GPGPU programming

Presented by Lukas Gamper and Alex Kosenkov

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

software carpentry

Agenda

- CUDA
- (OpenCL)
- thrust
- OpenACC

CUDA

- Compute Device Unified Architecture
- General-Purpose computing on Graphics Processing Units

CUDA: $ module load cuda

Makefile

```
objects = main.o vectoradd.o

all: $(objects)
    nvcc -arch=sm_20 $(objects) -o app

%.o: %.cpp
    nvcc -x cu -arch=sm_20 -I. -dc $< -o $@

clean:
    rm -f *.o app
```

```
$ make
$ nvcc -x cu -arch=sm_20 -I. -dc main.cpp -o main.o
$ nvcc -x cu -arch=sm_20 -I. -dc vectoradd.cpp -o vectoradd.o
$ nvcc -arch=sm_20 main.o vectoradd.o -o app
```

vectoradd.h

```
#ifndef VECTORADD_H
#define VECTORADD_H

#include <cuda_runtime.h>

__global__ void vectorAdd(const float *A, const float *B, float *C, int n);
__device__ void vectorAdd_impl(const float& A, const float& B, float& C);

#endif
```

vectoradd.cpp

```
#include "vectoradd.h"

__global__ void vectorAdd(const float *A, const float *B, float *C, int n){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i < n) vectorAdd_impl(A[i], B[i], C[i]);
}

__device__ void vectorAdd_impl(const float& A, const float& B, float& C){
    C = A + B;
}
```

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

CSCS Centro Svizzero di Calcolo Scientifico Swiss National Supercomputing Centre

software carpentry

CUDA: $ module load cuda

main.cpp

```
#include <stdio.h>
#include "vectoradd.h"
#define N 50000

int main(){
    size_t size = N*sizeof(float);
    float *h_A = (float *)malloc(size);
    float *h_B = (float *)malloc(size);
    float *h_C = (float *)malloc(size);
    float *d_A = NULL; cudaMalloc((void **)&d_A, size);
    float *d_B = NULL; cudaMalloc((void **)&d_B, size);
    float *d_C = NULL; cudaMalloc((void **)&d_C, size);

    for(int i = 0; i < N; ++i){
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    printf("CUDA kernel launch with %d blocks of %d threads\n", blocksPerGrid, threadsPerBlock);
    vectorAdd <<< blocksPerGrid, threadsPerBlock >>> (d_A, d_B, d_C, N);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    free(h_A); free(h_B); free(h_C);
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    cudaDeviceReset();
    return 0;
}
```

CUDA: $ module load cuda

Launch:

```
#define N 50000

int threadsPerBlock = 256;
int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

vectorAdd <<< blocksPerGrid, threadsPerBlock >>> (d_A, d_B, d_C, N);
```

Kernel:

```
__global__ void vectorAdd(const float *A, const float *B, float *C, int n){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i < n) C[i] = A[i] + B[i];
}
```

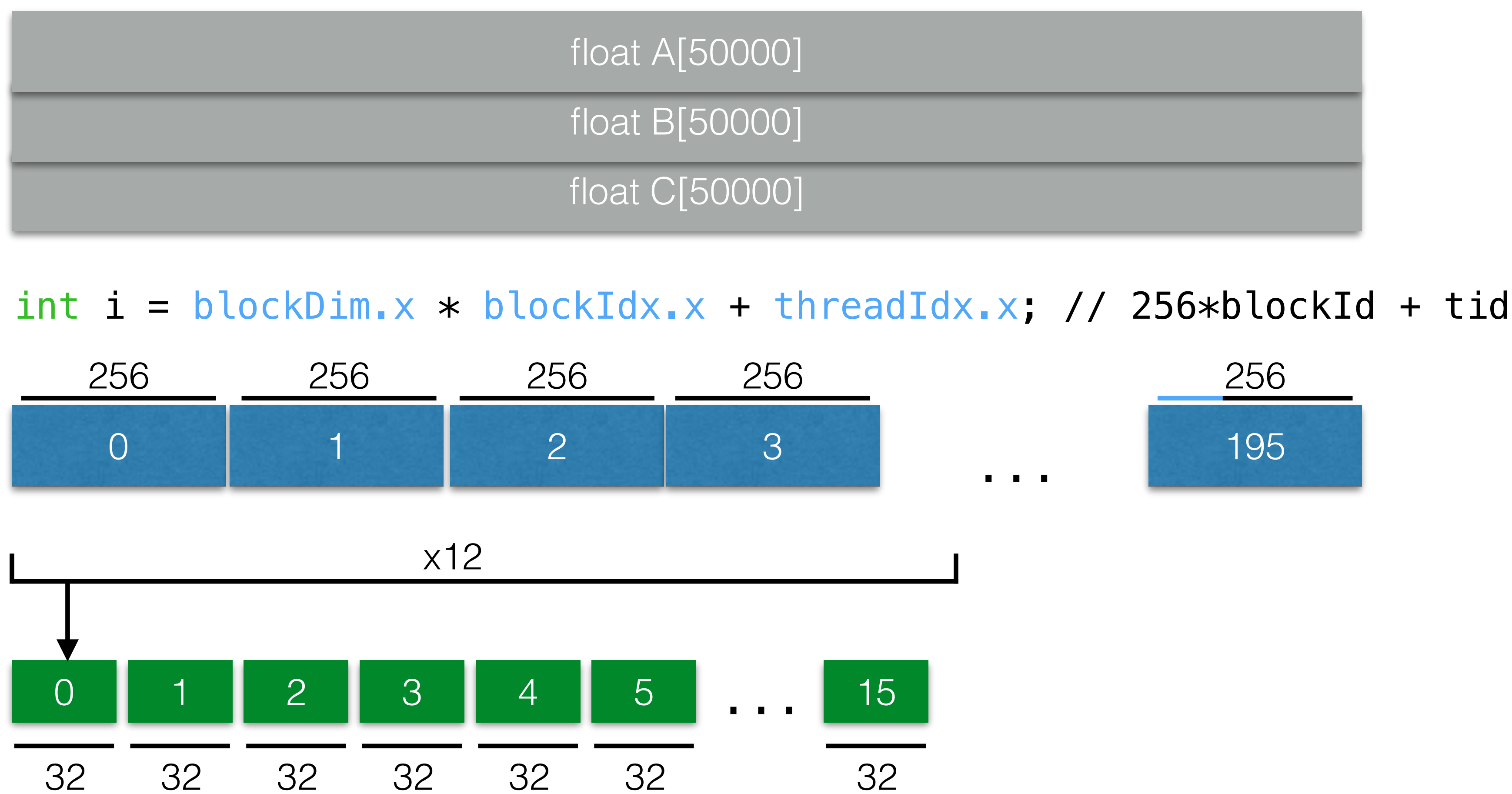
```
CUDA kernel launch with 196 blocks of 256 threads
```

