Part 4B: GPU programming with CUDA
GPU programming with CUDA

• We will use NVIDIA GPUs and CUDA programming environment
  • CUDA code ≈ C++ code with some extensions
  • compile with nvcc, run as usual

• Just compiling your code with nvcc doesn’t do anything yet
  • *your* `main()` *function still runs on the CPU!*

• In your program, you need to specify what the GPU should do
  • you define a so-called “kernel” function
  • *you explicitly ask the GPU to run the “kernel” with many threads!*
CUDA basics

What we would like to parallelize

for (int i = 0; i < 100; ++i) {
    for (int j = 0; j < 128; ++j) {
        foo(i, j);
    }
}
CUDA basics

Parallel GPU solution

```c
__global__ void mykernel() {
    int i = blockIdx.x;
    int j = threadIdx.x;
    foo(i, j);
}

int main() {
    mykernel<<<100, 128>>>()
}
```

GPU will run these operations, possibly in parallel:
- `foo(0, 0)`
- `foo(0, 1)`
- `foo(0, 2)`
- `foo(0, 3)`
  ...
- `foo(99, 127)`

Which thread am I?

Create 100 blocks, each with 128 threads, and let them all run function mykernel!
CUDA basics

Parallel GPU solution

```c
__global__ void mykernel() {
    int i = blockIdx.x;
    int j = threadIdx.x;
    foo(i, j);
}

int main() {
    mykernel<<<100, 128>>>()
}
```

Equivalent sequential code

```c
int main() {
    for (int i = 0; i < 100; ++i) {
        for (int j = 0; j < 128; ++j) {
            foo(i, j);
        }
    }
}
```
Example: split evenly

• What is the best way to split $1^5, 2^5, 3^5, \ldots, 30^5$ in two parts such that their sums are as close to each other as possible?

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$1^5$</td>
<td>$2^5$</td>
<td>$3^5$</td>
<td>$4^5$</td>
<td>$5^5$</td>
<td>$6^5$</td>
<td>$7^5$</td>
<td>$8^5$</td>
<td>$9^5$</td>
<td>$10^5$</td>
</tr>
<tr>
<td>$11^5$</td>
<td>$12^5$</td>
<td>$13^5$</td>
<td>$14^5$</td>
<td>$15^5$</td>
<td>$16^5$</td>
<td>$17^5$</td>
<td>$18^5$</td>
<td>$19^5$</td>
<td>$20^5$</td>
</tr>
<tr>
<td>$21^5$</td>
<td>$22^5$</td>
<td>$23^5$</td>
<td>$24^5$</td>
<td>$25^5$</td>
<td>$26^5$</td>
<td>$27^5$</td>
<td>$28^5$</td>
<td>$29^5$</td>
<td>$30^5$</td>
</tr>
</tbody>
</table>
Example: split evenly

• What is the best way to split \(1^5, 2^5, 3^5, \ldots, 30^5\) in two parts such that their sums are as close to each other as possible?

\[
\begin{array}{cccccc}
1^5 & 2^5 & 4^5 & 6^5 & 10^5 & \\
11^5 & 12^5 & 13^5 & 15^5 & 17^5 & 19^5 \\
21^5 & 22^5 & 23^5 & 24^5 & 27^5 & 30^5 \\
\end{array}
\]

sum: 67,830,947

\[
\begin{array}{cccccc}
3^5 & 5^5 & 7^5 & 8^5 & 9^5 \\
14^5 & 16^5 & 18^5 & 20^5 \\
25^5 & 26^5 & 28^5 & 29^5 \\
\end{array}
\]

sum: 66,156,478
Example: split evenly

- What is the best way to **split** $1^5, 2^5, 3^5, \ldots, 30^5$ in two parts such that their **sums** are as close to each other as possible?

This clearly isn’t optimal — how to find the best solution?
Example: split evenly

• What is the best way to split $1^5, 2^5, 3^5, \ldots, 30^5$ in two parts such that their sums are as close to each other as possible?

• We will solve this with a naive brute force algorithm

• First with CPUs with a sequential program

• Then with GPUs with a massively parallel program
Example: split evenly

• What is the best way to split $1, 5, 2, 5, 3, 5, \ldots, 30, 5$ in two parts such that their sums are as close to each other as possible?

