Exercise Sheet 4: Intel Cilk and OpenCL

Gaurav Kukreja, Evangelos Drossos

December 16, 2013

1 Exercise 11: Cilk and OpenCL

1.1 Advantages of Cilk and keywords needed

Cilk is designed keeping in mind, that the developers and software producers are not interested in rewriting the code, to exploit the benefits of parallelism. Therefore it is designed in a way such that developers can exploit parallelism, by making small modifications to existing code. The resulting code continues to perform on older machines with single processors, and makes us of parallel hardware whenever possible.

Using Cilk is easier compared to other parallel programming constructs and frameworks. It supports 3 keywords _cilk_spawn, _cilk_sync and _cilk_for.

1.1.1 cilk_spawn

This keywords modifies a function call statement to tell the runtime system that the function may(but is not required to) run in parallel with the caller. This keyword may be used as follows

```
type var = cilk_spawn (func args) ; // func () returns a value
var = cilk_spawn func(args) ; // func () returns a value
cilk_spawn func(args) ; // func() may return void
```

1.1.2 cilk_sync

The cilk_sync statement indicates that the current function cannot continue in parallel with its spawned children. After the children all complete, the current function can continue.

```
cilk_sync ;
```

This statement only syncs with children spawned by this function. Children of other functions are not affected.

1.1.3 cilk_for

A cilk_for loop is a replacement for the normal C/C++ for loop that permits loop iterations to run in parallel. The syntax is as follows

```
cilk_for(declaration;
         conditional expression;
         increment expression)
body
```

There are some restrictions to use of cilk_for. The loop must be simple, with uniform stride length. The control variable, or iteration counter must not be changed inside the loop, other than in the increment expression of the cilk_for statement. The termination condition should be simple, and should not change between iterations.

1.2 Array Expressions with Intel C language extensions

C/C++ language extensions for array notations is an Intel-specific language extension that is part of Intel Cilk Plus feature supported by Intel Compiler. The extension provides following major benefits:

1. Allows use of array notations to program parallel operations

2. Achieves predictable performance based on mapping parallel constructs to the underlying multi-threaded and SIMD hardware.

3. Enables compiler parallelization and vectorization with less reliance on alias and dependence analysis.

1.2.1 Array Section Notation

A new section operator is introduced.

```
section_operator ::= [<lower_bound> : <length> : <stride >]
```

Here, *lower_bound*, *length* and *stride* are integers. The statement represents a set of integer values as follows

```
<lower_bound>, <lower_bound + stride>, <lower_bound + 2*stride>,
 ... , <lower_bound + (length-1)*stride>
```

A section operator can occur in place of a subscript operator. That is, elements of an array a[lb], a[lb+str], a[lb+2*str], ..., a[lb + (len-1)*str] can be represented as a[lb:len:str]. Further usage of the section operator is illustrated as follows:

```
a[0:3][0:4]       // refers to 12 elements in the 2-dimensional
                  array starting at row 0, column 0, and ending
                  at row 2,column 3
b[0:2:3]          // refers to elements 0 and 3 of 1 dimensional
                  array b
b[:]              // refers to the entire array b
```

1.2.2 Types of expressions

- **Operators**
  Most C/C++ operators are available for array sections. Following expressions can be used.

  ```
  a[:] * b[:]       //element-wise multiplication
  a[0:3][0:3] + b[1:4][1:4] // matrix addition
  a[0:3][0:3] + b[0][1] // allowed. adds scalar b[0][1] to an
          array section.
  ```

  Operations between sections of different size will fail.

- **Assignment**

  Assignment operator applies in parallel to every element of the array section.

  ```
  a[0:3] = b[0:3] + c;
  a[:][:] = d[:]; // error, different rank
  ```

- **Gather and Scatter**

  When an array section occurs directly under a subscript expression, it designates a set of elements indexed by the values of the array section.

  ```
  unsigned index[10] = {0,1,2,3,4,5,6,7,8,9};
  float out[10], in[10] = {9,8,7,6,5,4,3,2,1,0};
  out[0:5] = in[index[0:5]]; // gather
  out[index[5:5]] = in[0:5]; //scatter
  for(int i = 0; i < 5; i++){
          cerr << "out[" << i << "]" << out[i] << endl;
  }
  ```

- **Reduction Operations**

  There are some built in Reduction functions which use the section operator e.g. *__sec_reduce_add (a[:])*, which adds the values passed as arrays, and returns result. Similarly, following reduction functions are built-in.