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

CSCS Centro Svizzero di Calcolo Scientifico Swiss National Supercomputing Centre

software carpentry

CUDA: $ module load cuda



12 blocks x 8 warps per MP

CUDA: limitations

- Data transfer time
- Memory coalescing
- Execution divergence

CUDA: parallel reduction (nvidia example)

main.cpp (kernel invocation)

```
float hostReduce(float* input, unsigned int n){
    for(unsigned int stride = 1; stride < n; stride *= 2)
    for(unsigned int k = stride; k < n; k += stride*2)
    input[k-stride] += input[k];
    return input[0];
}
...
void deviceReduce(int blocks, int block_size, float* input, float* output, unsigned int n){
    int shmemSize = block_size*sizeof(float);
    reduce<<< blocks, block_size, shmemSize >>>(input, output, n);
}
...
int main(){
...
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
...
    deviceReduce(blocksPerGrid, threadsPerBlock, d_C, d_A, N);
    deviceReduce(1, blocksPerGrid, d_A, d_B, blocksPerGrid);
    cudaMemcpy(h_B, d_B, sizeof(float), cudaMemcpyDeviceToHost);

    std::cout << "Host sum is " << hostReduce(h_C, N) << "\n";
    std::cout << "Device first sum is " << h_B[0] << "\n";
...
}
```

CUDA: parallel reduction #0

```
__global__ void reduce0(float *g_idata, float *g_odata, unsigned int n) {
    extern __shared__ float sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Tip: execution divergence

CUDA: parallel reduction #1

```
__global__ void reduce1(float *g_idata, float *g_odata, unsigned int n) {
    extern __shared__ float sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
    for (unsigned int s=1; s < blockDim.x; s *= 2) {
        int index = 2 * s * tid;
        if (index < blockDim.x) {
            sdata[index] += sdata[index + s];
        }
        __syncthreads();
    }
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Tip: memory banks conflicts

CUDA: parallel reduction #2

```
__global__ void reduce2(float *g_idata, float *g_odata, unsigned int n) {
    extern __shared__ float sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();
    for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Tip: idle threads

CUDA: parallel reduction #3

```
__global__ void reduce3(float *g_idata, float *g_odata, unsigned int n) {
    extern __shared__ float sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
    sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
    __syncthreads();
    for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Tip: useless instructions

CUDA: parallel reduction #4

```
__device__ void warpReduce(volatile float* sdata, int tid) {
    sdata[tid] += sdata[tid + 16];
    sdata[tid] += sdata[tid + 8];
    sdata[tid] += sdata[tid + 4];
    sdata[tid] += sdata[tid + 2];
    sdata[tid] += sdata[tid + 1];
}

__global__ void reduce4(float *g_idata, float *g_odata, unsigned int n) {
    extern __shared__ float sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockDim.x*2) + tid;
    sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
    __syncthreads();
    for (unsigned int s=blockDim.x/2; s>16; s>>=1) {
        if (tid < s) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    if (tid < 16) warpReduce(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

CUDA: parallel reduction #5

