

Final thesis

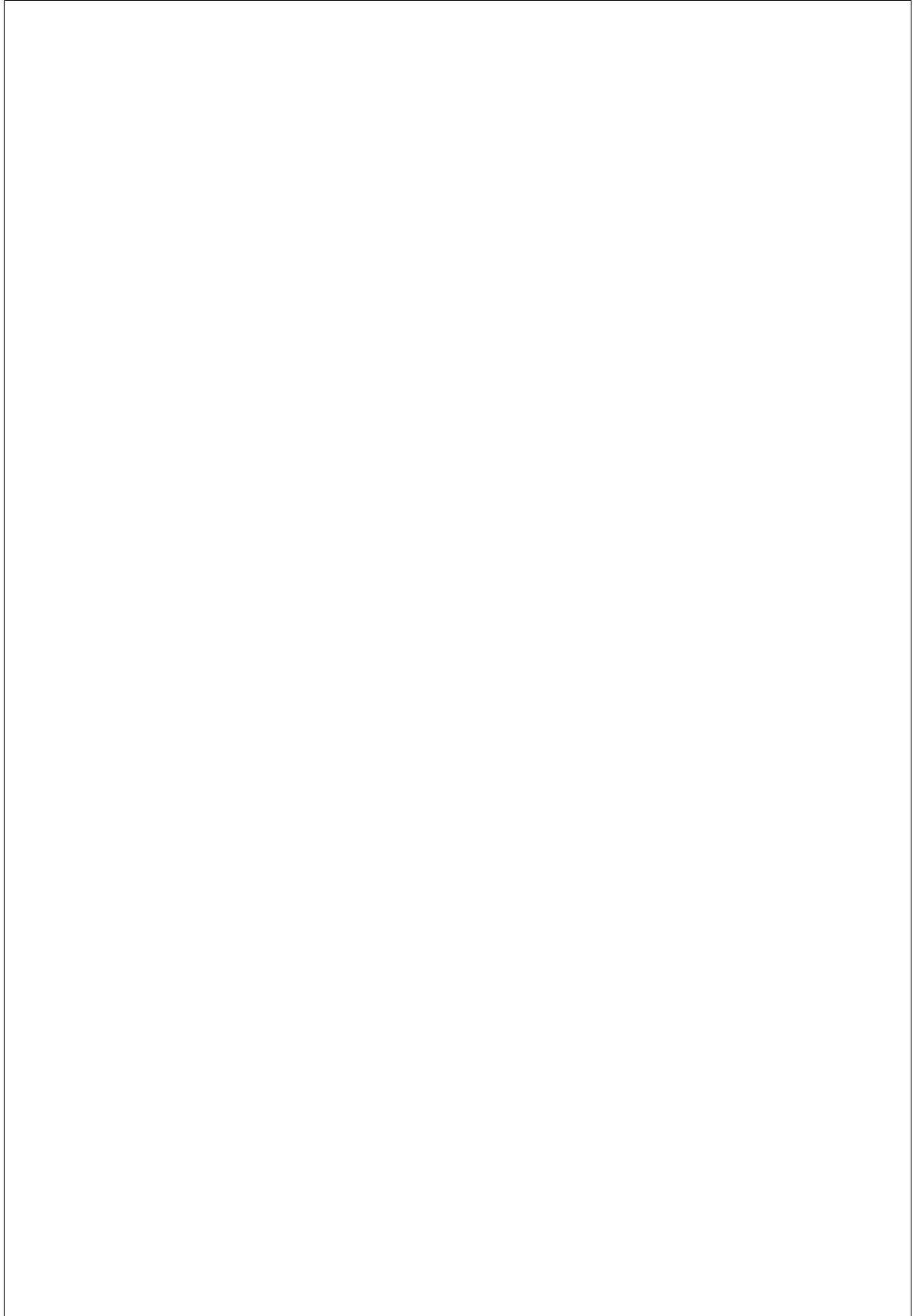
**Comparative study of parallel
programming models for multicore
computing**

