Sebastian von Alfthan
Sami Ilvonen
Mikko Byckling

Introduction to Accelerators

December 8-10, 2015
PRACE Advanced Training Centre
CSC – IT Center for Science Ltd, Finland

```
                    double *dC, const double *dA, const double *dB,
                    stream *s, float *gputime, int tib)
{
    cudaEvent_t start, stop;

    CUDA_CHECK( cudaEventCreate(&start) );
    CUDA_CHECK( cudaEventCreate(&stop) );
    CUDA_CHECK( cudaEventRecord(start) );

    for (int i = 0; i < 2; ++i) {
        // Do the asynchronous copy-in calls for each stream (s[0] and s[1])
        int sidx = s[i].start;
        int slen = s[i].len;

        CUDA_CHECK( cudaMemcpyAsync((void *)&(dA[sidx]), (void *)&(hA[sidx]),
                                    sizeof(double) * slen,
                                    cudaMemcpyHostToDevice, s[i].strm) );

        CUDA_CHECK( cudaMemcpyAsync((void *)&(dB[sidx]), (void *)&(hB[sidx]),
                                    sizeof(double) * slen,
                                    cudaMemcpyHostToDevice, s[i].strm) );
    }

    for (int i = 0; i < 2; ++i) {
        int sidx = s[i].start;
        int slen = s[i].len;

        dim3 grid, threads;
        grid.x = (slen + tib - 1) / tib;
        threads.x = tib;
```

Agenda

Tuesday 28th

9:00-9:15	Course introduction
9:15-10:15	Introduction to Xeon Phi
10:15-10:30	Coffee Break
10:30-11:30	Offload
11:30-12:00	Exercises
12:00-12:45	Lunch break
12:45-13:45	Data access and performance
13:45-14:30	Exercises
14:30-14:45	Coffee Break
14:45-15:45	Exercises
15:45-16:15	Advanced topics and wrap-up

Wednesday 29th

9:00-9:30	Introduction to GPUs
9:30-10:30	OpenACC basics
10:30-10:45	Coffee break
10:45-12:00	Exercises
12:00-12:45	Lunch break
12:45-13:45	Data access and performance
13:45-14:30	Exercises
14:30-14:45	Coffee break
14.45-15:45	Exercises
15:45-16:15	Advanced topics and wrap-up

Thursday 30th

9:00-10:15	CUDA programming I
10:15-10:30	Coffee break
10:30-11:15	Exercises
11:15-12:15	CUDA programming II
12:00-13:00	Lunch break
13:00-14:00	Exercises
14:00-14:30	CUDA programming III
14.30-14.45	Coffee break
14:45-15:45	Exercises
15:45-16:15	Course wrap-up

Xeon Phi exercises

1. Compilation and running

Compile and run a simple OpenMP target test program, provided as **ex1_hello(.c|.F90).**

2. Work sharing

The file **ex2_sum(.c|.F90)** implements a skeleton for the simple summation of two vectors C=A+B. Add the computation loop and add an OpenMP parallel region with work sharing directives such that the vector addition is executed in parallel on the device.

3. Reduction

The file **ex3_dotprod(.c|.F90)** implements a simple dot product of two vectors. Try to parallelize the code on device by using OpenMP threading. Are you able to get the same results between different runs? What is needed for correct computation?

4. Parallelization and data transfers

The file **ex4_jacobi(.c|.F90)** implements a simple Jacobi iteration.

a) Add OpenMP directives to the code in such a way that the computational work is done in parallel.

b) Modify the OpenMP implementation in such a way that the computation is done entirely on the device, i.e., no additional data transfers take place between the host and the device.

c) (Bonus *) Optimize the performance of the implementation.

5. (Bonus *) Dense matrix-vector product

The file **ex5_mvp(.c|.F90)** implements a dense matrix vector product operation, i.e., y=Ax. When A is an m-by-n matrix, the computation takes O(m(2n-1)) floating point operations to complete. Parallelize the computation with OpenMP on the device and optimize its floating point performance.

OpenACC exercises

1. Compilation and running

Compile and run a simple OpenACC test program, provided as **ex1_hello(.c|.F90)**.

2. Work sharing

The file **ex2_sum(.c|.F90)** implements a skeleton for the simple summation of two vectors C=A+B. Add the computation loop and add an OpenACC parallel region with work sharing directives such that the vector addition is executed in parallel on the device.

3. Reduction

The file **ex3_dotprod(.c|.F90)** implements a simple dot product of two vectors. Try to parallelize the code by using OpenACC parallel loop or kernels pragmas. Are you able to get the same results between different runs? Is the behaviour different from what you expect?

4. Parallelization and data transfers

The file **ex4_jacobi(.c|.F90)** implements a simple Jacobi iteration.

a) Add OpenACC directives to the code in such a way that the computational work is done in parallel on the device.

b) Modify the OpenACC implementation in such a way that the computation is done entirely on the device, i.e., no additional data transfers take place between the host and the device.

c) (Bonus *) Optimize the performance of the implementation.

5. (Bonus *) Dense matrix-vector product

The file **ex5_mvp(.c|.F90)** implements a dense matrix vector product operation, i.e., y=Ax. When A is an m-by-n matrix, the computation takes O(m(2n-1)) floating point operations to complete. Parallelize the computation with OpenACC on the device and optimize its floating point performance.

CUDA exercises

1. First program

See the lecture slides and write a simple program that implements the example presented at the lecture. The program should do the following:

1. Allocate memory for a vector from host memory and device memory
2. Call the kernel with two arguments: pointer to the allocated device memory and the length of the array
3. Implement a kernel that assigns the global thread index to each element in the vector
4. Copy the result vector from device memory to the allocated host memory buffer
5. Print out the values for checking

Pay close attention to the kernel call parameters, block and grid sizes!

2. Error checking

Start from the provided code skeleton and write a simple vector addition kernel that computes element-wise sum **C**=**A**+**B**. Print again the results to check the correctness of your program.

Note that the skeleton provides macros that can be used to check the error codes of CUDA runtime function calls and kernel calls.

Now experiment with different error cases:

1. What happens if you try to launch kernel with too large block size? When do you catch the error if you remove the **cudaDeviceSynchronize()** call ?
2. What happens if you try to dereference a pointer to device memory in host code (that is, you make mistake in keeping track of type of different pointers)?
3. What happens if in the kernel a pointer to host memory is used? Try running your program with **cuda-memcheck** and see the results.

3. 2D indices, events and timing

In this exercise we will implement a simple Jacobi iteration that is a very simple finite-difference scheme. Familiarize yourself with the provided skeleton. Then implement following things:

1. Write the missing CUDA kernel **sweepGPU** that implements the same algorithm as the **sweepCPU** function. Check that the reported average difference is in the order of numerical accuracy.
2. The skeleton provides the timing calls for the CPU part. Implement timing for the GPU part by adding the needed CUDA event routine calls.
3. Experiment with different grid and block sizes and compare the execution times.

4. Streams

In this exercise we modify the vector addition program and implement overlapping copying and computing using streams. Start from the provided skeleton and implement following parts:

1. Do the memory allocations so that host memory is page-locked
2. Add the missing parts to the function `streamtest` and do the vector addition using two streams