```
template <unsigned int blockSize>
__device__ void warpReduce (volatile float *sdata, unsigned int tid) {
    if(blockSize >= 64) sdata[tid] += sdata[tid + 32];
    if(blockSize >= 32) sdata[tid] += sdata[tid + 16];
    if(blockSize >= 16) sdata[tid] += sdata[tid + 8];
    if(blockSize >= 8)  sdata[tid] += sdata[tid + 4];
    if(blockSize >= 4)  sdata[tid] += sdata[tid + 2];
    if(blockSize >= 2)  sdata[tid] += sdata[tid + 1];
}
template <unsigned int blockSize>
__global__ void reduce5(float *g_idata, float *g_odata, unsigned int n) {
    extern __shared__ float sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
    __syncthreads();
    if(blockSize >= 512){ if(tid < 256) sdata[tid] += sdata[tid + 256]; __syncthreads(); }
    if(blockSize >= 256){ if(tid < 128) sdata[tid] += sdata[tid + 128]; __syncthreads(); }
    if(blockSize >= 128){ if(tid <  64) sdata[tid] += sdata[tid +  64]; __syncthreads(); }
    if(tid < 32) warpReduce<blockSize>(sdata, tid);
    if(tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

CUDA: parallel reduction #6

```

template <unsigned int blockSize>
__global__ void reduce(float *g_idata, float *g_odata, unsigned int n) {
    extern __shared__ float sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0.;
    while(i < n){ sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; }
    __syncthreads();
    if(blockSize >= 512){ if(tid < 256) sdata[tid] += sdata[tid + 256]; __syncthreads(); }
    if(blockSize >= 256){ if(tid < 128) sdata[tid] += sdata[tid + 128]; __syncthreads(); }
    if(blockSize >= 128){ if(tid <  64) sdata[tid] += sdata[tid +  64]; __syncthreads(); }
    if(tid < 32) warpReduce<blockSize>(sdata, tid);
    if(tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

CUDA: thrust

Containers:

thrust::device_vector< T, Alloc >
thrust::host_vector< T, Alloc >

Examples of supported algorithms:

thrust::for_each
thrust::fill, thrust::generate, thrust::transform
thrust::sort, thrust::find, thrust::find_if
thrust::copy, thrust::copy_if
thrust::merge, thrust::reduce, thrust::inner_product

CUDA: thrust

Example: basic usage

```cpp
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <algorithm>
#include <cstdlib>

int main(void){
    // generate 32M random numbers serially
    thrust::host_vector<int> h_vec(32 << 20);
    std::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());

    return 0;
}
```

CUDA: thrust

Example: saxpy (y[i] += a*x[i])

```
#include <thrust/transform.h>
#include <thrust/device_vector.h>
#include <thrust/host_vector.h>
#include <thrust/functional.h>
#include <iostream>
#include <iterator>
#include <algorithm>

struct saxpy_functor : public thrust::binary_function<float,float,float> {
    saxpy_functor(float _a) : a(_a) {}

    __host__ __device__ float operator()(const float& x, const float& y) const {
        return a * x + y;
    }
    const float a;
};

void saxpy_fast(float A, thrust::device_vector<float>& X, thrust::device_vector<float>& Y){
    thrust::transform(X.begin(), X.end(), Y.begin(), Y.begin(), saxpy_functor(A)); // Y <- A * X + Y
}

int main(void){
    float x[4] = {1.0, 1.0, 1.0, 1.0};
    float y[4] = {1.0, 2.0, 3.0, 4.0};

    thrust::device_vector<float> X(x, x + 4);
    thrust::device_vector<float> Y(y, y + 4);

    saxpy_fast(2.0, X, Y);
    return 0;
}
```

OpenACC

Example: stencil computation

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream>

#define M 64
#define N 128

void stencil(float A[M][N], float B[M][N]){

    #pragma acc kernels pcopyin(A[0:M]) pcopy(B[0:M])
    {
        float c11, c12, c13, c21, c22, c23, c31, c32, c33;
        c11 = +2.0f; c21 = +5.0f; c31 = -8.0f;
        c12 = -3.0f; c22 = +6.0f; c32 = -9.0f;
        c13 = +4.0f; c23 = +7.0f; c33 = +10.0f;

        #pragma acc loop gang(64)
        for (int i = 1; i < M - 1; ++i){
            #pragma acc loop worker(128)
            for (int j = 1; j < N - 1; ++j){
                B[i][j] = c11 * A[i - 1][j - 1] + c12 * A[i + 0][j - 1] + c13 * A[i + 1][j - 1]
                        + c21 * A[i - 1][j + 0] + c22 * A[i + 0][j + 0] + c23 * A[i + 1][j + 0]
                        + c31 * A[i - 1][j + 1] + c32 * A[i + 0][j + 1] + c33 * A[i + 1][j + 1];
            }
        }
    }
}

int main(){
    float A[M][N];
    float B[M][N];

    for(int i = 0; i < M; i++)
    for(int j = 0; j < N; j++)
    A[i][j] = std::rand();

    stencil(A,B);
    return 0;
}
```

Questions

End of the fourth part