Simulation and High-Performance Computing

Part 15: Introduction to GPU Computing

Steffen Börm

Christian-Albrechts-Universität zu Kiel

October 7th, 2020

Computing with graphics cards

Idea: Graphics processors (GPUs) frequently offer far more computing power than general-purpose processors (CPUs).
If we can make use of GPUs, our programs may run significantly faster.

Challenges:

- Graphics processors are typically vector processors, i.e., they are very good a performing identical operations on large blocks of data and have trouble with branches.
- Graphics processors are intended for fairly small programs.
- Access to graphics memory is typically faster than for a CPU, but there is far less memory available.
- In order to use graphics cards for computation, we have to explicitly move data from and to graphics memory and coordinate parts of the program running on the CPU and the GPU.

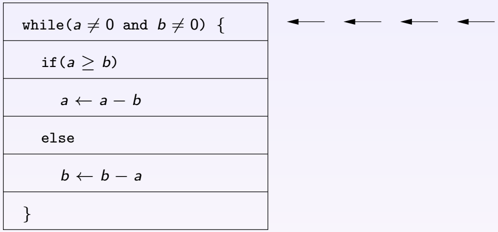
SIMT: Single Instruction, Multiple Threads

Friendly vectorization: Compared to using intrinsics to implement AVX code, graphics processors use a far more user-friendly approach:

- The program is split into threads. These threads essentially are represented only by an instruction pointer and a few registers for local variables.
- Threads are assigned to multiprocessors.
- Every cycle, the multiprocessor chooses a set of threads and an instruction of these threads' programs.
  This instruction is executed by all threads that need it.

Example: Euclid's algorithm

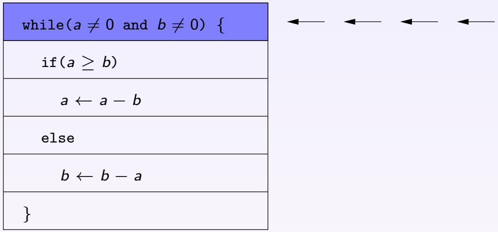
Approach: Apply Euclid's algorithm to four different starting values in four different threads.



```
while($a \neq 0$ and $b \neq 0$) {
  if($a \geq b$)
    $a \leftarrow a - b$
  else
    $b \leftarrow b - a$
}
```

Example: Euclid's algorithm

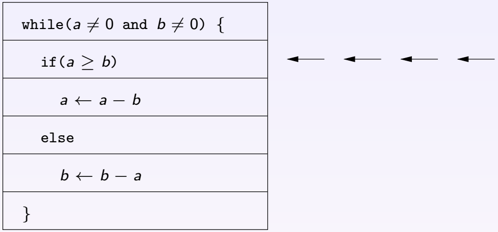
Approach: Apply Euclid's algorithm to four different starting values in four different threads.



```
while($a \neq 0$ and $b \neq 0$) {
  if($a \geq b$)
    $a \leftarrow a - b$
  else
    $b \leftarrow b - a$
}
```

Example: Euclid's algorithm

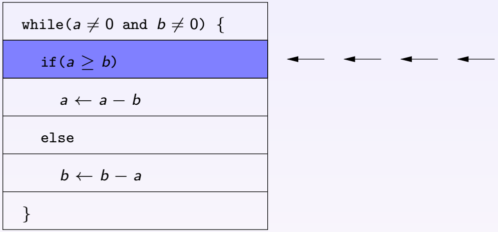
Approach: Apply Euclid's algorithm to four different starting values in four different threads.



```
while($a \neq 0$ and $b \neq 0$) {
  if($a \geq b$)
    $a \leftarrow a - b$
  else
    $b \leftarrow b - a$
}
```

Example: Euclid's algorithm

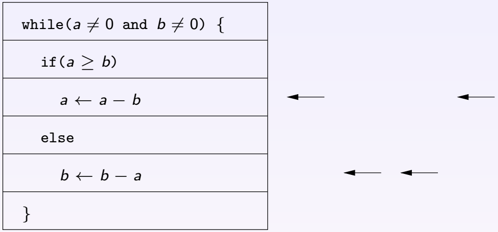
Approach: Apply Euclid's algorithm to four different starting values in four different threads.



```
while($a \neq 0$ and $b \neq 0$) {
  if($a \geq b$)
    $a \leftarrow a - b$
  else
    $b \leftarrow b - a$
}
```