by

Akhtar Ali

LITH-IDA-EX-2013/039

2013-06-20



Final thesis

**Comparative study of parallel
programming models for multicore
computing**

by

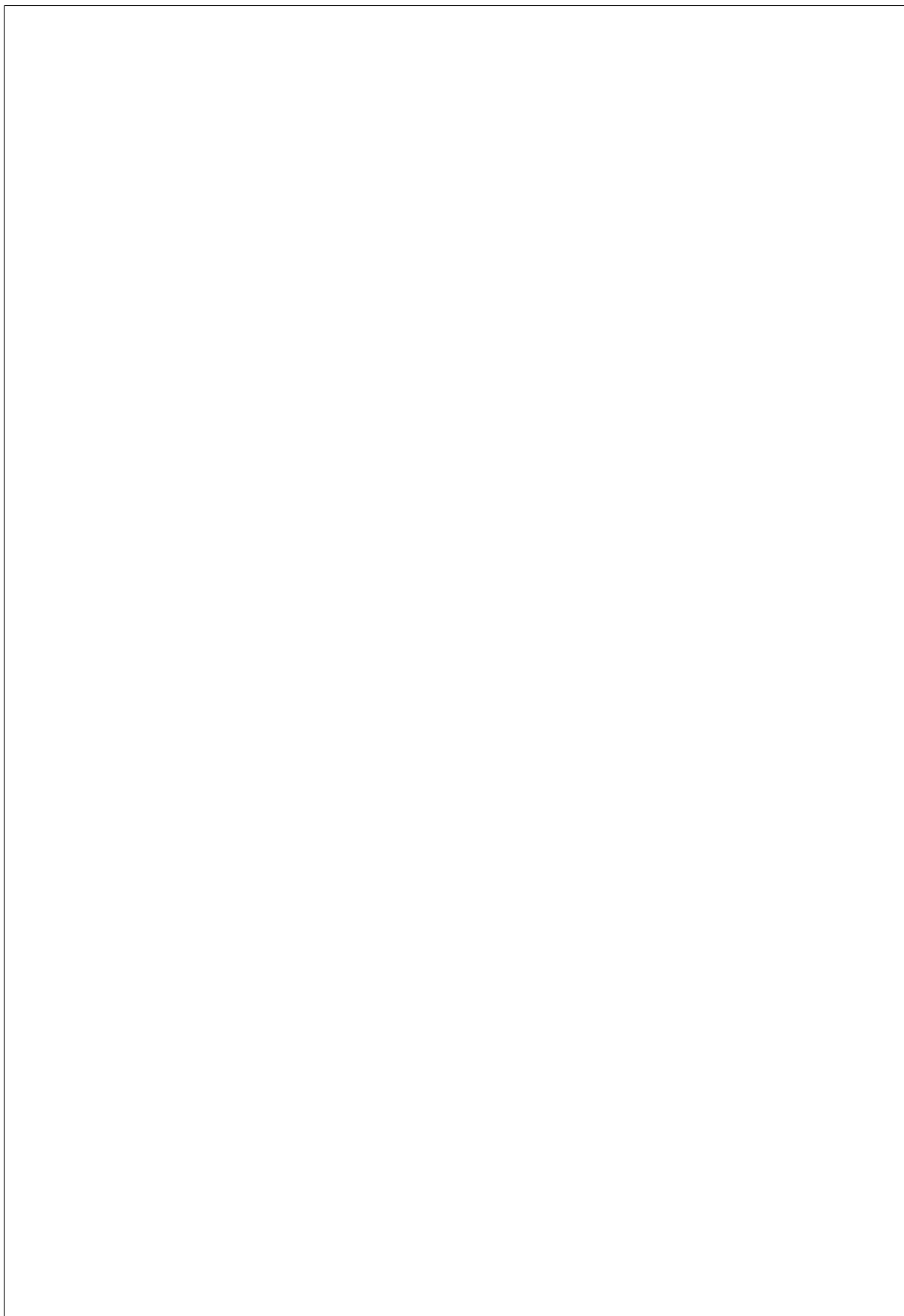
Akhtar Ali

LITH-IDA-EX-2013/039

2013-06-20

Supervisor: Usman Dastgeer

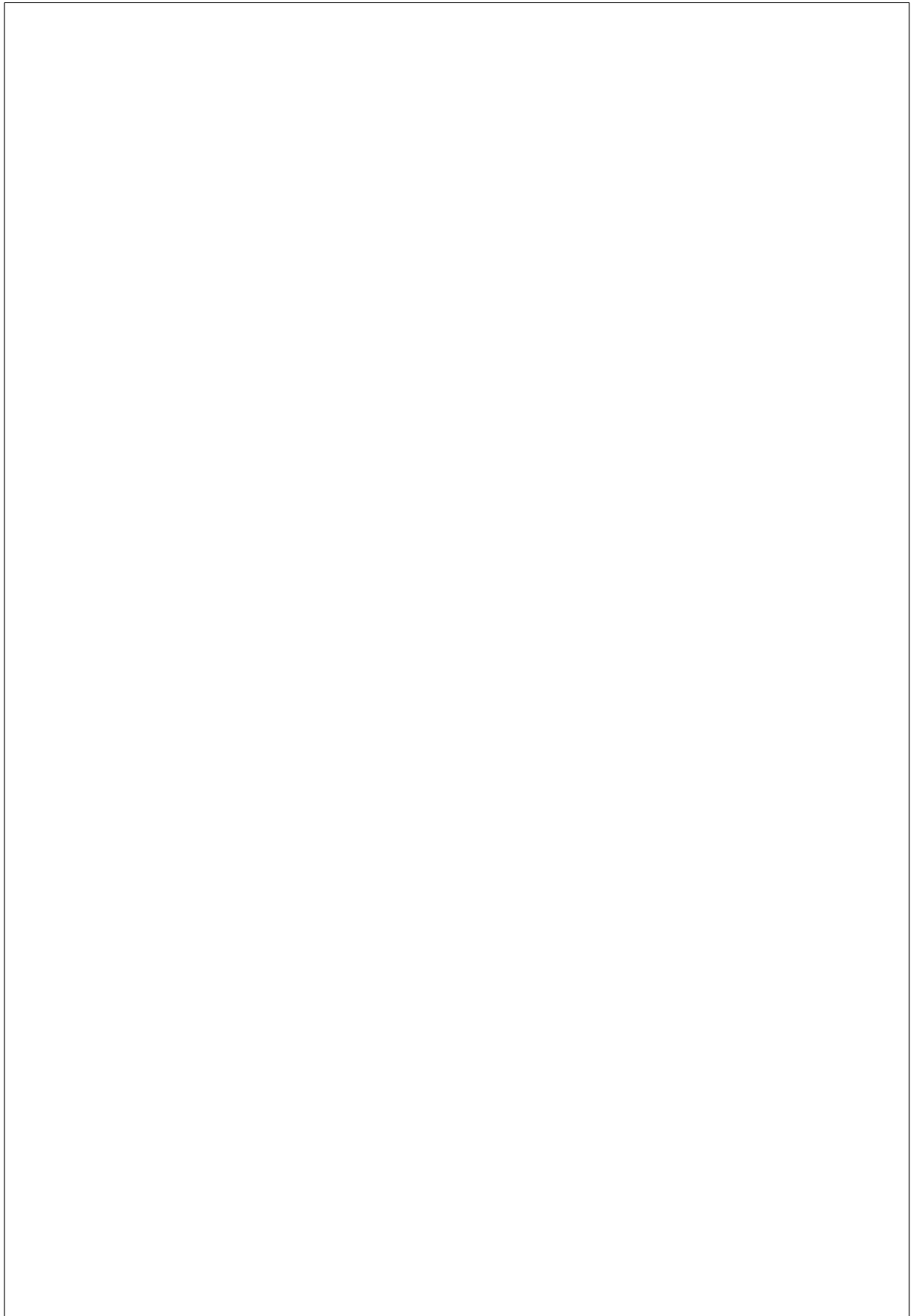
Examiner: Christoph Kessler



Abstract

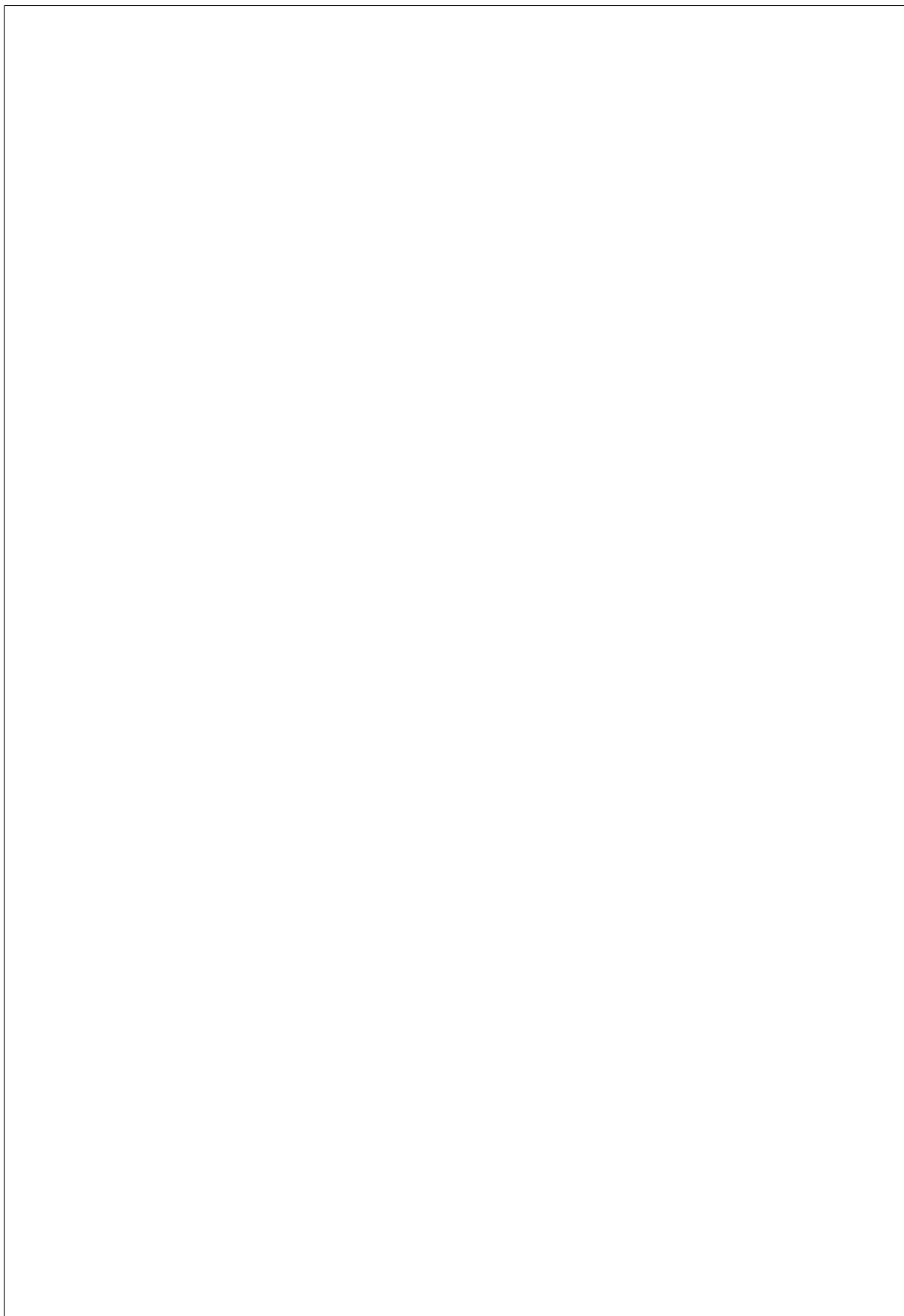
Shared memory multi-core processor technology has seen a drastic development with faster and increasing number of processors per chip. This new architecture challenges computer programmers to write code that scales over these many cores to exploit full computational power of these machines. Shared-memory parallel programming paradigms such as OpenMP and Intel Threading Building Blocks (TBB) are two recognized models that offer higher level of abstraction, shields programmers from low level details of thread management and scales computation over all available resources. At the same time, need for high performance power-efficient computing is compelling developers to exploit GPGPU computing due to GPU’s massive computational power and comparatively faster multi-core growth. This trend leads to systems with heterogeneous architectures containing multicore CPUs and one or more programmable accelerators such as programmable GPUs. There exist different programming models to program these architectures and code written for one architecture is often not portable to another architecture. OpenCL is a relatively new industry standard framework, defined by Khronos group, which addresses the portability issue. It offers a portable interface to exploit the computational power of a heterogeneous set of processors such as CPUs, GPUs, DSP processors and other accelerators.

In this work, we evaluate the effectiveness of OpenCL for programming multi-core CPUs in a comparative case study with two CPU specific stable frameworks, OpenMP and Intel TBB, for five benchmark applications namely matrix multiply, LU decomposition, image convolution, Pi value approximation and image histogram generation. The evaluation includes a performance comparison of the three frameworks and a study of the relative effects of applying compiler optimizations on performance numbers. OpenCL performance on two vendor-dependent platforms Intel and AMD, is also evaluated. Then the same OpenCL code is ported to a modern GPU and its code correctness and performance portability is investigated. Finally, usability experience of coding using the three multi-core frameworks is presented.



Acknowledgements

I would like to thank my supervisor Usman Dastgeer and examiner Christoph Kessler and appreciate their timely support during my work and an early revision to my report and results. I acknowledge the funding by EU FP7, project PEPPHER, which gave me the opportunity to travel to MULTI-PROG’12, Paris, to present this work. I am also thankful to the NSC Neolith, Triolith and IDA Fermi teams for their support to setup the needed environment on these machines and lending me these computing resources.



Contents

1	Introduction	1
1.1	Background	1
1.2	Related Work	2
1.3	Our Study	3
1.4	Overview of the thesis	3
2	Frameworks	4
2.1	OpenMP	4
2.2	Intel TBB	6
2.3	OpenCL	8
3	Benchmark Applications	11
3.1	Matrix Multiplication	11
3.2	LU Decomposition	12
3.3	Image Convolution	13
3.4	Histogram Generation	14
3.5	Pi Approximation	15
4	Performance Evaluation	17
4.1	Environment	17
4.2	Performance	17
4.2.1	Matrix Multiplication	19
4.2.2	LU Decomposition	21
4.2.3	Image Convolution	23
4.2.4	Pi Calculation	25
4.2.5	Histogram Generation	28
4.2.6	Summary	28
4.3	Scalability	29
4.3.1	Matrix Multiplication	30
4.3.2	LU Decomposition	31
4.3.3	Image Convolution	32
4.3.4	Histogram Generation	33
4.3.5	Pi Calculation	34
4.3.6	Summary	34

5	OpenCL Platforms Evaluation	36
5.1	AMD and Intel OpenCL	36
5.1.1	Matrix Multiplication	38
5.1.2	LU Decomposition	39
5.1.3	Histogram Generation	40
5.1.4	Image Convolution	41
5.1.5	Pi Calculation	43
5.1.6	Summary	44
5.2	Code/Performance Portability	45
6	Usability Evaluation	49
6.1	Usability	49
7	Related Work	51
8	Discussion and Conclusion	53
8.1	Discussion	53
8.2	Conclusion	54
8.3	Future Work	54
	Appendices	56
	Appendix A Matrix Multiplication	57
A.1	OpenMP	57
A.2	TBB	58
A.3	OpenCL	59
	Appendix B LU Decomposition	60
B.1	OpenMP	60
B.2	TBB	61
B.3	OpenCL	62
	Appendix C Convolution	64
C.1	OpenMP	64
C.2	TBB	65
C.3	OpenCL	66
	Appendix D Histogram Generation	68
D.1	OpenMP	68
D.2	TBB	68
D.3	OpenCL	70
	Appendix E Pi Calculation	71
E.1	OpenMP	71
E.2	TBB	71
E.3	OpenCL	72

Chapter 1

Introduction

1.1 Background

Computational power of single processor machines deviated recently from the exponential curve of Moore’s law due to physical limiting factors such as clock speed, heat/power problem, limited instruction-level parallelism and memory access bottleneck. These limitations associated with the conventional single processor technology gave birth to the notion of multi and many-core machines sharing main memory to keep up with increasing computational power needs. But this shift of hardware technology from single core to many cores challenged software developers to write programs with parallelism to exploit all available cores. Operating systems had multi-threading support for many years but this was in essence concurrent processing using a single processor switching mechanism rather than actual parallel computing. With the advent of multi-core technology, the operating system scheduler though assigns different applications to different cores to enhance performance of the overall system but this is only helpful in an environment with a large number of applications and the scheduler cannot scale single algorithm workload on these many cores. Programming techniques and compiler optimizations such as loop unrolling, auto-vectorization and adapting to cache hierarchy of the target hardware help to accelerate performance of single application [20] but to a limited extent and does not exploit the collective massive power of all the cores of multi-core technology. POSIX Threads (PThreads) is one portable set of multi-threading interface developed by IEEE committees in charge of Portable Operating System Interface (POSIX). But PThreads engage programmers too much in thread management activities which makes it tedious and error prone. Programmers, on the other hand, are used to focus on the problem domain of their applications for years and they do not want to be involved too much with performance tuning or thread management activities. Parallel programming paradigms abstracting thread management activities evolved during this time to help

developers write algorithms that scale with available cores. We discuss two of these well recognized models for programming multi-core processors in our work, namely, Open Multi-Processing (OpenMP) and Intel’s Threading Building Blocks (TBB). OpenMP is a parallelization framework that extends C/C++ and Fortran compilers by adding a set of compiler directives and environment variables to express parallelism. These directives enable multi-threaded code generation at compile time [14]. Intel’s TBB is rather a runtime-based model [18], it is a parallelism library to C++. It uses generic programming and expresses parallelism within template classes and functions. Graphics processing units (GPUs) have recently been utilized in general purpose high performance computing because of their specialized architecture suitable for data parallel applications along with their high arithmetic intensity and faster multi-core growth compared to CPUs. GPU vendors started providing programming frameworks for their graphics processors such as CUDA from Nvidia and Brook+ from AMD for general-purpose computing. But this led to a code portability problem among GPUs and also across CPUs and GPUs since the language constructs of GPGPU computing evolved from graphical APIs which are different to those for conventional CPU programming. With growing focus on parallel computing, an environment of heterogeneous hardware architectures, sometimes working in collaboration to carry out bulky applications, was inevitable and thus the call for a multi-core programming framework offering a uniform interface to this heterogeneous set of architectures was realized. This need gave birth to Open Computing Language (OpenCL) standard API that is based on the ISO C99 standard and abstracts the underlying hardware and enables developers to write code that is portable across different shared-memory architectures. There are several implementations of the OpenCL standard from different vendors such as AMD, Intel, Nvidia and Apple.

1.2 Related Work

Individually these three frameworks have been studied extensively such as OpenMP [6, 23], TBB [24, 18] and OpenCL [29, 12]. There is still a lot of work focused merely on CPU specific [19, 14] and GPU specific [11, 16, 13] frameworks comparisons. There is an interesting work [27] which discusses programmability and performance of OpenMP and OpenCL. There is also some work [28], which discusses causes of OpenCL performance degradation on CPUs. OpenCL is becoming increasingly important since CPU is the most common kind of processor architecture and OpenCL implementations are available for CPUs by AMD and Intel, for example. Due to its promise for code portability, OpenCL could provide the programming foundation for modern heterogeneous systems. An important study in this direction is [20] examining overhead, scalability and usability of five shared memory parallelism frameworks including OpenCL on a single 2D/3D image registration application. We extend this topic in [chapter 7](#), where we compare our work

with these studies.

1.3 Our Study

In our work, we choose OpenMP and TBB as popular representatives of CPU specialized frameworks and compare these with OpenCL on five benchmark applications. We do the evaluation in the following aspects:

- **OpenCL as an alternative to Intel TBB and OpenMP for programming multicore CPUs:** How does it compare in terms of performance, scalability and compiler support.
- **OpenCL specific issues:** Such as code and performance portability, performance implications of different vendor’s OpenCL implementations, CPU-specific versus GPU-specific optimizations for an OpenCL algorithm and performance implications of these optimizations.
- **Usability:** Following an empirical approach, we look into the programmability aspect of these frameworks when compared with each other since it also plays a vital role in adopting certain frameworks.

1.4 Overview of the thesis

This thesis report is organized as follows. We present short description of the three frameworks in [chapter 2](#) and that of the benchmark applications in [chapter 3](#). Then we present the performance and scalability results comparison in [chapter 4](#). In [chapter 5](#), we compare the Intel and AMD platforms on these applications. Usability evaluation is done in [chapter 6](#). We compare and contrast our work with the related work in [chapter 7](#) followed by a general discussion and conclusion of the thesis work in [chapter 8](#). We put kernel codes of the three frameworks for all these applications in the Appendices.

Chapter 2

Frameworks

Three multicore parallel programming frameworks are used in this study. Two of them, namely OpenMP and Intel Threading Building Blocks, are CPU specialized frameworks. OpenCL, on the other hand, is used to write programs that run across a set of heterogeneous architectures. In this chapter, we present a brief introduction to these frameworks.

The serial code for one application kernel, matrix multiply, is given here for reference which multiplies *mat1* with *mat2* to get resultant matrix *mat3*. The kernel parallelized versions of this short code in these frameworks are given in corresponding sections.

Listing 2.1: Serial matrix multiplication

```

1 for (int k = 0; k < size; ++k)
2   for (int i = 0; i < size; ++i)
3     for (int j = 0; j < size; ++j)
4       mat3[i*size + j] += mat1[i*size + k] * mat2[k*size + j];

```

2.1 OpenMP

The Open Multi Processing (OpenMP) application programming interface (API) provides parallelizing facilities in C/C++ and Fortran by using pre-processor directives/pragmas, library routines and environment variables and is supported by most compilers today. These directives enable generation of multi-threaded code at compile time [14]. Although there exists commercial implementations of OpenMP on clusters[5], its prime target is shared memory architectures.

