Introduction to Parallel and Distributed Processing

CUDA Programming

Jaroslaw 'Jaric' Zola

http://www.jzola.org/

Suggested Reading

- Blog of Mark Harris (NVIDIA Chief Technologist for Software)
  https://devblogs.nvidia.com/parallelforall/author/mharris/

NVIDIA CUDA

- SDK and platform for programming NVIDIA's GPGPUs
- Designed for CUDA-enabled GPGPUs
- Compiler, runtime and supporting tools (e.g. profiler)
- Support for C, C++11 and Fortran

Basic CUDA Workflow

- Allocate memory on host/device
- Transfer data from host to device
- Configure and run kernel(s)
- Transfer data from device to host
- Free memory

Basic CUDA Workflow

- Allocate memory on host/device
- Transfer data from host to device
- Configure and run kernel(s)
- Transfer data from device to host
- Free memory
- Since CUDA 6 memory can be managed
  (i.e. no need to copy)

Simple CUDA Code

```
// ex01.cpp
#include <vector>
#include <cuda_runtime_api.h>

int main(int argc, char* argv[]) {
  int n = 4 * 1024 * 1024;
  int size = n * sizeof(float);

  std::vector<float> x(n);
  std::vector<float> y(n);

  float* d_x;
  float* d_y;

  cudaMalloc(&d_x, size);
  cudaMalloc(&d_y, size);

  cudaMemcpy(d_x, x.data(), size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y.data(), size, cudaMemcpyHostToDevice);
  // ...
  cudaMemcpy(d_y, y.data(), size, cudaMemcpyDeviceToHost);

  cudaFree(d_x);

  return 0;
}
```

Simpler CUDA Code

```cpp
// ex01.cpp
#include <vector>
#include <cuda_runtime_api.h>

int main(int argc, char* argv[]) {
  int n = 4 * 1024 * 1024;
  int size = n * sizeof(float);

  std::vector<float> x(n);
  std::vector<float> y(n);

  float* d_x;
  float* d_y;

  cudaMallocManaged(&d_x, size);
  cudaMallocManaged(&d_y, size);
  // ...
  // may need sync: cudaDeviceSynchronize();
  // ...
  cudaFree(d_x);

  return 0;
}
```

```
nvcc -O3 ex01.cpp -o ex01
nvprof ./ex01
```

Let's SAXPY

- Just a reminder, the most basic BLAS1 routine :-)

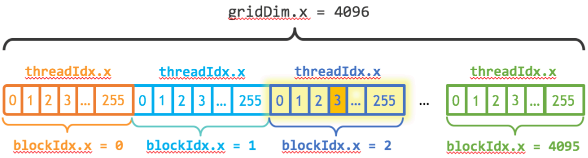
$$z = \alpha \cdot x + y$$

Let's SAXPY

- Idea: we want each CUDA thread to run just one operation: $y_i = a \cdot x_i + y_i$

```
void saxpy(int n, float a, const float* x, float* y) {
  // ...
  y[i] = a * x[i] + y[i];
}
```

- x and y are 1D so it makes sense to organize threads into 1D grid/blocks:




```
index = blockIdx.x * blockDim.x + threadIdx.x
index =    (2)     *    (256)    +    (3)      = 515
```

Let's SAXPY

```
// ex01.cu
__global__
void saxpy(int n, float a, const float* x, float* y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

__host__
void run_saxpy(int n, float a, const float* x, float* y) {
  const int block_size = 1024;
  int num_blocks = (n + block_size - 1) / block_size;
  saxpy<<<num_blocks, block_size>>>(n, 0.1, x, y);
}
```

