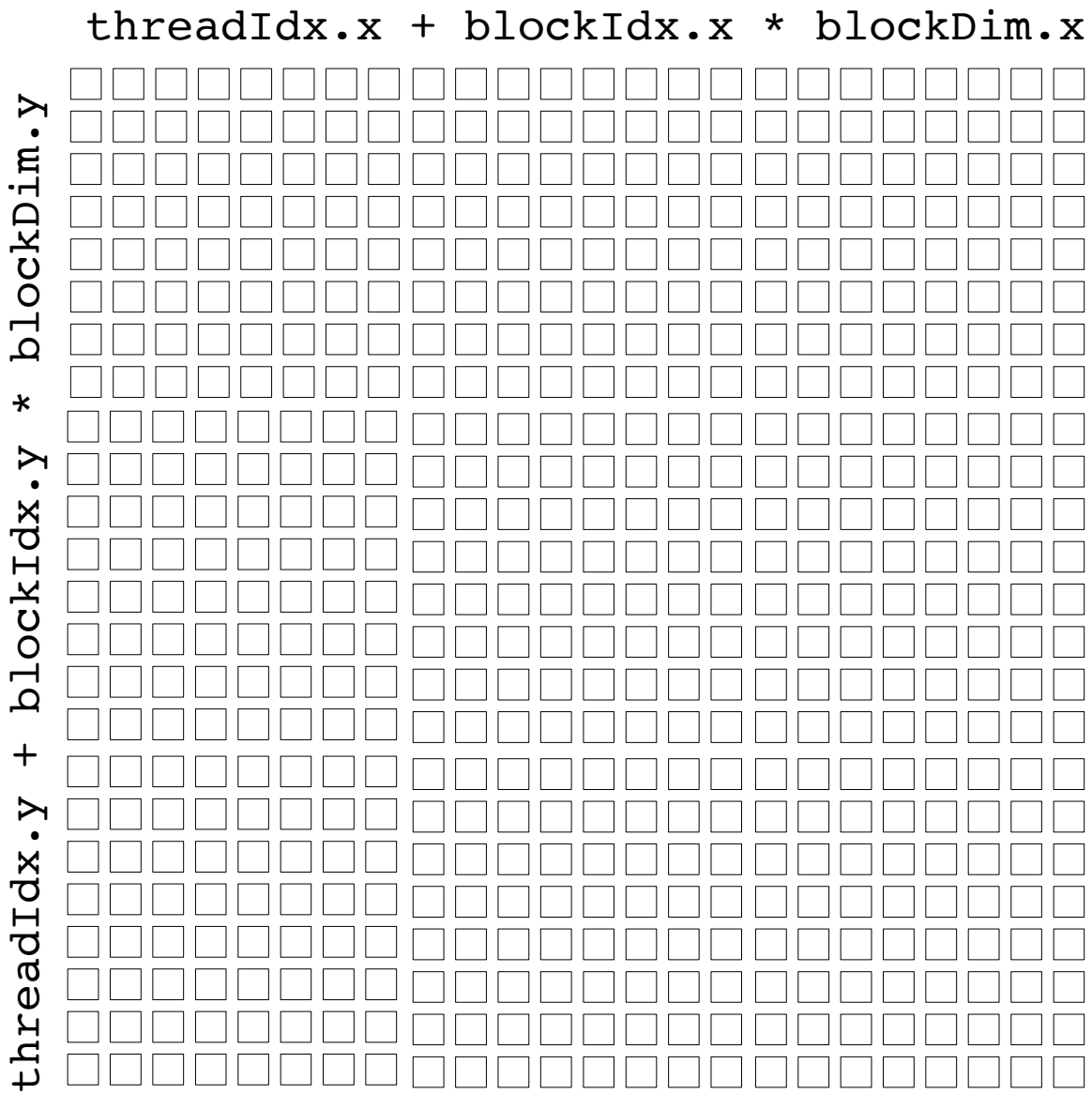


```
A + 0x0
A + 0x40
A + 0x80
A + 0xC0
A + 0x100
```

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



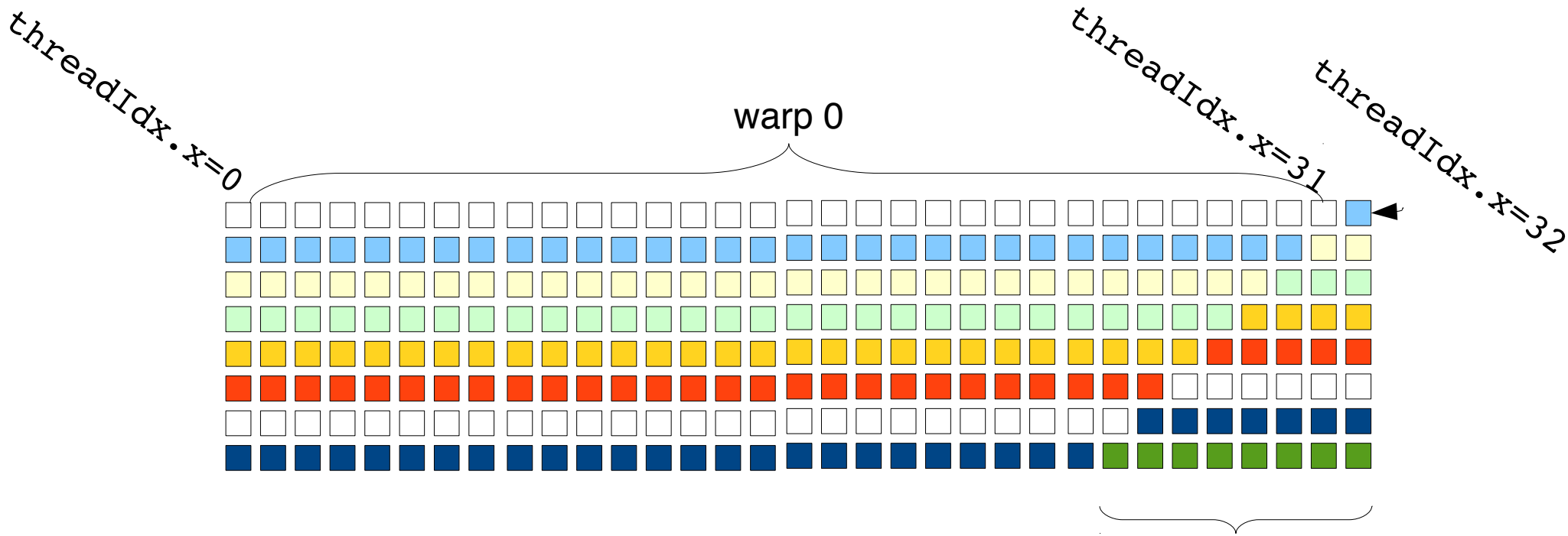
Mapping Threads to Matrix Elements



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.

GPU Warp Scheduling Basics

- GPU has at least one warp scheduler per SM
- 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) 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