• Algorithms: just try out all $2^{30}$ cases and see what is best
Example: split evenly

Each case is represented as a 30-bit binary number $x$

Bit 0 in position $i$: number $(i + 1)^5$ in the \textbf{first part}

$$x = 1 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0$$

Bit 1 in position $i$: number $(i + 1)^5$ in the \textbf{second part}
inline int p5(int i) { return i * i * i * i * i; }

inline int value(int x) {
    int a = 0;
    for (int i = 0; i < 30; ++i) {
        if (x & (1 << i)) {
            a += p5(i+1);
        } else {
            a -= p5(i+1);
        }
    }
    return abs(a);
}

x = one way to split our numbers
value(x) = absolute difference between the sum of the first part and the sum of the second part
inline int p5(int i) { return i * i * i * i * i; }

inline int value(int x) {
    int a = 0;
    for (int i = 0; i < 30; ++i) {
        if (x & (1 << i)) {
            a += p5(i+1);
        } else {
            a -= p5(i+1);
        }
    }
    return abs(a);
}
Sequential CPU solution

```c
constexpr int total = 1 << 30;
int best_x = 0;
int best_v = value(best_x);
for (int x = 0; x < total; ++x) {
    int v = value(x);
    if (v < best_v) {
        best_x = x;
        best_v = v;
    }
}
```

Find $0 \leq x < 2^{30}$ that minimizes $\text{value}(x)$
GPU: splitting work

• We have got $2^{30}$ cases to check
• How many blocks to create?
• How many threads per block?
• How many cases does one thread check?
GPU: splitting work

- We have got $2^{30}$ cases to check
- How many *blocks* to create?
- How many *threads* per block?
- If we have e.g. $2^{30}$ threads in total, each thread will only check 1 case
  - too little useful work per thread
  - too much overhead e.g. in launching kernel, communicating result
GPU: splitting work

- We have got $2^{30}$ cases to check

**Blocks:**
- we need to have lots of blocks ready for execution
- our choice here: $2^{10} = 1024$ blocks

**Threads per block:**
- reasonable block size is a multiple of one warp = 32 threads
- our choice here: $2^6 = 64$ threads

- Each thread will need to check $2^{30} / (2^{10} \cdot 2^6) = 2^{14}$ cases
GPU: splitting work

- **Block index** (highest 10 bits)

- **Thread index** (next 6 bits)

- **Current iteration inside one thread** (lowest 14 bits)
Let’s keep things as simple as possible

Allocate one word of GPU memory per thread

**GPU: each thread will write its local optimum in GPU memory**

Copy results from GPU memory to CPU memory

**CPU: find the best split among local optima**
```c
__global__ void mykernel(int* r) {
    int x3 = blockIdx.x;
    int x2 = threadIdx.x;

    int best_x = 0;
    int best_v = value(best_x);

    for (int x1 = 0; x1 < iterations; ++x1) {
        int x = (x3 << 20) | (x2 << 14) | x1;
        int v = value(x);
        if (v < best_v) {
            best_x = x;
            best_v = v;
        }
    }

    r[(x3 << 6) | x2] = best_x;
}
```

What is my part of search space?

**x3**
(10 bits)

**x2**
(6 bits)

**x1**
(14 bits)

\[
x = 1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0
\]
```c
__global__ void mykernel(int* r) {
    int x3 = blockIdx.x;
    int x2 = threadIdx.x;
    int best_x = 0;
    int best_v = value(best_x);
    for (int x1 = 0; x1 < iterations; ++x1) {
        int x = (x3 << 20) | (x2 << 14) | x1;
        int v = value(x);
        if (v < best_v) {
            best_x = x;
            best_v = v;
        }
    }
    r[(x3 << 6) | x2] = best_x;
}
```
__global__ void mykernel(int* r) {
    int x3 = blockIdx.x;
    int x2 = threadIdx.x;
    int best_x = 0;
    int best_v = value(best_x);
    for (int x1 = 0; x1 < iterations; ++x1) {
        int x = (x3 << 20) | (x2 << 14) | x1;
        int v = value(x);
        if (v < best_v) {
            best_x = x;
            best_v = v;
        }
    }
    r[(x3 << 6) | x2] = best_x;
}
constexpr int blocks = 1 << 10;
constexpr int threads = 1 << 6;

int* rGPU = NULL;
cudaMalloc((void**)&rGPU, blocks * threads * sizeof(int));

mykernel<<<blocks, threads>>>(rGPU);

std::vector<int> r(blocks * threads);
cudaMemcpy(r.data(), rGPU, blocks * threads * sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(rGPU);