  - `__sec_reduce_mul (a[:])  // multiplies all values in array`
  - `__sec_reduce_all_zero (a[:])  // checks if all values are 0`
  - `__sec_reduce_all_nonzero (a[:])  // checks if all values are not 0`
  - `__sec_reduce_any_nonzero (a[:])  // tests for a value that is not 0`
  - `__sec_reduce_min (a[:]  // returns minimum value`
  - `__sec_reduce_max (a[:])// returns maximum value`
  - `__sec_reduce_min_ind (a[:])  // returns index of min value`
  - `__sec_reduce_max_ind (a[:])  // returns index of max value`

- **Shift Operations** Intel Cilk PLus supports shift and rotate operations on array sections. The function prototypes are as follows

  ```
  b[:] = __sec_shift(a[:], signed shift_val, fill_val) // Generic Shift Fun
  b[:] = __sec_rotate(a[:], signed shift_val) // Generic Rotate Function
  ```

- Function Maps Maps are implicitly defined on scalar functions. They take array sections, and run on each element in parallel, with no specific ordering. For example,

  ```
  a[:] = sin(b[:]);
  a[:] = pow(b[:], c);
  a[:] = pow(c, b[:]);
  ```

4

1.3 Memory Structure in OpenCL

OpenCL is a framework for exploiting parallel hardware found in CPUs, GPGPUs and other special hardware. Unlike CUDA, OpenCL is designed to be compatible with a vast variety of hardware. Therefore, OpenCL assumes a uniform hardware architecture for the sake of its execution and memory model. The appropriate features provided by the hardware are mapped to this architecture by the run-time system.

The following figure shows the hardware architecture assumed by OpenCL.
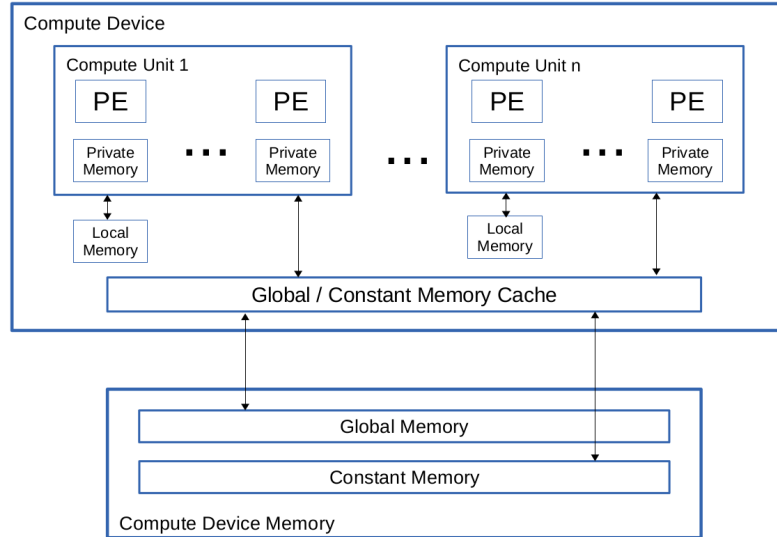

Figure 1: Memory and Hardware Structure in OpenCL

Let's consider GPGPUs for our reference Compute Device. The Compute Device consists of several Compute Units (Arithmetic Logical Units that execute code). The Compute Device has Global Memory, which can be accessed by all the compute units in the device. This is a high latency memory. This memory can also be accessed by the Host Device (CPU). The host device communicates with the Compute Device using this memory.

The Device also has a low latency Constant Memory. Like Global Memory, this memory can also be accessed by all Compute Units, but none of the Compute Units can write to this memory. Only the Host (CPU) will write to this memory. This memory can be used to provide input data which does not need to be modified during the execution of code. The actual hardware may not have the Constant Memory, in which case the run-time system will simply use the Global Memory.