It provides a very easy to use high level API for multi-threading computationally intensive data- and task-parallel programs. The directive based approach makes incremental parallelization possible which adds to the ease of use of the OpenMP model. Most OpenMP implementations use thread pools below the fork/join model, which exist throughout execution of the

program and therefore avoid the overhead of repeated thread creation and destruction before/after each parallel region [1]. The number of threads spawned can be controlled using subroutine calls or environment variables. OpenMP **critical** and **lock-unlock** primitives guarantee exclusive access to a thread in a parallel region, which protects against data inconsistencies due to race conditions. The **barrier** is a global synchronization directive which makes the leading threads wait until all other threads reach that particular point and then they continue to proceed executing in parallel after it. Two other synchronization directives, **single** and **master**, allow the corresponding block of code in the scope of these directives to be executed only by one thread in which the latter directive ensures that the executing thread must be the main thread. OpenMP creates a team of threads when it encounters a **parallel** construct and the code block in the scope is executed in parallel by all the threads including the main thread itself. The OpenMP **sections** directive encloses a block of code which can be divided into further blocks each annotated with a **section** directive. It makes these portions of the code to be run on different threads. While **parallel** creates a team of threads, **parallel for** divides the loop iterations among all those threads according to the scheduling policy. One of the loop scheduling policy is **schedule(static[, chunk])**, where chunk size is optional. It divides loop iterations statically into equal chunks and assign them to available threads. By default, each thread gets chunk size equal to the loop size divided by the number of threads. OpenMP **schedule(dynamic[, chunk])** assigns chunk size iterations (default is 1) to each thread as it becomes available at run-time. Another policy **schedule(guided[, chunk])** also allocates loop iterations dynamically, but the number of iterations per thread is reduced exponentially with each allocation. There is an implicit barrier at the end of both **parallel** and **parallel for** blocks. Another important clause is **reduction** which is used with other work-sharing constructs to carry out reduction on a shared variable. The reduction procedure takes a collection of data and reduces it to a single element according to the given operator. There is an implicit **barrier** at the end of each enclosing work-sharing construct, unless another clause **nowait** is specified [23].

Portions of the code that are intended to be multi-threaded must be revised and modified to manage any dependencies before annotating those portions with OpenMP parallelization constructs. No branching into or out of such an annotated block of code is allowed except via the **exit()** statement which ends the program. The directive based approach, support for incremental parallelism and its capability to coexist with other parallelism frameworks makes this model easy to use for parallelizing new as well as existing applications [23].

An OpenMP version of the matrix multiply kernel is given below to show how code blocks are annotated with OpenMP pragmas.

First, we set *nthreads* as maximum number of threads available in the subsequent region through an OpenMP provided function. The first pragma

Listing 2.2: OpenMP matrix multiplication

```

1 omp_set_num_threads(nthreads);
2 int i, j, k;
3 #pragma omp parallel shared (mat1, mat2, mat3) private (i, j, k)
4 {
5     #pragma omp for schedule (static)
6     for(i=0;i<size;i++)
7     {
8         for(j=0;j<size;j++)
9         {
10            for(k=0;k<size;k++)
11                mat3[i*size + j] += mat1[i*size + k] * mat2[k*size + j];
12        }
13    }
14 }
```

line indicates that the enclosing block of code is to be run by multiple (*nthreads*) threads. Keyword **shared** makes the corresponding data variables shared among multiple threads. While **private** makes threads to make private copies of those variables. The second pragma indicates that the outer loops are to be divided among available threads. Scheduling policy **static** makes equal division of outer loop iterations and assigns them to those threads.

2.2 Intel TBB

Intel Threading Building Blocks (TBB) is part of Intel’s technology for parallelism, comprising a template library for C++. This library can be exploited using any compiler supporting ISO C++. It relies on generic programming and supports both task parallelism and data parallelism. TBB’s idea of parallelism is essentially object-oriented since the parallelizable code is encapsulated into template classes in a special way and then invoked from the main program.

Instead of dealing directly with low-level heavy threading constructs, which is tedious and error prone, TBB provides a high-level abstraction to the raw threads. Users have to specify logical parallelism and the TBB runtime library maps it into threads ensuring efficient usage of available resources with less programming effort [24].

Intel TBB supports coarse-grained task parallel programming but it mainly focuses on fine-grained data parallelism. Task parallel programming requires to break a program up into many manageable functional blocks and then they are assigned to different threads. But this technique does not scale well since the number of cores in the system and, therefore, the available threads increases with time while there is a fixed number of functional blocks in the program. In case of a data parallel solution, the performance

usually enhances with the addition of more processors.

Intel TBB relies on generic programming. Traditional libraries used to specify interfaces in terms of base classes or specific types. The C++ Standard Template Library (STL) is one example of generic programming. In STL, interfaces are specified by requirements on types. Since TBB templates specify requirements on generic types, not particular types, this makes it adapt to different data representations and deliver good performance algorithms with broad applicability.

It also provides concurrent containers which manage multiple threads' access and updating of elements at the same time. Concurrent containers, e.g., `concurrent_queue`, `concurrent_vector`, `concurrent_hash_map`, makes Intel TBB even more suitable for data parallel programming. Standard template library (STL) containers need to be wrapped with a mutex which ensures that only one thread can operate on a container simultaneously. But since concurrency is compromised this way, the speed-up gained is minimal. TBB's concurrent containers offer concurrency using either of the following techniques [31]:

- Fine-grained locking; instead of locking the whole container when a thread needs to access it, it locks only that portion of the container which it has to access. This way, as long as different threads need to operate on different elements of the container, they can access the container concurrently.
- Lock-free techniques; threads account and correct for the effects of other interfering threads.

Using these techniques to take care of any inconsistencies, the TBB system incurs some extra overhead with these concurrent containers compared to the use of regular STL containers. Therefore, it is recommended to use them only when they can bring some additional speed-up compared to the regular containers. Besides offering concurrent containers, TBB also provides mutual exclusion through mutexes and locks.

In contrast to other models of parallelism suggesting static division of work, TBB rather relies on recursively breaking a problem down to reach the right granularity level of parallel tasks and this dynamic scheduling technique shows better results than the former one in complex situations. It also fits well with a task stealing scheduler [24]. TBB is not designed to solve all threading situations, therefore, it has the capability to coexist with other threading models.

As an example TBB class encapsulating parallelizable code, a TBB version of the matrix multiply kernel is given below.

This class can be instantiated and invoked from the main program with following line of code.

```
1 parallel_for(blocked_range<int>(0,size), matmul(mat1, mat2, mat3));
```

Listing 2.3: TBB function object class for matrix multiplication

```

1  class matmul
2  {
3      float* a;
4      float* b;
5      float* c;
6  public:
7      matmul(float* mat1, float* mat2, float* mat3) :
8          a(mat1), b(mat2), c(mat3) {}
9      void operator()(blocked_range<int>& r) const
10     {
11         int last = r.end();
12         int i, j, k;
13         for (i = r.begin(); i != last; ++i)
14         {
15             for (j = 0; j < size; ++j)
16             {
17                 for (k = 0; k < size; ++k)
18                 {
19                     c[i*size + j] += a[i*size + k] * b[k*size + j];
20                 }
21             }
22         }
23     }
24 };

```

Nested loops of matrix multiplication are moved to a templated class *matmul*. This class takes the integer range of iterations to work on, and takes three matrices in the constructor and assign its own pointers to them. Then the function call operator is overloaded which makes it a function object, and the matrix multiplication code is put in it. The `parallel_for` internally splits the given range $[0, \text{size})$ into multiple blocks which correspond to the division of the outermost for loop of the application.

2.3 OpenCL

OpenCL is the first royalty-free, open standard for programming modern heterogeneous architectures. An OpenCL program can run on multicore processors, graphics cards and has a planned support for DSP like accelerators [20]. The kernels of these programs are just-in-time (JIT) compiled during runtime which prevents dependencies on the instruction set and supporting libraries and thus enables utilization of the underlying devices' latest software and hardware features such as SIMD capability of hardware [29].

In OpenCL terminology, a program runs on an OpenCL device (CPU, GPU etc.) that holds compute units (one or more cores) which further may include one or more single-instruction multiple-data (SIMD) processing elements. Besides hiding threads, OpenCL goes a step forward by abstracting

hardware architectures and provides a common parallel programming interface. It creates a programming environment comprising a host CPU and connected OpenCL devices which may or may not share memory with the host and might have different machine instruction sets [29]. A program can simply be divided into two parts; the *hostprogram* running on the OpenCL host machine and the *kernel* part that is enqueued by the host to a specific device which is then scaled onto available compute-units/cores of that device. A host program enqueues a command to a command-queue which is attached to a compute device. There are three types of commands; kernel execution, memory management and synchronization. A kernel command executes a kernel on a device, a memory command manages a buffer object and a synchronization command puts ordering among these commands. The OpenCL runtime system executes the enqueued synchronization or memory commands directly while it schedules the enqueued kernel commands on its associated compute device. When a kernel is enqueued for execution, an abstract index space is defined that is used to execute the kernel. This index space is called NDRange which is an N -dimensional space where N is either 1, 2 or 3. NDRange is defined by an N -tuple of integers which specifies the size and dimension of the problem domain. An instance of the kernel that executes for each point in this index space is called *workitem* and this is uniquely identified by its global ID (N -tuple). One or more work-items grouped together makes a *workgroup* which is a more coarse-grained decomposition of the index space and it is also identified by a unique ID called workgroup ID. A workgroup itself assigns a local ID to each work-item within it. This way a work-item can also be identified by a combination of local ID plus workgroup ID [26, 17].

There are two synchronization domains in OpenCL; work-items in a workgroup and commands enqueued to the command queue. A barrier inside a workgroup enforces synchronization among the work-items for that workgroup but there is no mechanism available for synchronization among workgroups. While a command queue barrier is used to synchronize commands inside a particular command queue and events are used to synchronize different command queues. [17]

The OpenCL memory model consists of host side memory and four types of memories on the device side: global, constant, local and private. Global memory allows read/write to all work-items in all workgroups but has high access latency so its use must be kept minimal. Constant memory is a part of global memory which retains its constant values throughout kernel execution. Local memory can be used to make variables shared for a workgroup as all work-items of the workgroup can read/write to it. Private memory is only visible to individual work-items and each can modify or read only its own visible data. GPUs have on-chip Local Data Share (LDS) and a separate private memory bank with each compute unit which is OpenCL local and private memory respectively. CPUs on the other hand implement private memory as register/L1 cache, local memory as L2/L3 cache, global

as main memory, and constant OpenCL memory as main memory/cache but their exact implementations are architecture and OpenCL implementation dependent [3]. The host program running on the host CPU creates memory objects in global memory using OpenCL APIs while en-queueing memory commands operating on these memory objects which can be synchronized using command-enqueue barriers or context events [12]. AMD, Intel, IBM, Nvidia and Apple are some well-known vendors who have implemented the OpenCL standard.

An OpenCL kernel example is given below for the matrix multiply application. This is the simplest kernel translating serial code into OpenCL code without any extra optimizations tuned into it.

Listing 2.4: OpenCL matrix multiplication kernel

```

1  __kernel void matmul(__global float* A, __global float* B,
2                      __global float* C, int widthA, int widthB)
3  {
4      int IDx = get_global_id(0);
5      int IDy = get_global_id(1);
6      float sum = 0.0f;
7      for (int i = 0; i < widthA; ++i)
8      {
9          float tempA = A[IDy*widthA + i];
10         float tempB = B[i*widthB + IDx];
11         sum += tempA * tempB;
12     }
13     C[IDy*widthA + IDx] = sum;
14 }
```

The `__global` indicates that all the three matrices are in OpenCL global memory. Each *workitem* takes a unique combination of *IDx* and *IDy* using OpenCL functions as above. Every *workitem* then uniquely accesses its corresponding elements in the two matrices A and B in a loop and calculates a resultant private *sum* value. Then all these *workitems* write their calculated private *sum* value to the corresponding value in resultant matrix C in the global memory.

Chapter 3

Benchmark Applications

Image processing and linear algebra computations are ubiquitous in science and engineering and have many applications in e.g., graphics programming, artificial intelligence and data mining. For these reasons linear algebra solutions are studied extensively and standard libraries and subroutines are available such as BLAS (Basic Linear Algebra Subroutines), Intel’s MKL (Math Kernel Library) and LAPACK (Linear Algebra Package) [4]. Common problem solving in this area involves matrix computations. These computations comprise a large amount of data and calculations which follow certain access patterns that can be represented by (nested) looping structures. Thus these problems require powerful computing resources. They are, therefore, good candidates for parallel computing frameworks and machines.

Five benchmarks, namely matrix multiplication, LU factorization, 2D Gaussian image convolution, Pi value approximation and histogram generation are chosen for investigating parallelism frameworks.

3.1 Matrix Multiplication

Matrix computations such as matrix multiplications are widely used in scientific computing [21], such as digital image and video processing applications. This operation is a standard problem in numerical linear algebra and serves as a building block to problem solving throughout scientific computing [25].

This fundamental operation is a bottleneck for many important algorithms and therefore, many researchers have been trying to optimize this operation. In this study, it is taken for comparing the efficiency of the stated parallelism frameworks.

A matrix multiplication algorithm in C/C++ uses nested loops with three levels each iterating through a different index. This leads to $3! = 6$ different ways by changing the order of execution of these three loops. If these different versions are denoted by the order of indices, we will get *ijk*, *jik*, *kij*, *ikj*, *jki* and *kji* versions. The matrix multiplication algorithm

with loop order kij is a moderate algorithm with respect to cache misses [32, 33] for serial multiplication.

In case of parallel implementations, we parallelize with respect to rows of the resultant matrix, i.e., with respect to i , so we choose ijk making i as the outermost loop as shown below. Actual parallel kernels are given in Appendix A.