Example: Euclid's algorithm

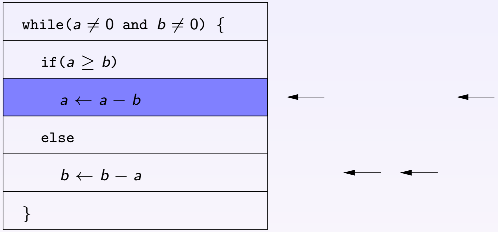
Approach: Apply Euclid's algorithm to four different starting values in four different threads.



```
while($a \neq 0$ and $b \neq 0$) {
  if($a \geq b$)
    $a \leftarrow a - b$
  else
    $b \leftarrow b - a$
}
```

Example: Euclid's algorithm

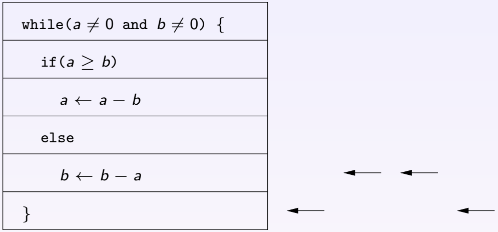
Approach: Apply Euclid's algorithm to four different starting values in four different threads.



```
while($a \neq 0$ and $b \neq 0$) {
  if($a \geq b$)
    $a \leftarrow a - b$
  else
    $b \leftarrow b - a$
}
```

Example: Euclid's algorithm

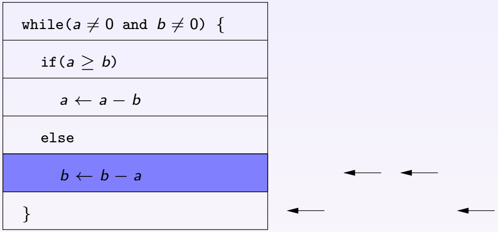
Approach: Apply Euclid's algorithm to four different starting values in four different threads.



```
while($a \neq 0$ and $b \neq 0$) {
  if($a \geq b$)
    $a \leftarrow a - b$
  else
    $b \leftarrow b - a$
}
```

Example: Euclid's algorithm

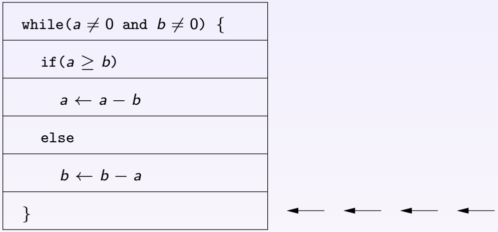
Approach: Apply Euclid's algorithm to four different starting values in four different threads.



```
while($a \neq 0$ and $b \neq 0$) {
  if($a \geq b$)
    $a \leftarrow a - b$
  else
    $b \leftarrow b - a$
}
```

Example: Euclid's algorithm

Approach: Apply Euclid's algorithm to four different starting values in four different threads.



```
while($a \neq 0$ and $b \neq 0$) {
  if($a \geq b$)
    $a \leftarrow a - b$
  else
    $b \leftarrow b - a$
}
```

Observation: Although the threads may diverge, the processor has a chance to restore unity.

CUDA: Compute Unified Device Architecture

Approach: Extension of C, adding code generation for graphics processors and functions for synchronization and data transfer between graphics card and main processor.

Advantages:

- Easy to learn.
- Very flexible.
- Active community.

Disadvantages:

- Owned by NVIDIA, supported exclusively by NVIDIA hardware.
- CUDA software develoment kit not available on all platforms.

CUDA devices

Host: Standard computer, runs an operating system, sets up the graphics card, and starts CUDA threads.

CUDA device: Consists of CUDA multiprocessors, graphics memory, and is connected to the host, e.g., via a PCIe bus.

Symmetric multiprocessor: Consists of CUDA cores, a thread scheduler, shared memory, and a register file.

CUDA cores: Carry out instructions of CUDA threads.

CUDA threads

Compared to OpenMP threads, CUDA threads are very simple, in order to allow very efficient thread execution.

- The variables and the instruction pointer are assigned to registers. Each multiprocessor offers only a limited number of registers.
- There is no stack, therefore function calls are limited.
- Threads are organized in blocks.
  All threads in a block run on the same multiprocessor, share its resources, and can synchronize via barriers.

