Known methods of parallelism

**MPI**

- Clusters
- Over networks

**OpenMP**

- One computer
- Parallelism over multiples cores
What is CUDA?

- CUDA means: Compute Unified Device Architecture.
- CUDA is developed by NVIDIA for computing over graphic devices.
- The architecture used from G8x.
- There are many flavors (C, Fortran, OpenCL, Python, etc.)
Former Graphics Pipelines

- Input
- Vertex shader
- Geometry shader
- Rasterizer
- Pixel shader
- Output merger
NVIDIA GPUs structure

2006: unified structure.

Why unify?
- Vertex Shader
- Pixel Shader

Heavy Geometry
Workload Perf = 4

Heavy Pixel
Workload Perf = 8
Let's see a CPU core
CPU on diet

Fetch / Decode

ALU

Contexto
Parallelism
SIMD

Fetch / Decode

ALU  ALU  ALU  ALU
ALU  ALU  ALU  ALU
ALU  ALU  ALU  ALU

Ctx  Ctx  Ctx  Ctx
Ctx  Ctx  Ctx  Ctx

Contexto compartido
More parallelism
What if...

not everyone executes the same code?
And the memory access?

- No more cache, more(?) latency
- More parallelism available
  - Many more threads than execution units.
Hiding memory latency
Hardware examples

G110 (GeForce GTX580, Tesla c2090):

- 16 Streaming Multiprocessors (SM)
- 32 CUDA cores per SM
- Two instructions per CUDA core per cycle @ 1554MHz (GTX 580) = 1581 GFLOPS
- 192 GBPS to memory = 33 instructions per float access
- Adicional information:
  - 128KB registers per SM
  - 64KB shared memory / cache L1 per SM
  - 12KB read only cache per SM
  - 768KB global cache L2
Compared to CPU high-end

~15 times more throughput, ~10 times more memory bandwidth

Comparison between Core i7 and GTX280 Performance.
Some differences with CPU

• More threads is better
• Cost launching thread: ~0
• Cost in context switch between threads: ~0
• Cost terminating threads: ~0
• Is better to recalculate
• Use registers or shared memory instead of global memory
CUDA C

- C++ Syntax & Semantics
The kernel is the function which runs in each thread

Don't return values

__global__ prefix

A kernel can call __device__ functions

No recursion allowed
an example

```c
__device__ int abs(int a) {
    return a < 0 ? -a : a;
}

__global__ void distance(int * a, int * b, int * c) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    c[idx] = abs(a[idx] - b[idx]);
}
```
Blocks

Threads a grouped in *blocks* from 1 to 3 dimensions and run in the same SM

- Data declared as `__shared__` is shared between threads
- Threads in the same block can synchronize using `__syncthreads`
- The `threadIdx` predefined variable allocates the thread coordinate
- Predefined `blockDim` allocates the block size
__global__ void sum_all(int * a) {
    __shared__ int s[256];
    int idx = threadIdx.x;

    s[idx] = a[idx];
    __syncthreads();

    int sum = 0;
    for (int i = 0; i < 256; ++i) {
        sum += s[i];
    }
    a[idx] = sum;
}
Launching kernels

• Specify block size
• Put them in a (up to 2 dimensions) grid (in newer cards may be 3)
• Predefined blockIdx allocates the block coordinate
• Predefined blockDim allocates the grid size
```c
__global__ void distance(int * a, int * b, int * c) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    c[idx] = abs(a[idx] - b[idx]);
}

int main(...) {
    // ...
    dim3 dim_block, dim_grid;
    dim_block.x = 256;
    dim_grid.x = ceil(N / dim_block.x);
    distance<<<dim_grid, dim_block>>>(vector1, vector2, result);
    // ...
}
```
Memory management

• GPU owns memory addresses, where you can allocate device memory or map host memory
  • Can't receive host pointers
  • Programmer tracks which are host pointers and device pointers
• GPU memory management is similar to C language
  • `cudaMalloc` allocates memory and returns the pointer (as a parameter)
  • `cudaFree` frees allocated memory in a pointer
  • `cudaMemcpy` copies in/from/to the device
```c
int h_a[N];
initialize(a);

int *d_a;
cudaMalloc(&d_a, N * sizeof(int));
cudaMemcpy(d_a, h_a, N * sizeof(int), cudaMemcpyHostToDevice);

modify<<<grid,block>>>(d_a);

cudaMemcpy(h_a, d_a, N * sizeof(int), cudaMemcpyDeviceToHost);
```
Questions?
Bibliography


• Wolovick Nicolás - Bederián Carlos, Basic course PEAGPGPU, 2011.