Listing 3.1: Make the outermost loop parallel

```

1 for (int i = 0; i < size; ++i)
2   for (int j = 0; j < size; ++j)
3     for (int k = 0; k < size; ++k)
4       mat3[i*size + j] += mat1[i*size + k] * mat2[k*size + j];

```

3.2 LU Decomposition

Solving linear system of equations is another ubiquitous numerical computation method used in science and engineering. This computation problem can be solved using matrix computation as a triangular factorization of its coefficient matrix, which is termed as LU decomposition [7].

This algebraic process converts any matrix A into a product of two other matrices, a lower triangular matrix L and an upper triangular matrix U , thus the process is named as LU factorization. The major application of this process is to solve linear equation. Besides, it can also compute the determinant and the inverse of a matrix [9].

Consider a system of linear equations. Let A be an $n \times n$ matrix representing coefficients of each equation, x is a $n \times 1$ vector with the unknowns of the system and b another $n \times 1$ vector which represents the right hand side of the system of equations. Thus this system in matrix form becomes

$$Ax = b$$

Once LU decomposition is carried out, the above equation can be written as

$$LUx = b$$

This equation can be divided into two simpler equations

$$Ux = y$$

$$Ly = b$$

Both of these equations are similar and can be solved by simple substitution of variables since L and U are lower and upper triangular equations respectively [9].

We choose a simple variant of LU decomposition called the Cholesky algorithm. Cholesky decomposition solves symmetric matrices more efficiently. By definition, for a symmetric matrix A , $A(i, j) = A(j, i)$. The serial code for Cholesky LU decomposition is given below.

Listing 3.2: Cholesky algorithm code

```

1  for(int k=0; k<size-1; k++)
2  {
3      for(int i=k+1; i<size; i++)
4          A[i*size + k] = A[i*size + k]/A[k*size + k];
5
6      for(int j=k+1; j<size; j++)
7          for(int i=k+1; i<size; i++)
8              A[i*size + j] -= A[i*size + k] * A[k*size + j];
9  }
```

The first i loop divides the k column by the pivot element, i.e., $A(k, k)$, and the next i, j loops update elements along the column and row respectively. We can not parallelize the outer k loop since each next iteration uses updated values of the matrix from the previous iteration. We try to combine the i loops, which first divides by the pivot element and then continues updating along the i row (j iteration). The new code is given below while corresponding parallel kernel codes are given in Appendix B.

Listing 3.3: Cholesky algorithm re-structured code

```

1  for(k=0; k<size-1; k++)
2  {
3      for(i=k+1; i<size; i++) // Parallelize from here
4      {
5          A[i*size + k] = A[i*size + k] / A[k*size + k];
6          for(j=k+1; j<size; j++)
7              A[i*size + j] -= A[i*size + k]*A[k*size + j];
8      }
9  }
```

3.3 Image Convolution

Digital Image Processing (DIP) is used in a wide range of applications such as computer vision, medical areas and meteorology fields. This can be done for different purposes, e.g., image restoration, enhancement, addition of certain effects and filtering. Image convolution is the first application chosen from the area of DIP since it is one of the most important ones in this area [8].

Convolution is used to apply different kinds of effects such as blur, sharpen, emboss [15], edge detection, image smoothing, template matching [8]. The mask/filter is applied to the input image which generates a filtered/convolved output image characterizing the used filter [8]. Gaussian filter is used in our convolution which can be graphed as the famous “bell-shaped curve”. Both the image and the filter are represented by square matrices. A Gaussian filter is separable, i.e., symmetric around its center and can be decomposed into 1D row vector and column vector filters that can be applied to the image in horizontal and vertical directions respectively. This has the advantage of involving $n + n$ multiplications compared to $n \times n$

multiplications in the 2D case where n is the filter size in each dimension. In addition, it reduces the number of idle threads and memory accesses of each pixel [22]. Therefore, we decompose our filter in a column and a row vector. The serial code for these two separated operations is given below. We parallelize from the outer loop in both cases and corresponding parallel code follows in Appendix C.

Listing 3.4: Convolution separable kernels

```

1 void convRow( image_t* src, image_t* dst, float* filterW, int filterR, int
   imageW, int imageH)
2 {
3     int x, y, k, d;
4     float sum;
5     for(y = 0; y < imageH; y++) // Parallelize from here
6         for(x = 0; x < imageW; x++)
7         {
8             sum = 0;
9             for(k = -filterR; k <= filterR; k++)
10             {
11                 d = x + k;
12                 if(d >= 0 && d < imageW)
13                     sum += src[y * imageW + d] * filterW[filterR - k];
14             }
15             dst[y * imageW + x] = sum;
16         }
17 }
18 void convCol( image_t* src, image_t* dst, float* filterW, int filterR, int
   imageW, int imageH)
19 {
20     int x, y, k, d;
21     float sum;
22     for(y = 0; y < imageH; y++) // Parallelize from here
23     {
24         for(x = 0; x < imageW; x++)
25         {
26             sum = 0;
27             for(k = -filterR; k <= filterR; k++)
28             {
29                 d = y + k;
30                 if(d >= 0 && d < imageH)
31                     sum += src[d * imageW + x] * filterW[filterR - k];
32             }
33             dst[y * imageW + x] = sum;
34         }
35     }
36 }

```

3.4 Histogram Generation

Histogram generation is another important application in digital image processing. A histogram is a representation of the frequency distribution of an image. Histograms are used in many fields such as image processing. Here

the intensity spectrum of images can be visualized, compared or modified to know the similarities and differences among different images [19]. For a simple gray-scale image, a histogram generation function maps the frequency of the intensity levels in the image to the gray-level range. Histogram generation of a color image works on the same principle with intensity levels created for different colors.

A matrix of random numbers with values in the range 0-255 is generated which we chose as a representation of a gray scale image for the computation performance comparison of these frameworks. Thus the algorithms fill up 256 bins each representing a gray scale intensity level according to their appearances in the whole matrix. This is a reduction operation on bins values since multiple threads might simultaneously be incrementing the same bin values which they have found in different portions of the image in parallel. Serial code is given below.

Listing 3.5: Histogram serial algorithm

```

1  for(y=0; y<HEIGHT; y++)
2  {
3      for(x=0; x<WIDTH; x++)
4      {
5          int image_value = image[x + y*WIDTH];
6          hist[image_value] += 1;
7      }
8  }
```

In case of OpenMP, all the threads first calculate these bin values in their local memory and then we merge these subhistograms in a OpenMP **critical** section. TBB, on the other hand has a **parallel_reduce** construct which computes parallel reduction over a given range. The function object should be defined with a **join()** method. Each thread calculates a local subhistogram in the overloaded function operator and then these subhistograms are merged in the **join** method. The OpenCL kernel, taken from AMD OpenCL code examples, calculates subhistograms in a similar fashion and then merges them to form block-histograms which are then merged at the host side to generate the final histogram. The code for OpenMP, TBB and OpenCL is given in Appendix D.

3.5 Pi Approximation

Pi approximation computes the area under the curve

$$y = 4/(1 + x^2)$$

between 0 and 1, i.e., integral of the above equation over the interval [0,1]. The N in below code represents the number of points taken to calculate area under the above curve equation. This also controls the precision value in Pi approximation. It ranges from 10000 to 100000000 in our experiments.

Listing 3.6: Pi calculation serial algorithm

```

1 pi = 0.0;
2 double w = 1.0 / N;
3 double local;
4 for(unsigned int i = 0; i < N; i++)
5 {
6     local = (i + 0.5) * w;
7     pi += 4.0 / (1.0 + local * local);
8 }
9 pi *= w;
```

The code for computing the approximate value of Pi is taken from the OpenMP repository [10]. It is a classical test program in any parallel API. The Pi application uses the reduction property in the parallel algorithm in which all threads collectively calculate the final value of Pi. This application is parallelized using these other frameworks which all support ways to solve reduction problems. The OpenMP **reduction** and TBB template function **parallel_reduce** are used. While in case of OpenCL, the looping range is factorized into a small number of loop iterations times number of workitems. Thus each work item has to loop by a factor of actual serial loop size. We vectorize the OpenCL kernel with the vector size 4. This actually reduces looping steps by another factor of 4. The corresponding kernel codes for OpenMP, TBB and OpenCL follow in Appendix E.

Chapter 4

Performance Evaluation

4.1 Environment

Our test applications are parallelized from the sequential C/C++ implementations. Parallelizing the same serial algorithms and keeping the same data structures makes it appropriate to compare speedup of these testing frameworks. Heap memory is used for arrays representing these matrices since stack memory can accommodate matrices of small orders but is not sufficient in case of larger matrix orders. This is done in all cases in order to have the same basis for efficiency comparisons of these different frameworks. A five point mean average performance value is taken in each case in graphs. Unless specified otherwise, the experiments are carried out on Intel Xeon CPU E5345 with 8 cores running at 2.33GHz. AMD SDK-v2.4 supporting OpenCL 1.1 is used with Intel compiler version 10.1.

4.2 Performance

In this section, we do performance evaluation for OpenCL, OpenMP and Intel TBB implementations of each of the five benchmark applications. Furthermore, to investigate effectiveness of compiler optimizations for different frameworks, we have tried different compiler optimization switches with the Intel C++ compiler (icc version 10.1) and compiler optimizations available in the parallelism models. This is done to see the role of compiler optimizations in speeding up since compilers play a vital role in this area. The optimizations carried out by a compiler could have a profound effect on the actual performance. The OpenCL runtime compiler abstracts hardware architecture and provides a common base for programming. To show the effects of compilation on the actual performance, we compare the execution with *disabled* compiler optimizations to the one with *aggressive* compiler optimizations.

For executables with *disabled* compiler optimization, option `-O0` is used during the program compilation. For OpenCL dynamic compilation, there is a function for building a kernel program object as given below,

```
clBuildProgram(program, 0, NULL, options, NULL, NULL)
```

The fourth parameter named *options* can be used to specify the optimization level to OpenCL. To disable optimizations, a similar effect as `-O0` is achieved by specifying the *options* parameter in the above function as the constant character string `cl-opt-disable`. For executables with *aggressive* compiler optimization, option `-O3` is used during the program compilation which consists of option `-O2`¹ plus memory access and loop optimizations, such as loop unrolling, code replication to remove branches and loop blocking to improve cache usage. For OpenCL dynamic compilation, the same effect is achieved by passing the constant character string `cl-fast-relaxed-math`² option to *clBuildProgram* function which is a composite of two other OpenCL optimizations, namely `cl-finite-math-only` and `cl-unsafe-math-optimizations`.

In the following, we discuss speedup comparisons and the effects of compiler optimizations for different benchmark applications.

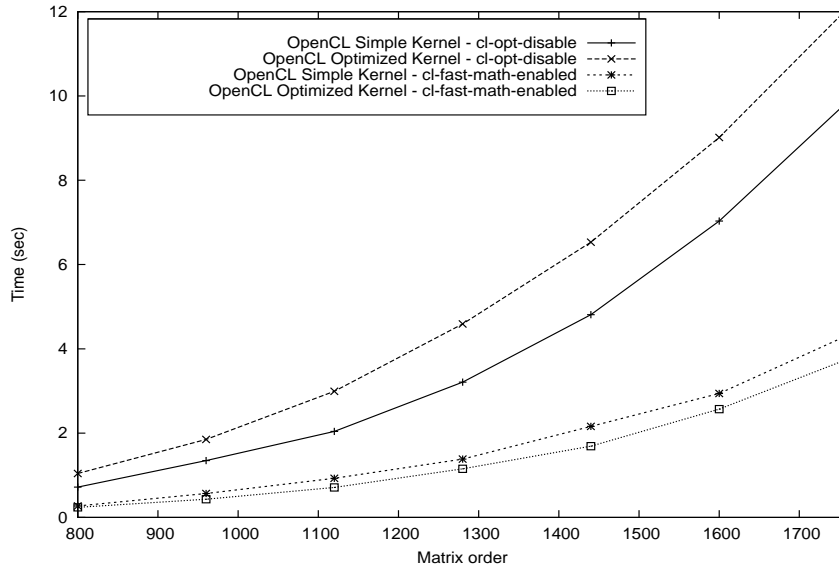


Figure 4.1: Matrix multiplication performance of OpenCL kernels

¹The `-O2` flag enables speed specific optimizations e.g., vectorization.

²This option is used for all applications except the Pi application as it is sensitive to high precision.

4.2.1 Matrix Multiplication

Figure 4.1 shows the matrix multiplication comparison among OpenCL runs with different optimizations tuned into kernel and compiler optimization flags. Two OpenCL kernels are used, one using direct transformation of the application to an OpenCL kernel while another kernel has optimized usage of local and private memories. The optimized kernel shows slightly better performance when compiler optimizations are enabled but performs worse than the normal kernel when they are disabled. This result thus shows no significant performance boost while using local/private memories on OpenCL CPUs.