Warps: For scheduling purposes, thread blocks are split into warps of 32 threads or less.
Every cycle, one instruction of one warp is executed.

Example: Gravitation

Goal: Evaluate gravitational potentials

$$\varphi_i = \sum_{\substack{j=1\\ j\neq i}}^{n} \frac{m_j}{\|x_j - x_i\|}.$$

```
__global__ void
grav(int n, const float *x1, const float *x2,
     const float *x3, const float *m,
     float *phi)
{
  float sum, d1, d2, d3, norm2;
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j;
  if(i < n) {
    /* compute phi[i] */
  }
}
```

Example: Gravitation

```
__global__ void
grav(int n, /* ... */)
{
  /* ... */
  if(i < n) {
    sum = 0.0;
    for(j=0; j<n; j++) {
      d1 = x1[j] - x1[i];
      d2 = x2[j] - x2[i];
      d3 = x3[j] - x3[i];
      norm2 = d1 * d1 + d2 * d2 + d3 * d3;
      if(j != i)
        sum += m[j] / sqrtf(norm2);
    }
    phi[i] = sum;
  }
}
```

Example: Gravitation

Pointers refer to device memory.

```
__global__ void
grav(int n, const float *x1, const float *x2, ...)
```

Example: Gravitation

Pointers refer to device memory.

```
__global__ void
grav(int n, const float *x1, const float *x2, ...)
```

Global thread index can be constructed from the block index, the size of the blocks, and the local thread index within the block.

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

Example: Gravitation

Pointers refer to device memory.

```
__global__ void
grav(int n, const float *x1, const float *x2, ...)
```

Global thread index can be constructed from the block index, the size of the blocks, and the local thread index within the block.

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

Idle threads: We may have more threads than array elements.

```
if(i < n) {
  ...
}
```

Example: Gravitation

Pointers refer to device memory.

```
__global__ void
grav(int n, const float *x1, const float *x2, ...)
```

Global thread index can be constructed from the block index, the size of the blocks, and the local thread index within the block.

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```

Idle threads: We may have more threads than array elements.

```
if(i < n) {
  ...
}
```

Branches are handled automatically by SIMT.

```
if(j != i)
  sum += m[j] / sqrtf(norm2);
```

Data transfer

Device memory has to be allocated by the host.

```
cudaMalloc(&dev_y1, sizeof(float) * n);
cudaMalloc(&dev_y2, sizeof(float) * n);
...
```

Data has to be transferred between host and device.

```
cudaMemcpy(dev_y1, y1, sizeof(float) * n,
           cudaMemcpyHostToDevice);
cudaMemcpy(dev_y2, y2, sizeof(float) * n,
           cudaMemcpyHostToDevice);
...

cudaMemcpy(phi, dev_phi, sizeof(float) * n,
           cudaMemcpyDeviceToHost);
```

In a standard PC, data is transferred via the comparatively slow PCIe bus (e.g., 64 GiB/s for PCIe 4.0 vs 936 GiB/s for graphics memory)

Kernel execution

Kernels are small programs that run on the GPU. In CUDA C programs, they are marked by the keyword `__global__`:

```
__global__ void
grav(int n, const float *x1, const float *x2,
     const float *x3, const float *m,
     float *phi)
{ ... }
```

We can start these programs from the host:

```
grav<<<(n+255)/256, 256>>>(n, dev_y1, dev_y2, dev_y3,
                           dev_m, dev_phi);
```

Here 256 is the block size, and `(n+255)/256` is the number of blocks. If n is not a multiple of 256, we round up.

Concurrency: The host does not wait for the kernel to complete, so the CPU can work in parallel with the GPU.

Experiment: Gravitation with CUDA

Goal: Evaluate gravitational potentials

$$\varphi_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{m_j}{\|x_j - x_i\|}.$$

Competition: Ryzen 7 3700X vs GeForce RTX 2060 SUPER

n	CPU	AVX/1	AVX/8	GPU
16 384	442	52	14	6
32 768	1 711	210	35	13
65 536	6 989	864	128	53
131 072	28 113	3 474	511	201

Experiment: Gravitation with CUDA

Goal: Evaluate gravitational potentials

$$\varphi_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} \frac{m_j}{\|x_j - x_i\|}.$$

Competition: Ryzen 7 3700X vs GeForce RTX 2060 SUPER

n	CPU	AVX/1	AVX/8	GPU
16 384	442	52	14	6
32 768	1 711	210	35	13
65 536	6 989	864	128	53
131 072	28 113	3 474	511	201

Result: GPU more than twice as fast as CPU.

