Fundamental CUDA Optimization

NVIDIA Corporation

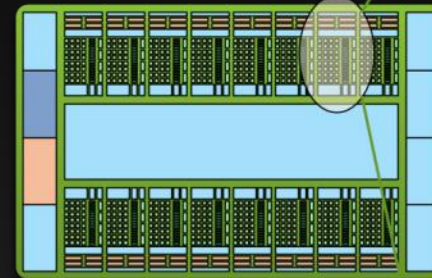
Outline

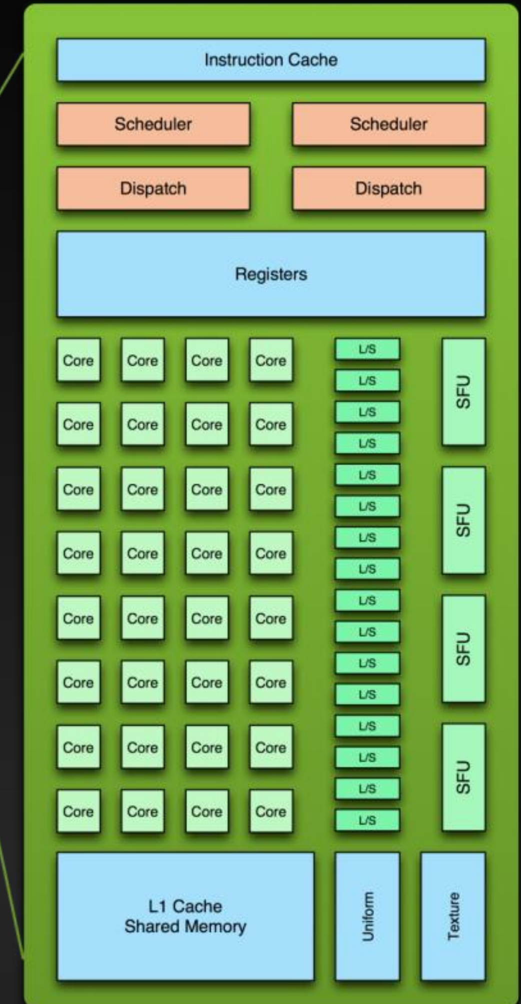
- Fermi Architecture
- Kernel optimizations
  - Launch configuration
  - Global memory throughput
  - Shared memory access
  - Instruction throughput / control flow
- Optimization of CPU-GPU interaction
  - Maximizing PCIe throughput
  - Overlapping kernel execution with memory copies

Most concepts in this presentation apply to *any* language or API on NVIDIA GPUs

20-Series Architecture (Fermi)

- 512 *Scalar Processor (SP) cores* execute parallel thread instructions
- 16 Streaming *Multiprocessors (SMs)* each contains
  - 32 scalar processors
    - 32 fp32 / int32 ops / clock,
    - 16 fp64 ops / clock
  - 4 Special Function Units (SFUs)
  - Shared register file (128KB)
  - *48 KB / 16 KB Shared memory*
  - 16KB / 48 KB L1 data cache





Kepler cc 3.5 SM (GK110)

- “SMX” (enhanced SM)
- 192 SP units (“cores”)
- 64 DP units
- LD/ST units
- 4 warp schedulers
- Each warp scheduler is dual-issue capable
- K20: 13 SMX’s, 5GB
- K20X: 14 SMX’s, 6GB
- K40: 15 SMX’s, 12GB

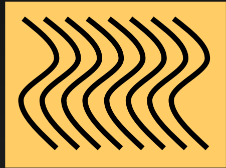
Execution Model

Software **Hardware**

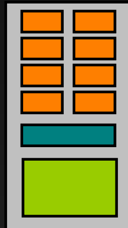
Thread

Scalar Processor

Threads are executed by scalar processors



Thread Block

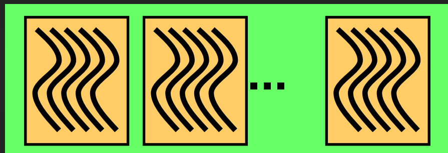


Multiprocessor

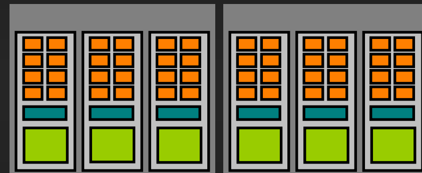
Thread blocks are executed on multiprocessors

Thread blocks do not migrate

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



Grid



Device

A kernel is launched as a grid of thread blocks

Warps

A thread block consists of 32-thread warps

A warp is executed physically in parallel (SIMD) on a multiprocessor

Memory Architecture

CPU-GPU Interaction

Heterogeneous Computing

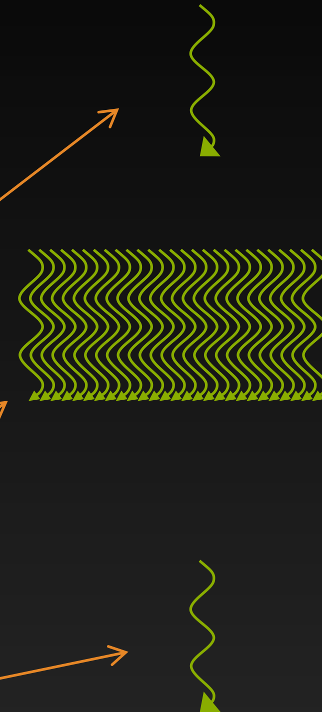
```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__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();

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

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

void fill_ints(int *x, int n) {
        fill_n(x, n, 1);
}

int main(void) {
        int *in, *out;            // host copies of a, b, c
        int *d_in, *d_out;        // device copies of a, b, c
        int size = (N + 2*RADIUS) * sizeof(int);

        // Alloc space for host copies and setup values
        in  = (int *)malloc(size); fill_ints(in,  N + 2*RADIUS);
        out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

        // Alloc space for device copies
        cudaMalloc((void **)&d_in,  size);
        cudaMalloc((void **)&d_out, size);

        // Copy to device
        cudaMemcpy(d_in,  in,  size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

        // Launch stencil_1d() kernel on GPU
        stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);

        // Copy result back to host
        cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

        // Cleanup
        free(in); free(out);
        cudaFree(d_in); cudaFree(d_out);
        return 0;
}
```

parallel fn

serial code

parallel code

serial code



Simple Processing Flow

CPU
Bridge
CPU Memory
PCI Bus
GigaThread™
Interconnect
L2
DRAM

1. Copy input data from CPU memory to GPU memory

Simple Processing Flow

CPU
Bridge
CPU Memory
PCI Bus
GigaThread™
DRAM

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

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

Questions?