The device has a Global/Constant Memory Cache. The content of this memory cannot be directly controlled by the Compute Units. This is low latency cache memory which is maintained by the run-time system.

Local Memory is a low latency memory, which is private to each Compute Unit. One Compute Unit may not access contents in another Units Local Memory. OpenCL provides programmers with the facility to designate data that needs to be held in Local Memory. The actual hardware may not have Local Memory, or may have only a small amount of Local Memory. In this case, the data will be stored in Global Memory.

For best results, the user must customize the code such that optimal use of the hardware resources can be achieved.

1.4 OpenCL Execution Model and Compilation of OpenCL Application

An OpenCL application consists of 2 parts

- Device Code

- Host Code

Host code runs on the Host Device eg. CPU. The host code is compiled by a standard C compiler like GCC. The host code contains boiler plate code, which is used to detect the hardware available on the system, and check the features. The host code initializes the hardware to be used. It then creates the executable device code. As per the application demands, the kernel along with input data is transferred to the Compute Device and execution is started. The Host then reads the result from the Compute Device.

The Device Code is written in a language very similar to C, but with some extra features and some restrictions. Special hardware specific compilers are used to compile the Device Code. The compiled Device code is called the kernel. The host code detects hardware(GPGPUs) on the system and selects an appropriate hardware. It then tries to find the corresponding compiler, and compiles the Device Code to create the kernel.

2 Exercise 12: Quicksort with Intel Cilk

2.1 Code Overview

For this exercise, we implemented the quicksort algorithm using Intel Cilk Plus. Since quicksort works in a recursive manner, we used the _cilk_spawn_ keyword to spawn new threads and parallelize the computation for the left and right subarrays. The _cilk_sync_ keyword helped us synchronize the children of the function at a given level of recursion. Following code fragment shows how our implementation works:

```
/* recursion */
cilk_spawn quicksort(data, right);
quicksort(&(data[left]), length - left);
cilk_sync;
```

2.2 Results and Analysis

We designed a number of tests to measure the performance of our implementation. We performed both weak and strong scaling for different problem sizes. Furthermore, we compared the performance of the Intel Cilk Plus Implementation with our OpenMP implementation from the second assignment.

Following figure shows the speedup achieved for different numbers of threads for the Intel Cilk implementation. Results are shown for three problem sizes: an array length of 4, 8 and 16 million elements.
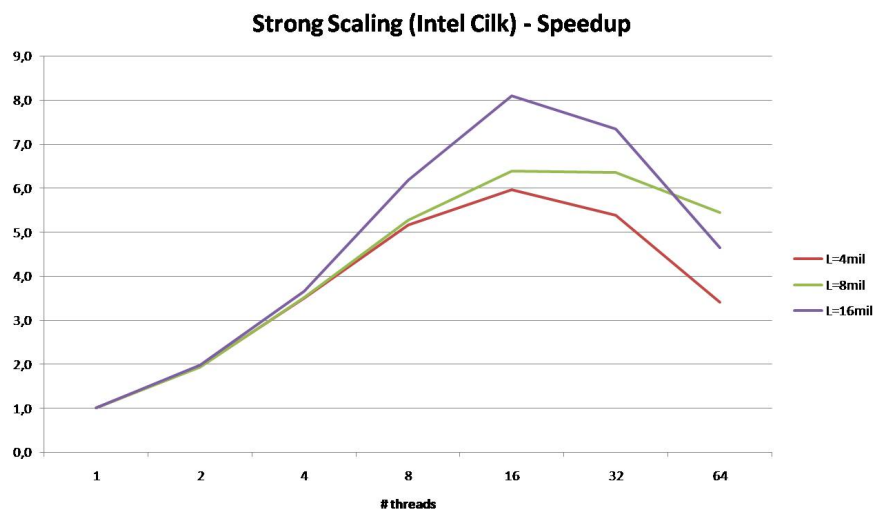

Figure 2: Speedup of Intel Cilk Plus implementation for three different problem sizes

As can be seen in figure 2, the highest speedup is achieved for 16 threads. After this point, performance levels off and a steep decline is visible for 64 threads. This is due

to the additional overhead, since the maximum number of threads for this partition has been exceeded. Speedup seems to be slightly better for larger problem sizes. It should also be noted that doubling the number of threads leads to less than twice as good performance, but performance improvements are still large.