Error handling and timing

Error handling: Most CUDA functions return an error code of type `cudaError_t`.

- If the error code equals `cudaSuccess`, the function was successful.
- Otherwise the function `cudaGetErrorString` can be used to obtain a readable error description.
- Starting a kernel does not return an error code, but we can use the function `cudaGetLastError` to obtain the code.

Error handling and timing

Error handling: Most CUDA functions return an error code of type `cudaError_t`.

- If the error code equals `cudaSuccess`, the function was successful.
- Otherwise the function `cudaGetErrorString` can be used to obtain a readable error description.
- Starting a kernel does not return an error code, but we can use the function `cudaGetLastError` to obtain the code.

Timing: Runtimes can be measure via events of type `cudaEvent_t`.

- `cudaEventCreate`, `cudaEventDestroy` create and destroy events.
- `cudaEventRecord` records an event.
- `cudaEventSynchronize` waits for the event to complete
- `cudaEventElapsedTime` returns the elapsed time between events.

Example: Error handling and timing

```
cudaEvent_t ev_start, ev_stop;
cudaError_t cu_error;

cudaEventRecord(ev_start);
grav<<<(n+255)/256, 256>>>(n, dy1, dy2, dy3, dm, dphi);

cu_error = cudaGetLastError();
if(cu_error != cudaSuccess) {
  printf("CUDA error: \"%s\"\n",
         cudaGetErrorString(cu_error));
  abort();
}

cudaDeviceSynchronize();
cudaEventRecord(ev_stop);
cudaEventSynchronize(ev_stop);
cudaEventElapsedTime(&t_kernel, ev_start, ev_stop);
printf("  %.3f milliseconds\n", t_kernel);
```

OpenCL: Open Compute Language

Approach: Function library for computing on GPUs, synchronization, and data transfer.

Advantages:

- Open standard, supported by Intel, AMD, NVIDIA, ARM.
- Very flexible.
- Powerful programming language for GPU code.

Disadvantages:

- Handling cumbersome, since not tightly integrated with compiler.
- Only limited support for NVIDIA hardware.

Example: Gravitation

Goal: Evaluate gravitational potentials

$$\varphi_i = \sum_{\substack{j=1\\ j\neq i}}^{n} \frac{m_j}{\|x_j - x_i\|}.$$

```
__kernel void
grav(int n, __global float *x1, __global float *x2,
     __global float *x3, __global float *m,
     __global float *phi)
{
  float sum, norm;
  float3 xi, xj;
  int i = get_global_id(0);
  int j;
  if(i < n) {
    /* compute phi[i] */
  }
}
```

Example: Gravitation

```
__kernel void
grav(int n, /* ... */)
{
  /* ... */
  if(i < n) {
    xi = (float3) (x1[i], x2[i], x3[i]);
    sum = 0.0;
    for(j=0; j<n; j++) {
      xj = (float3) (x1[j], x2[j], x3[j]);
      if(j != i)
        sum += m[j] / length(xj - xi);
    }
    phi[i] = sum;
  }
}
```

Example: Gravitation

Host program a “little” longer than for CUDA:

1. Choose the device via `clGetPlatformIDs` and `clGetDeviceIDs`.
2. Create an OpenCL context via `clCreateContext`.
3. Build the GPU program via `clCreateProgramWithSource`, `clBuildProgram`.
4. Create a command queue via `clCreateCommandQueue`.
5. Allocate graphics memory via `clCreateBuffer`.
6. Copy data via `clEnqueueWriteBuffer`.
7. Set up the GPU program via `clCreateKernel`, `clSetKernelArg`.
8. Run the GPU program via `clEnqueueNDRangeKernel`.
9. Copy data via `clEnqueueReadBuffer`.

Summary

Graphics processors offer significant computing power that can be used for some applications.

SIMT: Most current graphics processors are essentially vector computers with a user-friendly thread-based programming model.

CUDA is a popular standard for programming NVIDIA graphics cards. (Competitors are, e.g., open standards like OpenCL and Vulkan)

Hosts are responsible for initializing computing devices, moving data into and out of device memory, and starting kernels on the devices.

Devices consist of multiprocessors that can execute blocks of threads working with device memory.