CUDA Syntax Basics

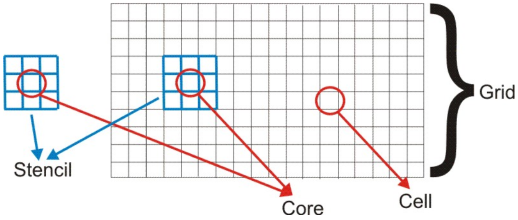
- `__global__` function runs on device but callable from host
- `__device__` runs and callable from device only
- `__host__` function runs on host only
- `fun<<< >>>(...)` calls kernel with grid/block configuration
- `threadIdx`, `blockIdx`, `blockDim` are built-in variables
- Inside block threads indexed always in row-wise manner
- CUDA code must be in `.cu` file

SAXPY with Thrust

```
// ex02.cpp
#include <thrust/device_vector.h>
#include <thrust/functional.h>
#include <thrust/transform.h>

using thrust::placeholders;

int main(int argc, char* argv[]) {
  int n = 4 * 1024 * 1024;

  thrust::host_vector<float> x(n);
  thrust::host_vector<float> y(n);

  thrust::device_vector<float> d_x = x;
  thrust::device_vector<float> d_y = y;

  float a = 0.1;
  thrust::transform(d_x.begin(), d_x.end(),
                    d_y.begin(), d_y.begin(), a * _1 + _2);

  y = d_y;

  return 0;
}
```

Stencil Computation

- Stencil: pattern in which vector/array is updated

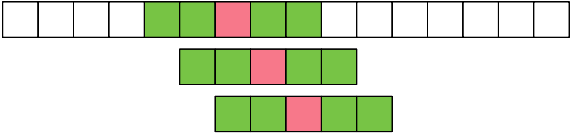


- One of the most basic (parallel) operations signal processing, statistics, ML, etc.
- Example 1D stencil:

$$y_i = \frac{1}{2r+1} \sum_{j=i-r}^{i+r} x_j$$

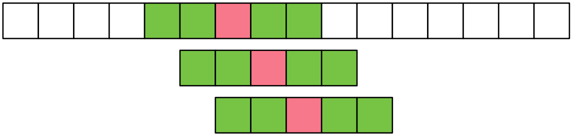
CUDA Stencil

- Again 1D pattern: each thread computes y_i
- To compute stencil around x_i we need $2r$ flanking elements
- Hence, each element will be accessed $2r + 1$ times



CUDA Stencil

- Again 1D pattern: each thread computes y_i
- To compute stencil around x_i we need $2r$ flanking elements
- Hence, each element will be accessed $2r + 1$ times



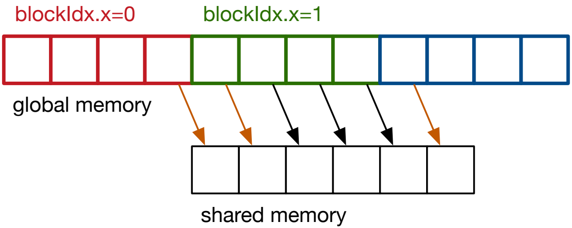
- Idea: make threads to cooperate within their block to "cache" and share/reuse x_i

Shared Memory

- Shared memory: extremely fast on-chip local memory access in 4 cycles
- Shared by threads within block, hence can be used for communication
- Declared with `__shared__`

Shared Memory

- Shared memory: extremely fast on-chip local memory access in 4 cycles
- Shared by threads within block, hence can be used for communication
- Declared with `__shared__`
- Shared memory in stencil ($r = 1$):



CUDA Stencil

```
template <int block_size, int radius>
__global__ void stencil1D(int n, const float* x, float* y) {
  __shared__ float buf[block_size + 2 * radius];

  int gidx = blockIdx.x * block_size + threadIdx.x;
  int lidx = threadIdx.x + radius;

  buf[lidx] = x[gidx]; // yey, let's load

  if (threadIdx.x < radius) { // ghost region
    buf[lidx - radius] = x[gidx - radius];
    buf[lidx + block_size] = x[gidx + block_size];
  }

  __syncthreads();

  float S = 0.0;
  for (int i = -radius; i <= radius; ++i) S += buf[lidx + i];
  y[gidx] = S / (2 * radius + 1);
}
```

For Fun

- Write host code to invoke stencil and then profile it!