The next figure compares the Intel Cilk variant of quicksort to the OpenMP implementation of the algorithm. It is immediately clear that both implementations achieve almost the same performance. The highest performance gain is achieved for 16 threads.
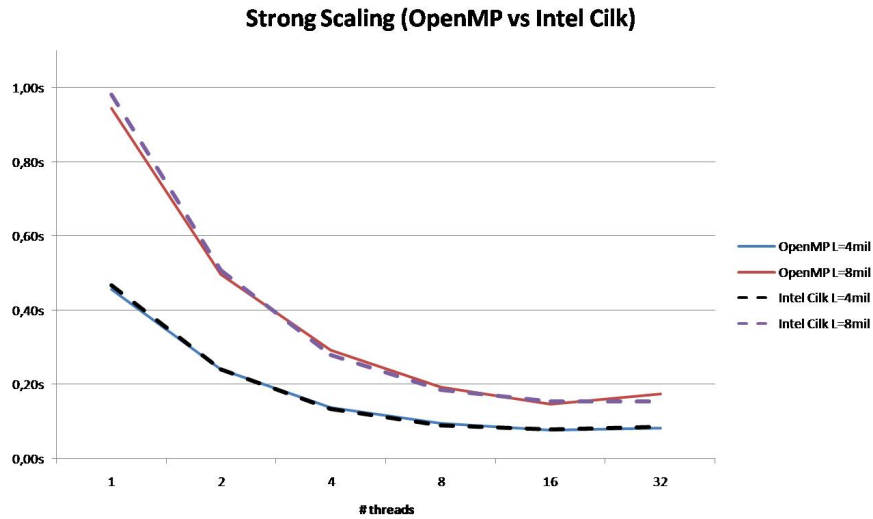

Figure 3: Strong scaling comparison for Intel Cilk and OpenMP implementations of quicksort. The vertical axis shows time elapsed in seconds and the horizontal axis the number of threads used

Lastly, we also attempted a number of weak scaling tests to compare both implementations. We started with a problem size of 1 million elements and 1 thread and iteratively doubled both problem size and the number of threads up to a cumulative factor of 32.

As demonstrated by figure 4, performance is almost identical. At first, elapsed time does not change much at each step. However, at 16 threads the needed time is already almost 3 times the time needed for initial problem of 1 million elements running on one thread. From that point onwards the problem does not scale very well.
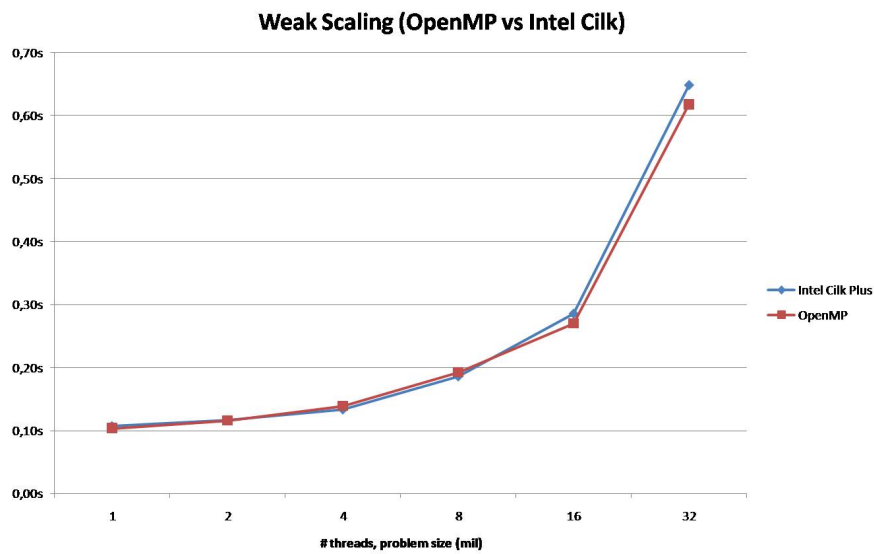
Figure 4: Weak scaling comparison for Intel Cilk and OpenMP implementations of quick-sort. The vertical axis shows time elapsed in seconds and the horizontal axis the number of threads spawned as well as the array length in million elements

3 Exercise 13: Laplace Equation with Intel Array Extensions

3.1 Code Overview

3.2 Results and Analysis

4 Exercise 12: Matrix Multiplication with OpenCL

4.1 Overview

We were able to achieve Matrix Multiplication using OpenCL with Local Memory Blocking Optimization. Further details of our implementation are discussed below.