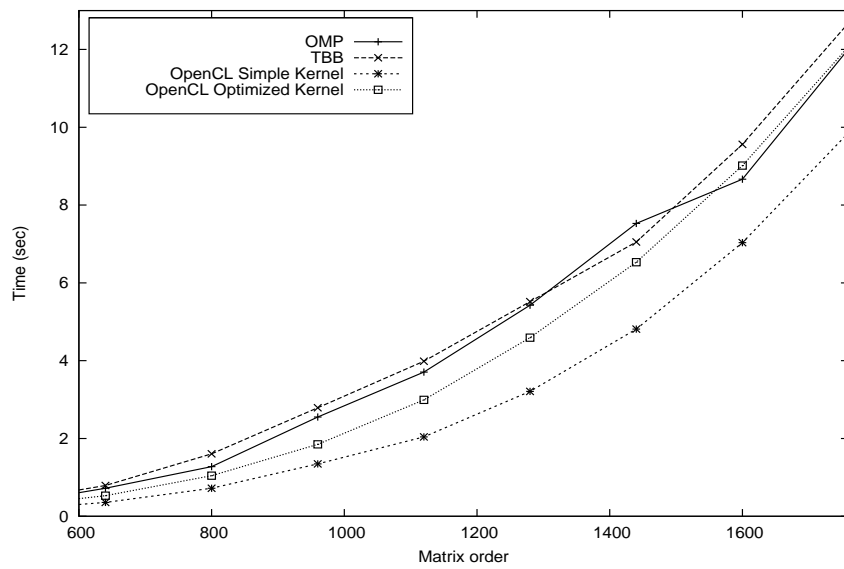


Figure 4.2: Matrix multiplication performance of OMP, TBB, OpenCL - (O0+cl-opt-disable)

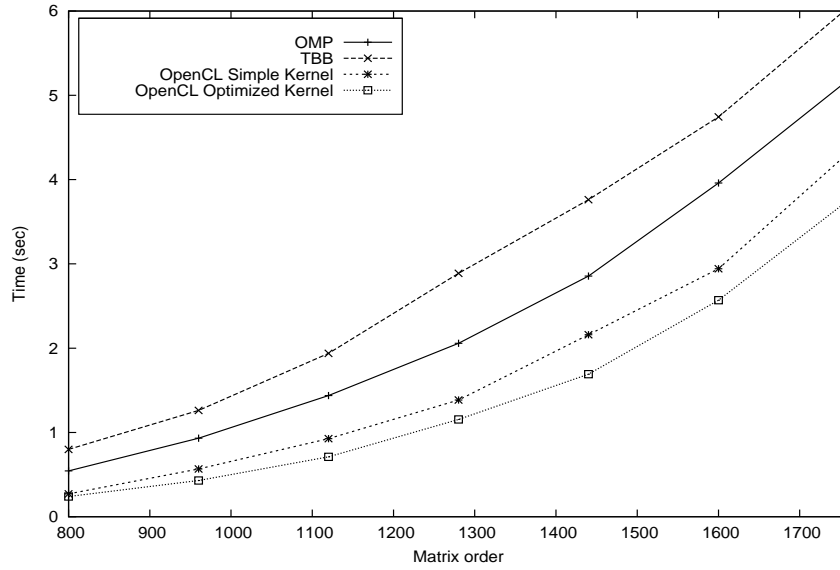


Figure 4.3: Matrix multiplication performance of OMP, TBB, OpenCL - (O3+cl-fast-relaxed-math)

Figure 4.2 and 4.3 shows that OpenCL outperforms the other two models in matrix multiplication at both optimization levels by taking into account all the three frameworks. TBB and OpenMP performance is fairly equal when no compiler optimizations are present, as shown by Figure 4.2, while OpenMP is the winner when compiler optimization support is enabled. Thus OpenMP benefited from compiler support and this can be seen in Figure 4.3 as compared to Figure 4.2.

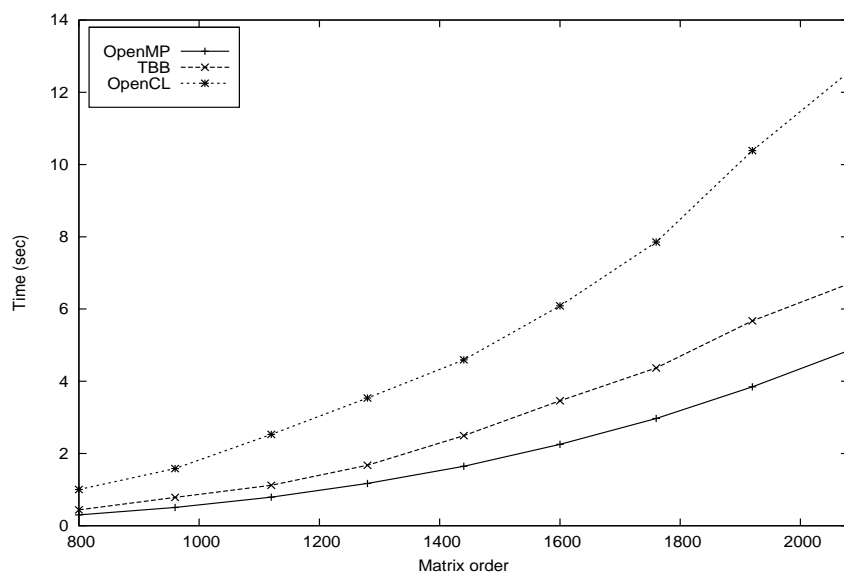


Figure 4.4: LU decomposition performance of OMP, TBB, OpenCL - (O0+cl-opt-disable)

4.2.2 LU Decomposition

Figure 4.4 shows results of the LU factorization application with no compiler support. Here OpenCL is way slower than the other two while OpenMP is faster than TBB comparatively but both curves have little deviation from each other for increasing matrix sizes.

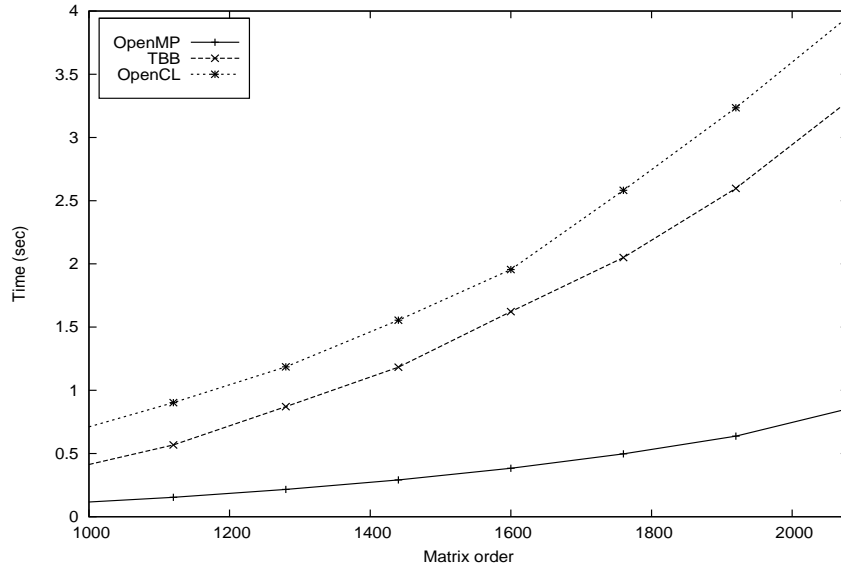


Figure 4.5: LU decomposition performance of OMP, TBB, OpenCL - (O3+cl-fast-relaxed-math)

While in Figure 4.5, which shows results with compiler support enabled, OpenCL is still slowest but closer to TBB in performance while OpenMP is much faster than the other two. The gap between OpenMP and TBB is much wider with the compiler optimizations enabled, which shows that OpenMP benefited more than TBB. It can be seen from both these figures that OpenMP yields the best performance for all inputs in LU factorization while OpenCL shows slowest results comparatively. The rationale behind this could be traced to the kernel algorithm which sets a workgroup along each row of the matrix for synchronization purpose using local memory. The OpenCL runtime should be allowed to choose their optimal workgroup size otherwise. The gap between TBB and OpenMP widens at the aggressive optimization level in Figure 4.5 which means that OpenMP again benefited more from compiler optimization than TBB.

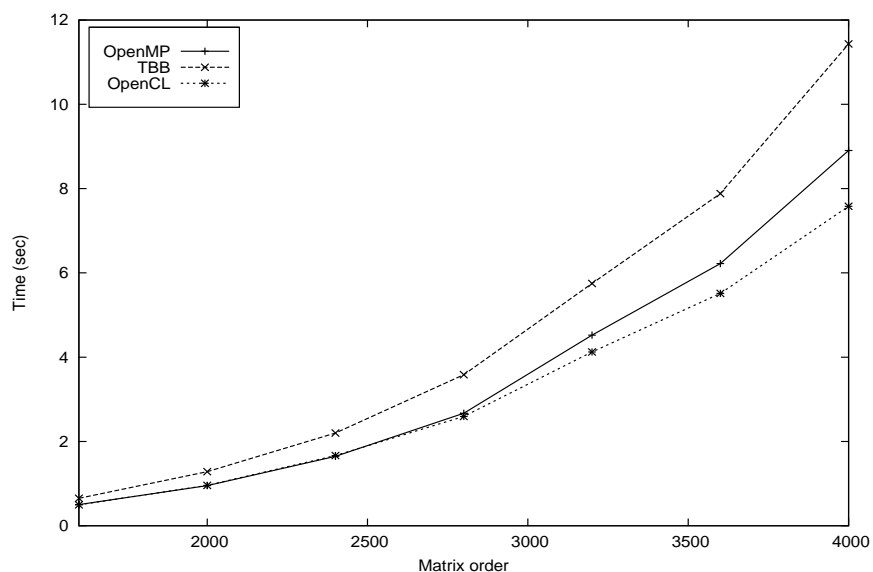


Figure 4.6: Gaussian image convolution performance of OMP, TBB, OpenCL - (O0+cl-opt-disable)

4.2.3 Image Convolution

For 2D image convolution with no compiler support, TBB performs comparatively slower while OpenMP and OpenCL perform equally well with a slightly better OpenCL performance at large input sizes, as shown in Figure 4.6.

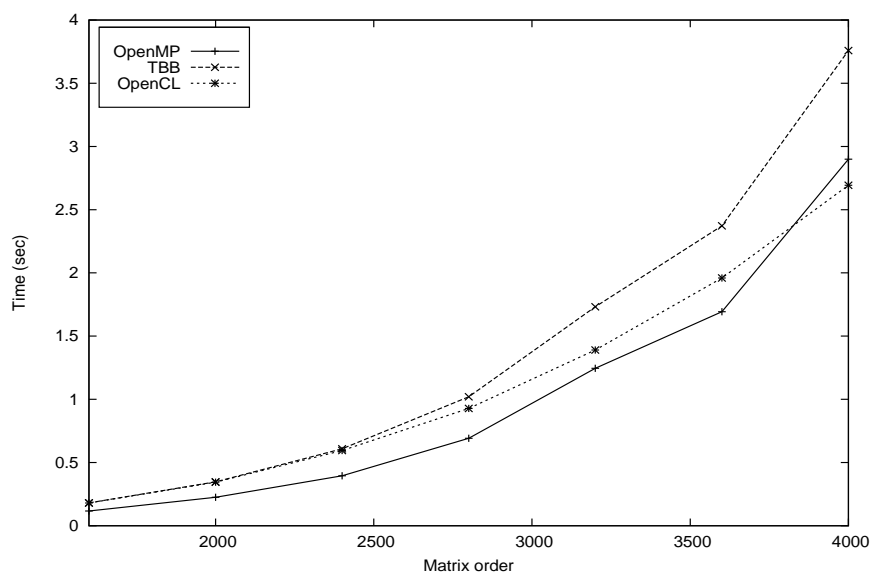


Figure 4.7: Gaussian image convolution performance of OMP, TBB, OpenCL - (O3+cl-fast-relaxed-math)

Figure 4.7 represents performance comparisons with compiler optimization support. It demonstrates that the performance gap among the three frameworks narrows when compiler optimizations were enabled while TBB still is a little slower at high matrix input orders.

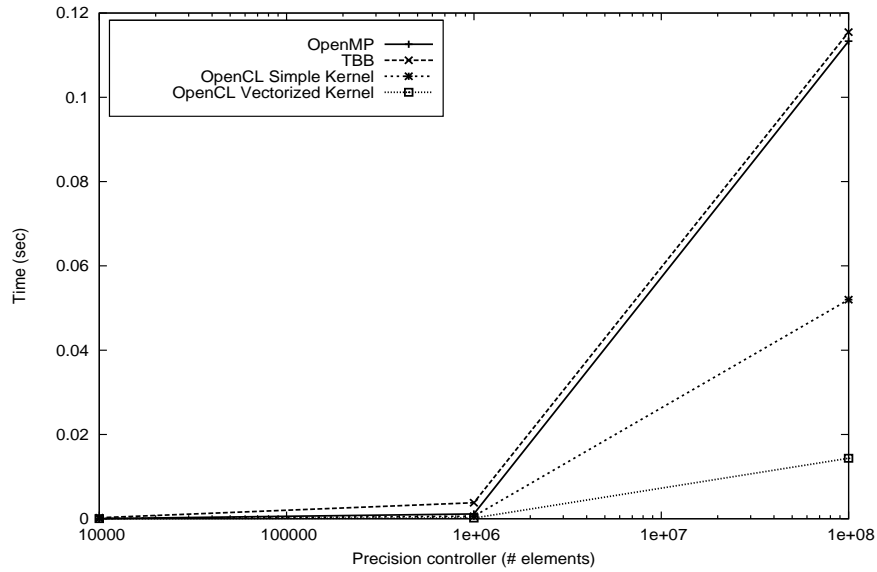


Figure 4.8: PI approximation performance of OMP, TBB, OpenCL - No auto-optimization

4.2.4 Pi Calculation

Pi value approximation uses reduction. In Figure 4.8, OpenMP and TBB present identical performance with no compiler optimizations while OpenCL shows the best performance. There are again two OpenCL kernels used in which OpenCL vectorization is performed in one of the kernels. The graph clearly demonstrates that an explicitly vectorized OpenCL kernel significantly beats the simple OpenCL kernel and all other models in speedup.

