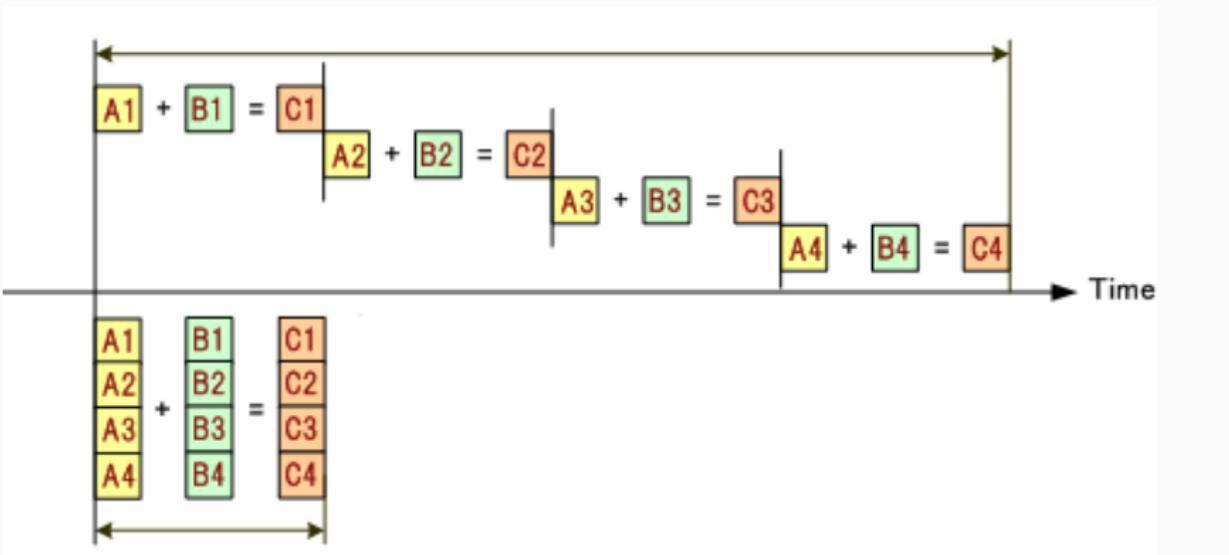


Introduction to Cuda

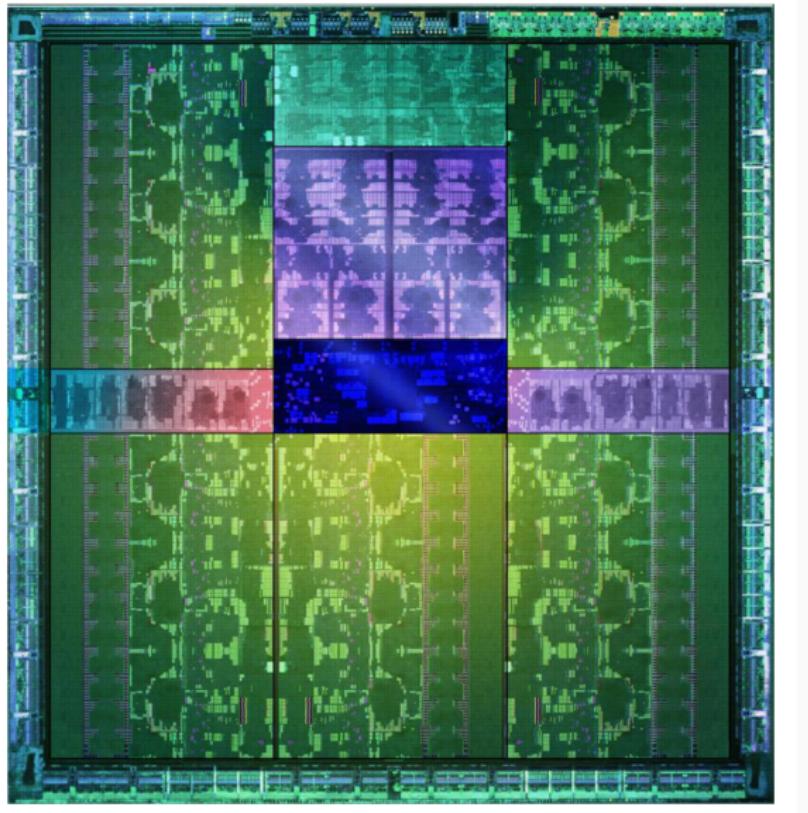
Jeremy Iverson

College of Saint Benedict & Saint John's University

execution model



gpu architecture

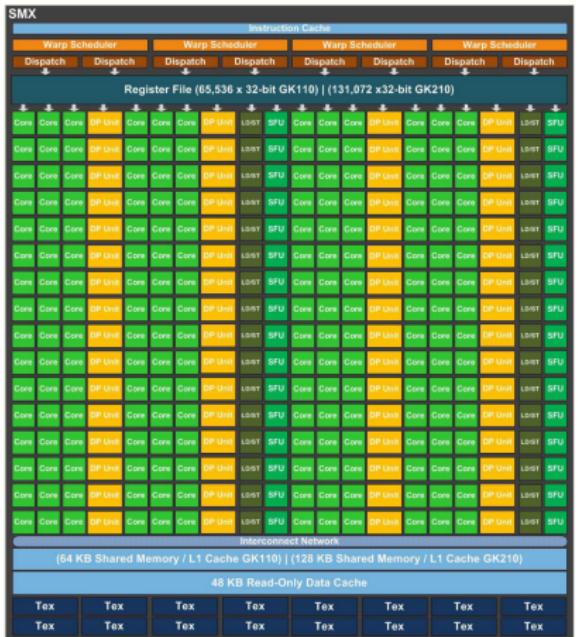


gpu architecture



k210 streaming multiprocessor (SMX)

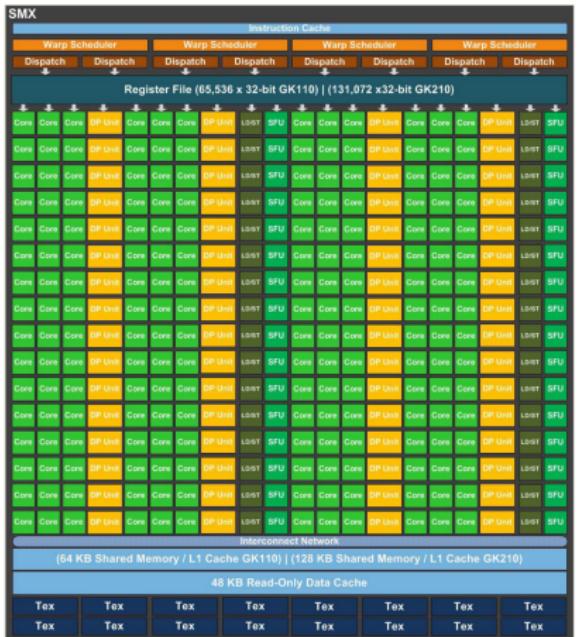
- 13 SMX per device
 - 192 SP CUDA cores
 - 64 DP CUDA cores
 - 128KB shared memory / L1 cache
 - 48KB read-only data cache
 - 128K 32-bit registers
- 1536KB L2 cache (shared)
- 12GB DRAM
- 562MHz base w/ 824MHz boost



1. $824,000,000 \times 2496 \times 2 = 4,113,408,000,000 = 4.113\text{TFlop/s}$ – the final $\times 2$ is because the GPU can issue a floating-point multiply and floating-point add as a single instruction (fused-multiply-add), so one instruction counting as two operations.

k210 streaming multiprocessor (SMX)

- 13 SMX per device
 - 192 SP CUDA cores
 - 64 DP CUDA cores
 - 128KB shared memory / L1 cache
 - 48KB read-only data cache
 - 128K 32-bit registers
- 1536KB L2 cache (shared)
- 12GB DRAM
- 562MHz base w/ 824MHz boost
- Theoretical 4.113TFLOP/s