The project consists of 2 files.

- **main_benchmark.c** - Host Code

- **matrix_mul_kernel.c** - Device Code

4.2 Algorithm

A simple implementation of Matrix Multiplication creates a 2 dimensional array of threads, one for each element of the product matrix. Each thread calculates the value of the element that it corresponds to. The threads are divided into Work Groups which run in parallel on the Compute Units in the device. Each Compute Unit has a number of Processing Elements, which process the Work Items (threads) in parallel. Each thread can be uniquely identified with combination of *global* and *local* ids in each dimension, which can be used to identify which element of the product matrix the thread must evaluate.

As can be easily seen, the simple approach is suboptimal. The Computational Intensity of this approach is very low. For a matrix of width 1024, each thread performs 2048 Reads, 1024 Floating Point Multiplications, 1024 Floating Point Additions and 1 Write operation. If we are able to reduce the number of read operations from global memory, by making use of the Local Memory, we can optimize the performance.

The Local Memory on the GPUs connected to Linux Cluster is 32 KB. We can store 3 matrices of width 32 in this memory, while still keeping space for temporary variables. In our approach, we divide matrices into square blocks of width 32. Each work group correspondingly is also a square matrix of width 32 i.e. it has 32x32 threads.

Each thread in a Work Group reads one element from each input matrix from Global Memory and stores it into Local Memory. Now we have the entire block of size 32x32 in low latency Local Memory. Each thread computes the partial result for its corresponding element in the product matrix from these values. This way, each thread performs 2 Read operations, 32 Floating Point Multiplications, 32 Floating Point Additions and 1 Write operation for calculating the partial result. The computational intensity of this approach is almost 32 times better than the simple approach.

4.3 Results and Analysis

When talking about performance results from OpenCL, care must be taken to understand that apart from the processing time, a lot of overhead is spent in compiling the kernel sending the data to the Global Memory of the GPU, and reading the results back from the GPU.

Our results show a very high performance if we consider only time spent to compile the kernel. This is expected, owing to high parallelism of an application like Matrix Multiplication, and the large number of small cores on the GPU. Following image is the graphic of the result considering only the time used to execute the kernel.
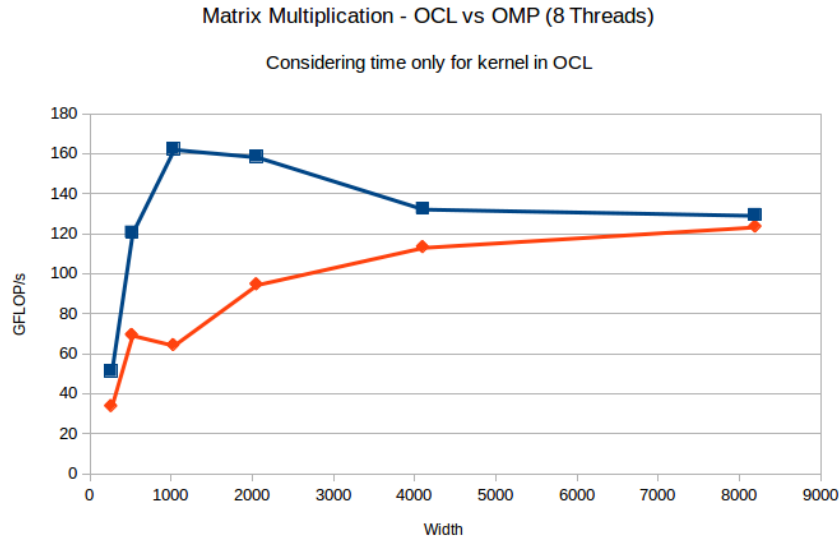

Figure 5: **Comparison of GFLOP/s achieved for Matrix Multiplication.** Blue line depicts performance using OpenCL, and Orange line depicts performance with OpenMP. We consider only time spent in executing the kernels.