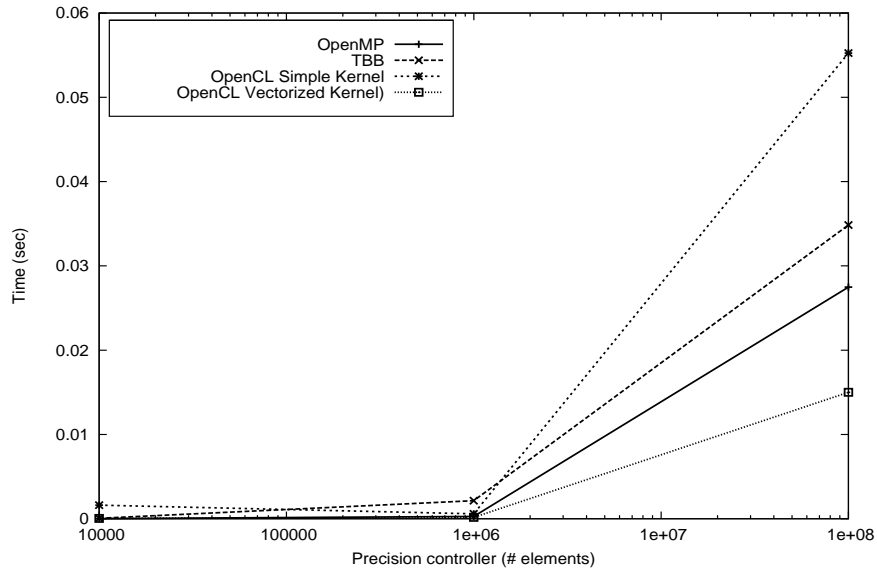


Figure 4.9: PI calculation performance of OMP, TBB, OpenCL - default optimization

The aggressive optimization level, that is, `-O3` support from the Intel compiler and `cl-fast-relaxed-math` from the OpenCL compiler, is not used in the Pi application since this application is sensitive to rounding-off errors and demands high precision. So the second test is done on the compilers' default optimization level and the result is shown in Figure 4.9. When the Intel compiler's default optimization level (`-O2`) was used with the default OpenCL optimization, there is a narrow gap between OpenMP and TBB performances with OpenMP showing again slightly better result while the OpenCL vectorized kernel is still way faster. This shows that enabling explicit vectorization in OpenCL, when possible, enhances speedup on CPUs. It also indicates that OpenMP gained more than TBB with the compiler support enabled.

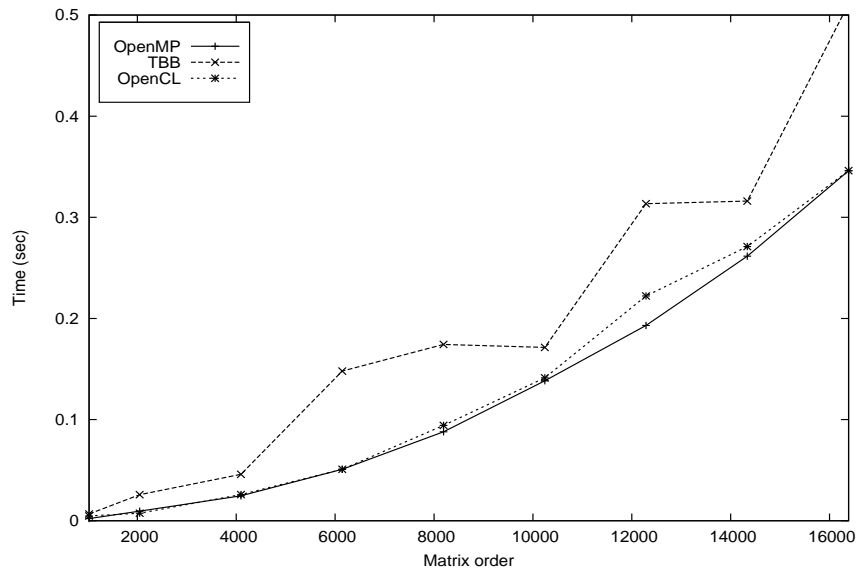


Figure 4.10: Histogram generation performance of OMP, TBB, OpenCL - (O0+cl-opt-disable)

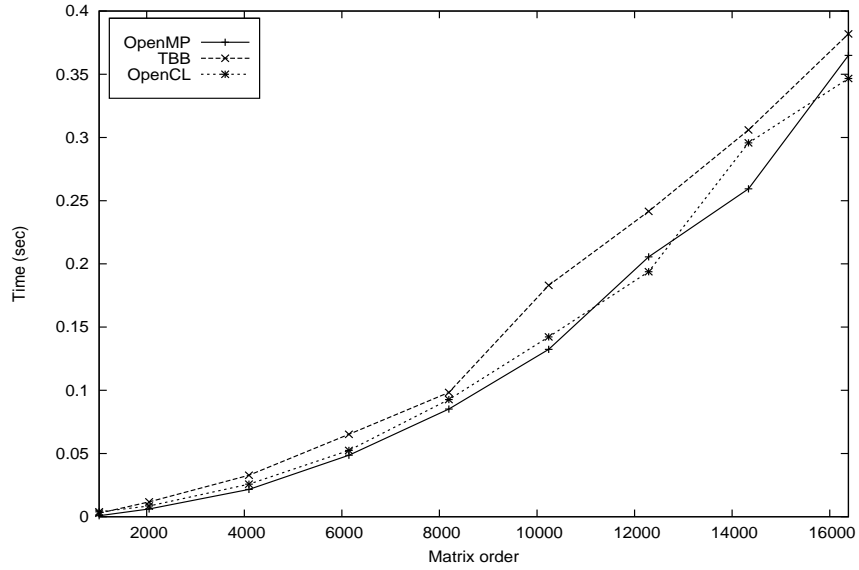


Figure 4.11: Histogram generation performance of OMP, TBB, OpenCL - (O3+cl-fast-relaxed-math)

4.2.5 Histogram Generation

Histogram generation is also a reduction application. Its performance graph with no compiler optimizations is shown in Figure 4.10 where TBB shows slower performance with glitches (with unknown reason for now), while OpenMP and OpenCL match well for nearly the whole range of matrix size. When compiler optimizations are enabled as shown in Figure 4.11, it neutralizes and smooths out the TBB performance graph with TBB narrowly slower than the others two while OpenMP and OpenCL almost match throughout. This performance behavior of the three frameworks in Figure 4.11 is somewhat similar to the compiler optimized version of convolution as shown previously by Figure 4.7.

4.2.6 Summary

These performance numbers collectively present interesting results. OpenMP shows better performance than TBB in most of the cases but OpenCL comes up with more interesting outcome. OpenCL, being a framework to program heterogeneous architectures, has actually demonstrated competitive performance compared to state-of-the-art CPU specific frameworks, namely OpenMP, and even better results than TBB in most cases. Similar results are drawn in the study [27] where OpenCL outperformed OpenMP in more

cases. It could be partly because OpenCL kernels are compiled for the given hardware, and therefore, it better exploits hardware specific optimizations such as SIMD parallelism. Another compelling result is the more enhanced performance of OpenMP compared to other frameworks when compiler optimizations are enabled since OpenMP is based on compiler directives. Compiler optimizations have, to some extent, effect on overall performance of all the frameworks which shows that modern compilers play a significant role in performance output. This has positive future implications when compilers may take some of the performance tuning responsibilities from programmers. We have also seen that vectorizing the OpenCL kernel boosts performance but the use of OpenCL *local memory* has no major advantage on CPUs as could be seen in Figure 4.2 and 4.3.

4.3 Scalability

In this section, we investigate how the three parallelism models scale with increasing number of cores in the system. It also shows how much threading overhead these frameworks incur when there is only one thread or a few threads available. The same system as used in performance evaluation, i.e., Intel Xeon CPU E5345 with 8 cores running at 2.33GHz, is used in scalability evaluation too. Default compiler optimizations i.e., `-O2`, are kept in these tests. In OpenCL, the fourth parameter to the `clBuildProgram` function is specified as `NULL` which enables default optimizations in the OpenCL compiler. Keeping the same domain matrix size in an application, the number of cores is varied from 1 to 8 to test speedup. The number of threads is controlled through available techniques in each of the three models. OpenMP has a clause named `num_threads(n)` and a function named `omp_set_num_threads(n)` to set an upper limit on the number of available threads for running the subsequent parallel region in an application (`n` threads in this case). The clause overrides the function if both are used simultaneously. We have used the OpenMP function to be flexible and could increase the number of threads in each run. Intel TBB gives this facility through the object creation of the `task_scheduler_init` class where the number of threads is specified in its constructor. OpenCL, on the other hand, provides an extension called *device fission* to set the maximum number of threads for a kernel. It is actually an interface which can be used for sub-dividing an OpenCL device into a number of sub-devices or groups. We made such a group with number of cores from 1 to 8 in each OpenCL kernel run to see how well speedup scales in comparison to OpenMP and Intel TBB.

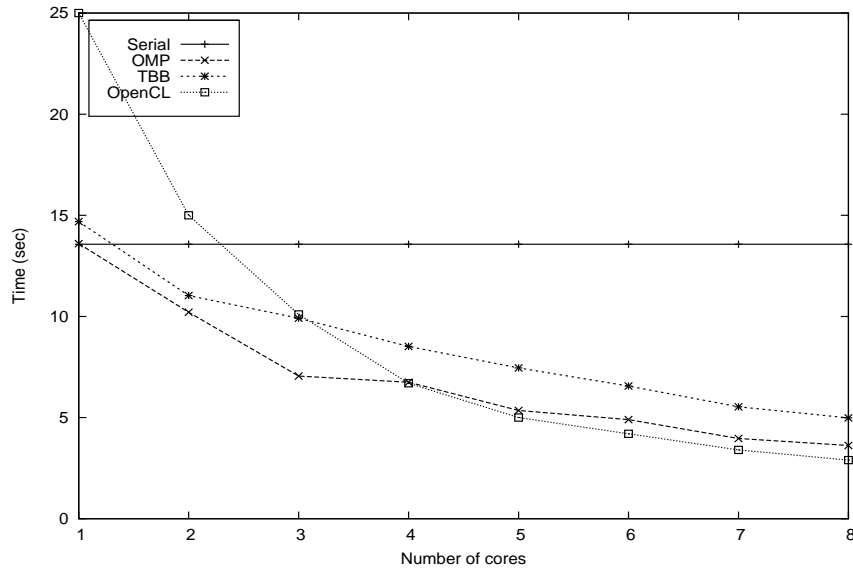


Figure 4.12: Matrix multiplication performance scaling with number of cores

4.3.1 Matrix Multiplication

Figure 4.12 shows the performance of the matrix multiplication algorithm on different numbers of cores as specified on the horizontal axis. As can be seen in the figure, OpenCL shows slowest performance for one core while OpenMP and TBB are close in performance with TBB behaving a bit slower than OpenMP. The straight horizontal line shows serial execution time which means that OpenMP behaved exactly like serial execution when run on a single core with no threading overhead. TBB and OpenCL, on the other hand, incur threading overhead with OpenCL involving significantly more overhead time than TBB. This figure implies that OpenCL is slower than serial even when two cores are used in the matrix multiplication application. While both OpenMP and TBB show lower execution time than serial when more than one cores are used. OpenCL starts showing better performance than serial execution when three or more cores are used. Beyond four cores, all the three frameworks catch up well in performance with each other with OpenCL exhibiting best performance on the highest number of cores and TBB a bit slower than the other two.

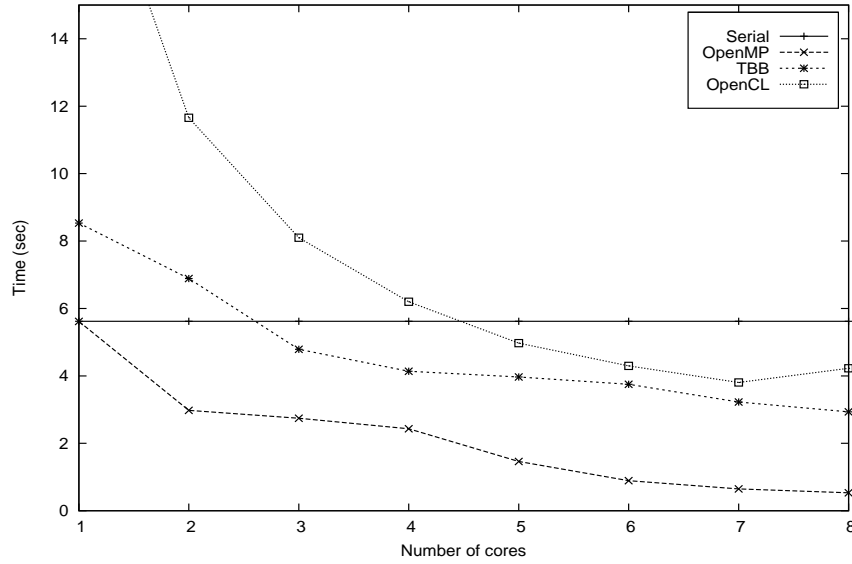


Figure 4.13: LU decomposition performance scaling with number of cores

4.3.2 LU Decomposition

OpenCL involves the highest overhead in LU factorization and shows performance improvement from five cores as shown in Figure 4.13. TBB incurs comparatively low overhead for a small number of cores but scales almost like OpenCL for high number of cores. OpenMP beats TBB and OpenCL significantly and scales best for all numbers of cores and incurs zero overhead in case of a single core.