1. $824,000,000 \times 2496 \times 2 = 4,113,408,000,000 = 4.113\text{TFlop/s}$ – the final $\times 2$ is because the GPU can issue a floating-point multiply and floating-point add as a single instruction (fused-multiply-add), so one instruction counting as two operations.

gpu execution model

- SIMD (single-instruction multiple-thread)
- SMX executes threads in groups of 32 threads, called warps
 - two independent instructions per warp can be issued per cycle
- warps are organized into blocks
 - a set of concurrently executing threads that can cooperate through synchronization and shared memory
- hardware multi-threading (as opposed to OS)
 - context switching is basically free

1. a result of dual issue is that 16 threads will be executing the same instruction... always
2. hardware multi-threading as opposed to OS, i.e., logic is built into card to support multi-threading, makes context switching basically free

memory scopes

- per thread private memory (allocated in register file)
- per block shared memory (allocated in shared memory)
 - threads in different blocks cannot access each others per block shared memory
- global memory (allocated in DRAM)
 - can be accessed by any thread, any time

CUDA

- Program GPU to complement CPU
- Augment C/C++ with minimalist abstractions
- Provide straightforward mapping onto hardware
 - good fit to GPU architecture
 - maps well to multi-core CPUs too
- Scale to 1000s of parallel threads
 - gpu cores are lightweight (create / switch is free)
 - gpu uses 1000s of threads to hide latency

- a parallel computation is initiated by executing a kernel function from the host (CPU)
 - kernel function runs to completion and returns control back to the host
- parallel threads are organized in a two-level hierarchy: threads and blocks
 - a computation contains a set of blocks, each block containing the same number of threads
 - threads in a block can communicate and synchronize
 - in general, threads in different blocks cannot synchronize

- blocks themselves are organized into a 3D grid – this is for convenience to the programmer – it can help map blocks to the parts of a problem that they are working
- blocks can be scheduled for execution on an SMX in any order
 - multiple blocks can be scheduled to the same SMX, as long as there are sufficient resources available
 - once scheduled, a block will run to completion

- Function qualifiers:
 - `__global__ void my_kernel()`
 - `__device__ float my_device_func()`
- Execution configuration:
 - `dim3 grid_dim(100, 50); // 5000 thread blocks`
 - `dim3 block_dim(4, 8, 8); // 256 threads per block`
 - `my_kernel<<<grid_dim,block_dim>>>(...); // Launch`
- Built-in variables and functions valid in device code:
 - `dim3 blockDim; // Grid dimension`
 - `dim3 blockDim; // Block dimension`
 - `dim3 blockIdx; // Block index`
 - `dim3 threadIdx; // Thread index`
 - `void __syncthreads(); // Thread synchronization`

vector addition

```
1 __global__ void
2 vector_add(float *A, float *B, float *C) {
3     int idx = blockIdx.x * blockDim.x + threadIdx.x;
4     C[idx] = A[idx] + B[idx];
5 }
6
7 int main() {
8     // Initialization code
9     ...
10    // Run N/256 blocks of 256 threads each
11    vector_add<<<N/256,256>>>(d_A, d_B, d_C);
12 }
```

vector addition cont'd

```
1 // allocate host (CPU) memory
2 float *h_A = ..., *h_B = ...;
3
4 // allocate device (GPU) memory
5 float *d_A, *d_B, *d_C;
6 cudaMalloc( (void**) &d_A, N * sizeof(float));
7 cudaMalloc( (void**) &d_B, N * sizeof(float));
8 cudaMalloc( (void**) &d_C, N * sizeof(float));
9
10 // populate h_A and h_B
11 ...
12
13 // copy host memory to device
14 cudaMemcpy(d_A, h_A, N * sizeof(float),
15            cudaMemcpyHostToDevice);
16 cudaMemcpy(d_B, h_B, N * sizeof(float),
17            cudaMemcpyHostToDevice);
18 }
```



except where otherwise noted, this worked is licensed under creative commons attribution-sharealike 4.0 international license