The performance steeply degrades when we consider the overhead spent in compiling the kernel, sending the input data to the GPU and reading the output back from the GPU memory. Following image shows the result keeping this overhead in consideration.

You can also observe that performance with OpenCL nearly matches performance with OpenMP at larger width. The impact of the overhead is less when the problem size is large.

The OpenMP implementation used for this analysis, is running on 8 cores. It is optimized for Memory Blocking and uses Intel Compiler Intrinsics features to facilitate vectorization.
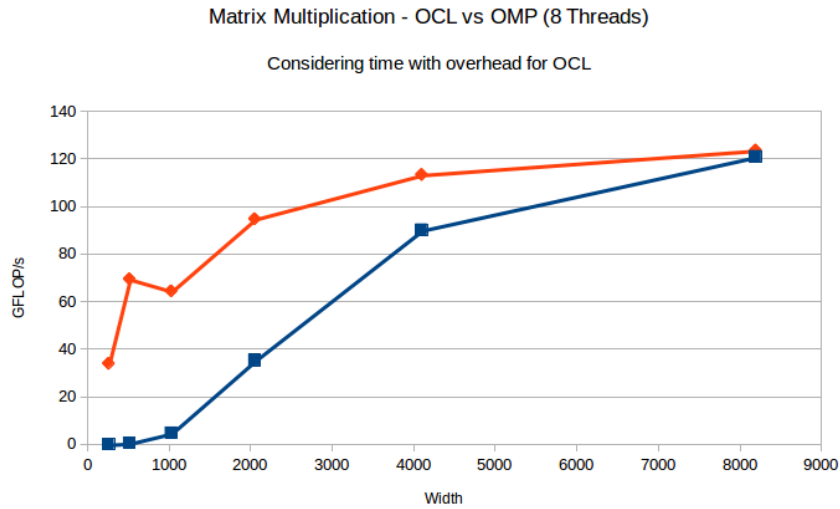
Figure 6: **Comparison of GFLOP/s achieved for Matrix Multiplication with overhead.** Blue line depicts performance using OpenCL, and Orange line depicts performance with OpenMP. Here, we consider the total time spent in compiling kernels, transferring input matrices to GPU, execution of kernels and reading output matrix from GPU.

5 Exercise 12: Laplace Equation with OpenCL

5.1 Code Overview

The challenging aspect in trying to parallelize Laplace Equation Solver with OpenCL, is to efficiently parallelize the *g_product_operator()* and *g_dot_product()* functions.

We could correctly parallelize the *g_product_operator()* function. We create one thread for calculating each element of the result matrix. Further optimizations are possible, but unfortunately we could not invest enough time into it.

We tried to parallelize *g_dot_product()* function, but failed. We created one thread for each row of the matrix. Each thread calculates the scalar product of the row vectors of the matrix. Later, we try to reduce the value calculated by each thread. We somehow could not fix some bug with reduce functionality. We could get the reduce functionality to work in a sample application trial.cpp (included in source), but simillar code does not work for the real implementation. We will continue to give more time to this.

5.2 Results and Analysis

We think that OpenCL is not an ideal choice for parallelizing this application. The kernels are small, and the overhead spent in compiling kernels, and transferring data between GPUs is very expensive. This is also what is being reflected in the results we have received. However, it must be noted that our results are with only *g_product_operator()*

function parallelized.

Possibly, there are more avenues to exploit parallelism but we could not get time to check them.

The results with only *g_product_operator()* function parallelized are as follows.

Problem Size	Serial (time)	OpenCL (time)
512	0.26554 s	0.369021 s
1024	0.038892 s	0.351882 s
2048	0.390686 s	0.055191 s

As can be seen from these results, we got a lot of random times. These times reported here are average times over a set of runs. We could not clearly understand the reason for the trends.