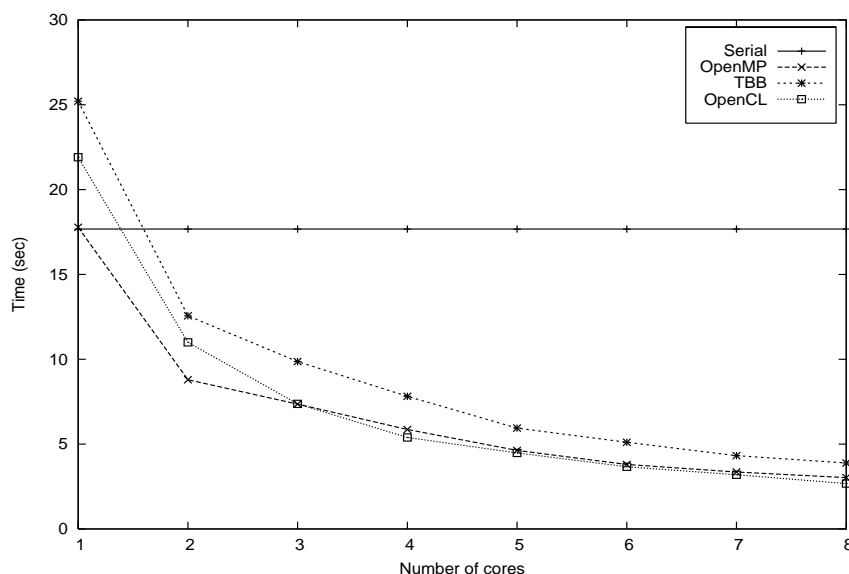


Figure 4.14: Image convolution performance scaling with number of cores

4.3.3 Image Convolution

Figure 4.14 illustrates that both TBB and OpenCL incur some overhead on single core but all of the three frameworks demonstrate identical performance when multiple cores are used. OpenMP with loops statically scheduled again shows no overhead for one core and scales better for all numbers of cores compared to dynamically scheduled loops in TBB.

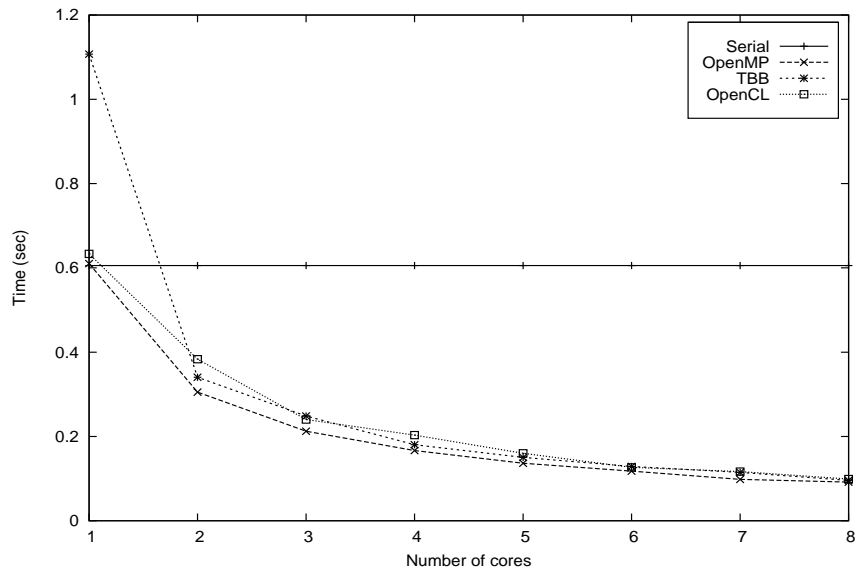


Figure 4.15: Histogram generation performance scaling with number of cores

4.3.4 Histogram Generation

All the three frameworks show similar performance graphs as can be seen in Figure 4.15. TBB here incurs the most overhead on a single core (using `parallel_reduce`) and OpenCL kernel shows less overhead than other applications (using AMD samples kernel).

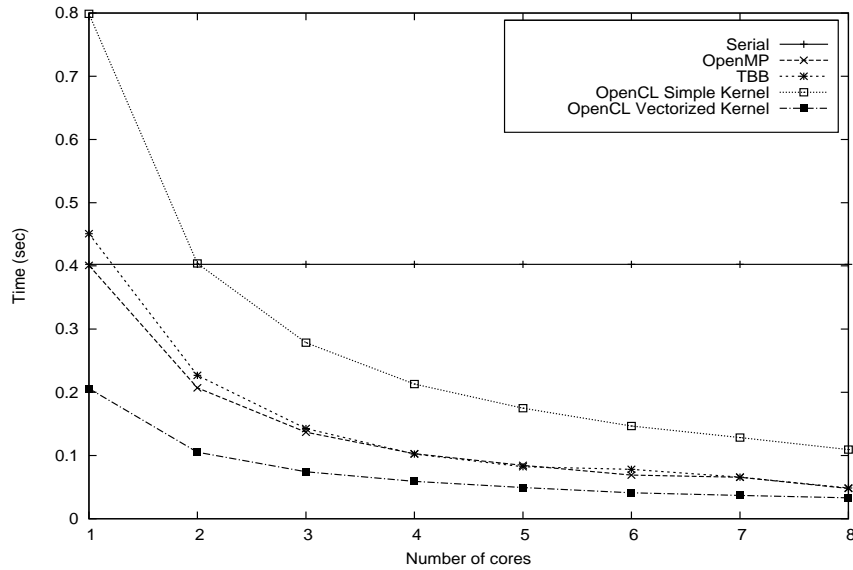


Figure 4.16: Pi value calculation performance scaling with number of cores

4.3.5 Pi Calculation

The Pi value calculation performance graph is very interesting. As shown by Figure 4.16, OpenMP and TBB solutions have almost identical performance for all number of cores. OpenCL’s simple kernel solution is considerably slower than the others two. But, the vectorized OpenCL kernel shows the best performance in comparison with all the other three solutions. Most interestingly, the OpenCL vectorized kernel utilizing SIMD (single instruction multiple data) instructions, yields 2x performance on a single core compared to the serial run.

4.3.6 Summary

It is interesting to see in all these scalability tests that OpenMP showed no threading overhead when run on a single core. The zero overhead in OpenMP execution on a single core suggests that some OpenMP implementations have a special code path for a single thread that effectively adds a single branch. The OpenCL and TBB platforms, on the other hand, show some overhead on a single core but they catch up with OpenMP when multiple cores are used. OpenCL shows a competitive performance scaling on CPU compared to the CPU specialized frameworks but with a little higher overhead on a single or few number of cores.

These results conform with [27] where OpenCL gives competitive perfor-

mance numbers compared to OpenMP. Also this study is in agreement with [14] but is in contrast with [20] where OpenMP incurs most overhead and TBB scales best for all numbers of cores. The rationale behind this contrast could be compiler differences and suitability of different frameworks to different applications. In our case, OpenMP static scheduling compared to TBB’s dynamic scheduling suited well in most cases since these applications have a fixed number of loop iterations with equal distribution of work across different loop iterations.

Chapter 5

OpenCL Platforms Evaluation

In this chapter, we test our OpenCL applications on three implementations of the OpenCL standard available from AMD, Intel and Nvidia. At first, all applications are run with AMD and Intel OpenCL platforms using an Intel CPU to see performance implications of these platforms. Later, we run our implementations on the Nvidia GPU using Nvidia OpenCL platform and compare it with GPU optimized applications to see both code and performance portability.

5.1 AMD and Intel OpenCL

In this section, we evaluate two OpenCL implementations available for multicore CPUs from two different vendors. AMD OpenCL SDK 2.5 and Intel OpenCL SDK LINUX 1.1, both implementing the OpenCL 1.1 specification, were experimented with on a system with Intel Xeon CPU E5520 hosting 16 cores, each running at 2.27 GHz. And as compiler, we used gcc version-4.6.1.

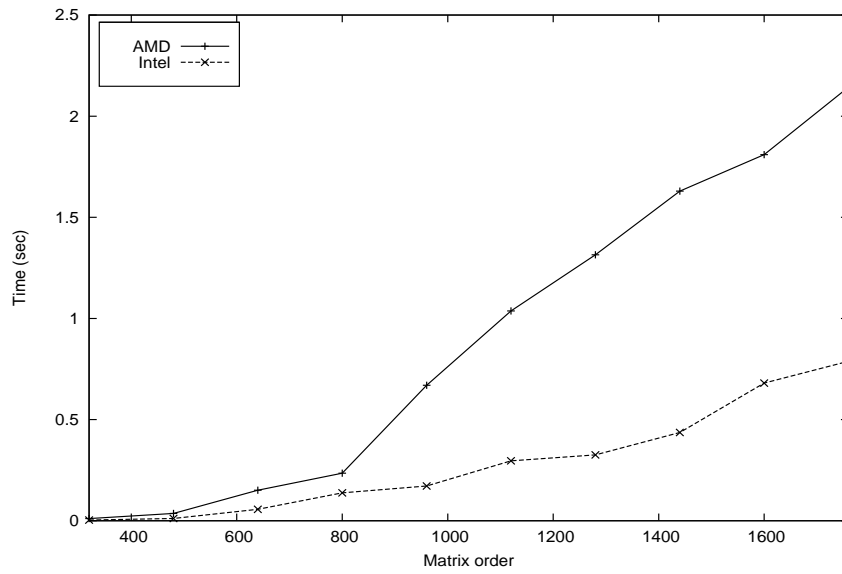


Figure 5.1: Matrix multiplication performance of OpenCL from AMD, Intel on an Intel CPU

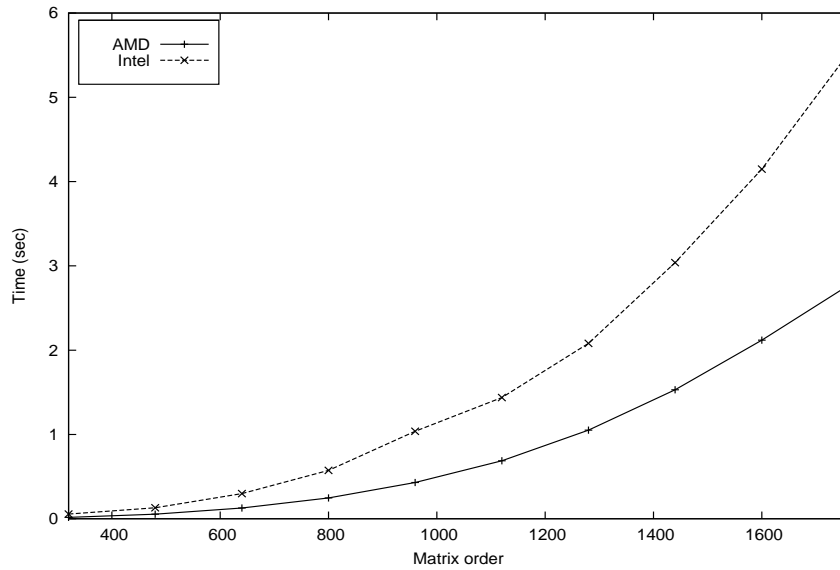


Figure 5.2: Matrix multiplication performance of the OpenCL kernel using local-memory/work-groups from AMD, Intel platforms

5.1.1 Matrix Multiplication

Figure 5.1 shows that the Intel OpenCL platform performs better than the AMD OpenCL platform when the same application of matrix multiplication is run on both of them. We earlier saw that when the matrix multiplication kernel was optimized with respect to local and private memories, it hardly benefited from those optimizations or even degraded the resultant performance as was shown in Figure 4.1. When the same memory optimized kernel was run on these OpenCL platforms, the AMD performance remained unaffected while that of Intel degraded significantly as shown by Figure 5.2. This suggests that the Intel Platform was tuning the kernel by optimizations but when we directed memory optimizations, those auto-optimizations were lost.

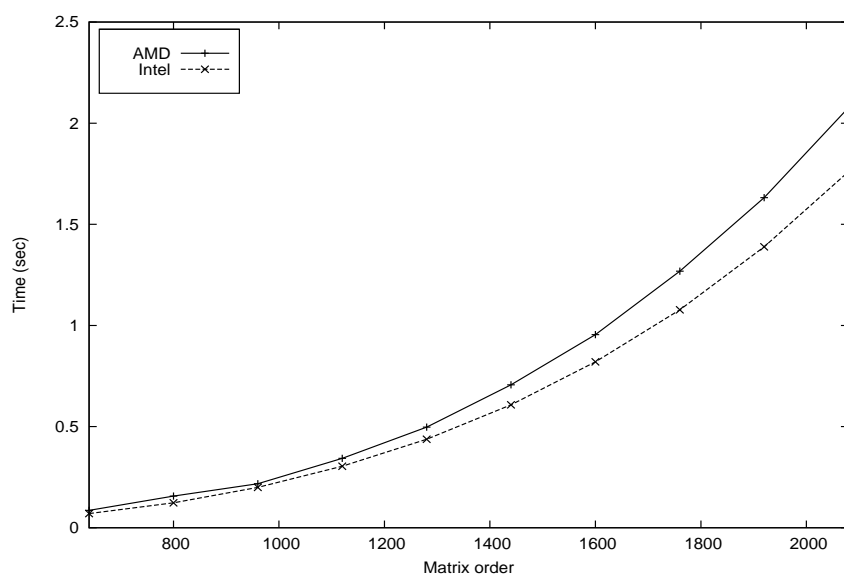


Figure 5.3: LU decomposition performance of OpenCL with AMD, Intel platforms

5.1.2 LU Decomposition

The LU factorization application performed similarly on both OpenCL platforms. Figure 5.3 indicates that the Intel platform performed slightly faster than the AMD OpenCL on the highest order matrices.