// Find x in r that minimizes value(x)
constexpr int blocks = 1 << 10;
constexpr int threads = 1 << 6;

int* rGPU = NULL;
cudaMalloc((void**)&rGPU, blocks * threads * sizeof(int));

mykernel<<<blocks, threads>>>(rGPU);

std::vector<int> r(blocks * threads);
cudaMemcpy(r.data(), rGPU, blocks * threads * sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(rGPU);

// Find x in r that
// minimizes value(x)
constexpr int blocks = 1 << 10;
constexpr int threads = 1 << 6;

int* rGPU = NULL;
cudaMalloc((void**)&rGPU, blocks * threads * sizeof(int));

mykernel<<<blocks, threads>>>(rGPU);

std::vector<int> r(blocks * threads);
cudaMemcpy(r.data(), rGPU, blocks * threads * sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(rGPU);

// Find x in r that
// minimizes value(x)
constexpr int blocks = 1 << 10;
constexpr int threads = 1 << 6;

int* rGPU = NULL;
cudaMalloc((void**)&rGPU, blocks * threads * sizeof(int));

mykernel<<<blocks, threads>>>(rGPU);

std::vector<int> r(blocks * threads);
cudaMemcpy(r.data(), rGPU, blocks * threads * sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(rGPU);

// Find x in r that // minimizes value(x)
```cpp
constexpr int blocks = 1 << 10;
constexpr int threads = 1 << 6;

int* rGPU = NULL;
cudaMalloc((void**)&rGPU, blocks * threads * sizeof(int));

mykernel<<<blocks, threads>>>(rGPU);

std::vector<int> r(blocks * threads);
cudaMemcpy(r.data(), rGPU, blocks * threads * sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(rGPU);

// Find x in r that // minimizes value(x)
```

```c++
constexpr int blocks = 1 << 10;
constexpr int threads = 1 << 6;

int* rGPU = NULL;
cudaMalloc((void**)&rGPU, blocks * threads * sizeof(int));

mykernel<<<blocks, threads>>>(rGPU);

std::vector<int> r(blocks * threads);
cudaMemcpy(r.data(), rGPU, blocks * threads * sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(rGPU);

// Find x in r that
// minimizes value(x)
```

Now vector “r” contains the best result for each thread, just check which of these is the global optimum
Try it out

• Compile & link with “nvcc” instead of “g++”

• Run as usual  
  • sequential CPU solution: 38 seconds  
  • parallel GPU solution: 0.3 seconds

\[
1^5 + 2^5 + 3^5 + 4^5 + 5^5 + 9^5 + 10^5 + 12^5 + 15^5 + 16^5 + 17^5 + 19^5 + 22^5 + 23^5 + 24^5 + 25^5 + 27^5 + 28^5 = 66\,993\,712
\]

\[
6^5 + 7^5 + 8^5 + 11^5 + 13^5 + 14^5 + 18^5 + 20^5 + 21^5 + 26^5 + 29^5 + 30^5 = 66\,993\,713
\]
constexpr int blocks = 1 << 10;
constexpr int threads = 1 << 6;

int* rGPU = NULL;
cudaMalloc((void**)&rGPU, blocks * threads * sizeof(int));

mykernel<<<blocks, threads>>>(rGPU);

std::vector<int> r(blocks * threads);
cudaMemcpy(r.data(), rGPU, blocks * threads * sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(rGPU);

// Find x in r that minimizes value(x)
constexpr int blocks = 1 << 10;
constexpr int threads = 1 << 6;

int* rGPU = NULL;
CHECK(cudaMalloc((void**)&rGPU, blocks * threads * sizeof(int)));

mykernel<<<blocks, threads>>>(rGPU);
CHECK(cudaGetLastError());

std::vector<int> r(blocks * threads);
CHECK(cudaMemcpy(r.data(), rGPU, blocks * threads * sizeof(int), cudaMemcpyDeviceToHost));
CHECK(cudaFree(rGPU));

// Find x in r that
// minimizes value(x)
Typical program structure

• **GPU side:**
  • “**kernel**” that does one small part of work

• **CPU side:**
  • do pre-processing if needed
  • allocate GPU memory for input & output
  • copy input from CPU memory to GPU memory
  • **launch kernel** (lots of blocks, lots of threads)
  • copy result back from GPU memory to CPU memory
  • release GPU memory
  • do post-processing if needed