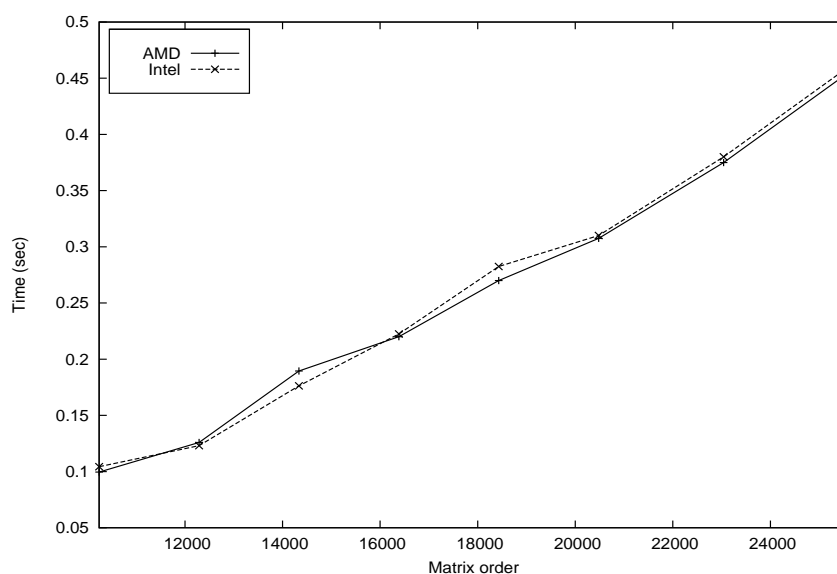


Figure 5.4: Image histogram generation performance of OpenCL from AMD, Intel platforms

5.1.3 Histogram Generation

The image histogram generation application shows the closest performance behavior of the AMD and Intel platforms through a wide range of matrix orders, shown in Figure 5.4.

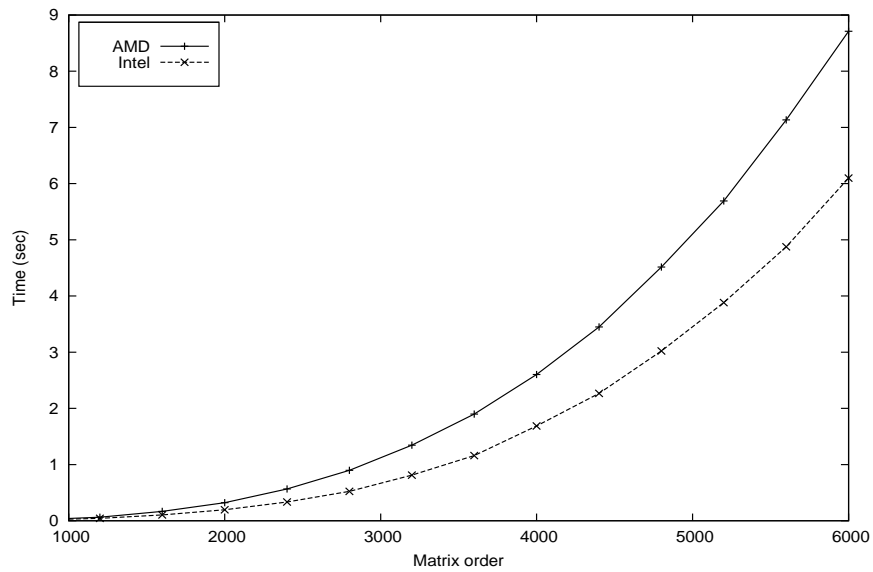


Figure 5.5: Image convolution performance of OpenCL from AMD, Intel platforms

5.1.4 Image Convolution

The performance graph for image convolution in Figure 5.5 is similar to that of LU decomposition except that the Intel SDK shows better performance from start of matrix orders.

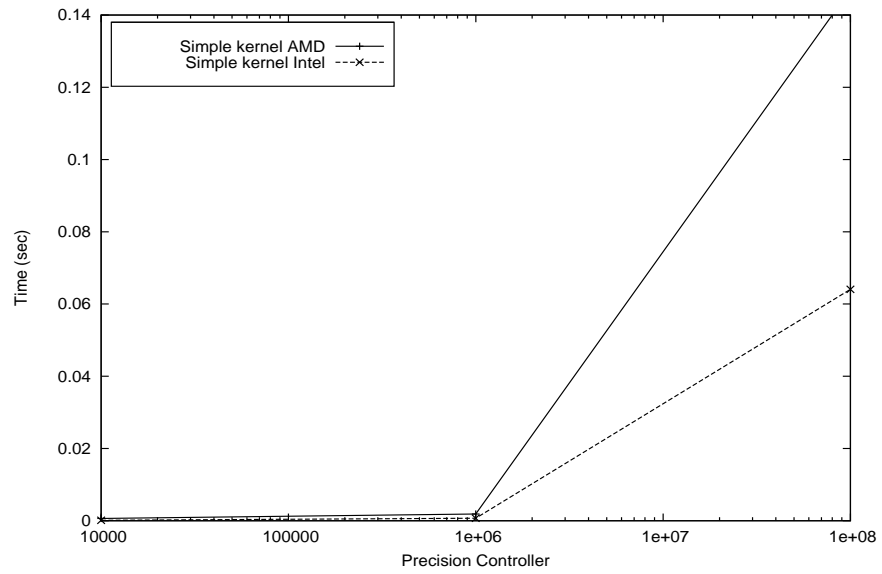


Figure 5.6: Pi approximation performance of OpenCL simple kernel from AMD, Intel platforms

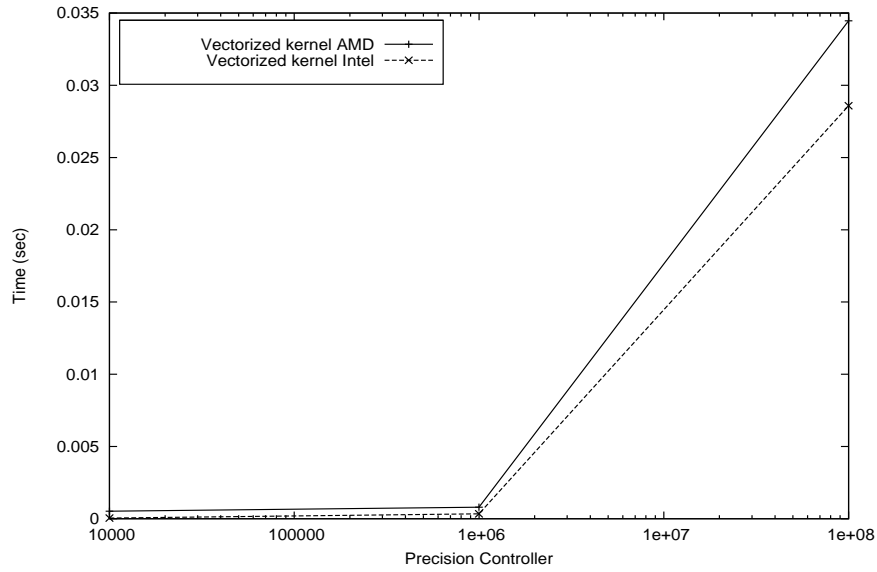


Figure 5.7: Pi approximation performance of OpenCL vectorized kernel from AMD, Intel platforms

5.1.5 Pi Calculation

The performance behavior of Intel OpenCL is no different in case of Pi value approximation than with the other applications as Figures 5.6 and 5.7 indicate. Both simple and vectorized kernels perform faster on the Intel platform. But the performance gap between AMD and Intel OpenCL is wider when the simple kernel is run. This gap narrows when the vectorized kernel is run on both platforms. This indicates that manually vectorizing the kernel helped AMD compiler more than the Intel OpenCL compiler.

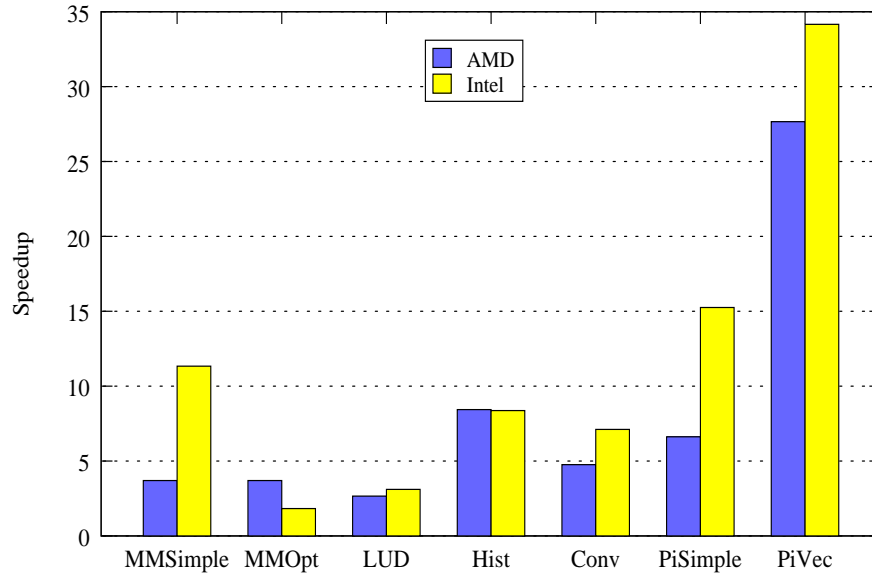


Figure 5.8: OpenCL vendor dependent platforms evaluation - AMD vs Intel

5.1.6 Summary

Interestingly Intel’s OpenCL outperformed AMD in four out of the five tested benchmark applications with a clear margin in performance tests on CPUs as shown by a combined speedup graph in Figure 5.8. Histogram generation is the only application in which the AMD has very similar speedup to the Intel OpenCL, and the optimized kernel of matrix multiplication degrades in performance on Intel SDK while it retains it on AMD SDK so AMD performs better there. All these tests are done on Intel Xeon CPU E5520 hosting 16 cores, each running at 2.27 GHz, as stated earlier.

This better behavior of Intel OpenCL platform suggests that Intel enables its CPU specific auto-optimizations, namely, auto-vectorization and better use of memories, using its JIT OpenCL compiler. This is in agreement with the results in [27] which claims that the Intel OpenCL compiler uses platform-specific optimizations (vectorization using SSE4.1). Therefore, it performs better than the AMD OpenCL compiler in most of the cases. When the matrix multiplication kernel was optimized using local OpenCL memories, it hugely degraded performance on the Intel platform while that of AMD was relatively unaffected. This suggests that the Intel SDK was doing such optimization itself according to its hardware architecture which was lost during our optimizations. While no effect on the AMD platform shows that AMD SDK was not doing any such optimizations which fits Intel’s hardware architecture. On the other hand, when the

Pi calculation kernel was explicitly vectorized, AMD running time dropped significantly compared to that of Intel though Intel is still outperforming AMD. As in Figure 5.8, the vectorized kernel increased Intel’s speedup by 1.25x and that of AMD by around 3.2x. This also implies that the AMD platform was not doing such optimizations on Intel CPU. This significant relative difference for AMD shows that the Intel OpenCL compiler was already exploiting auto-vectorization to some extent on Intel CPUs when it was not explicitly programmed in the kernel. These results show the magnitude of the OpenCL platforms conformance with their own hardware architectures. But the results from this study are somewhat biased since the Intel CPU is used which suits better to the Intel compiler. And therefore, these results should not be taken as the absolute performance gap between these two OpenCL compilers.

These results also imply that use of workgroups and OpenCL local/private memories may not always optimize performance on CPUs as in the matrix multiplication case since CPUs do not have such a strict memory architecture as GPUs. While use of vectorization, on the other hand, improves performance on CPUs.

5.2 Code/Performance Portability

Experiments in this section are done on a Nvidia Tesla M2050 GPU containing Nvidia OpenCL driver 280.13 which implements the OpenCL 1.1 specification.

To test code portability, we run our OpenCL implementations on the GPU. In all cases, the code was executed on the GPU without requiring any changes in the code. This strengthens the OpenCL claim for code portability across different hardware architectures.

Although the code is portable, the optimizations for OpenCL implementation on CPUs and GPUs are quite different. Optimizations such as usage of local and private memory may be helpful for GPU execution but they can affect negatively when executing on multicore CPUs as could be seen earlier in the matrix multiplication case. In the following, we compare performance of an OpenCL algorithms that are primarily written for CPU execution with optimized GPU implementations of the same set of applications on the GPU. The GPU-optimized implementations are taken from NVIDIA OpenCL SDK code samples.

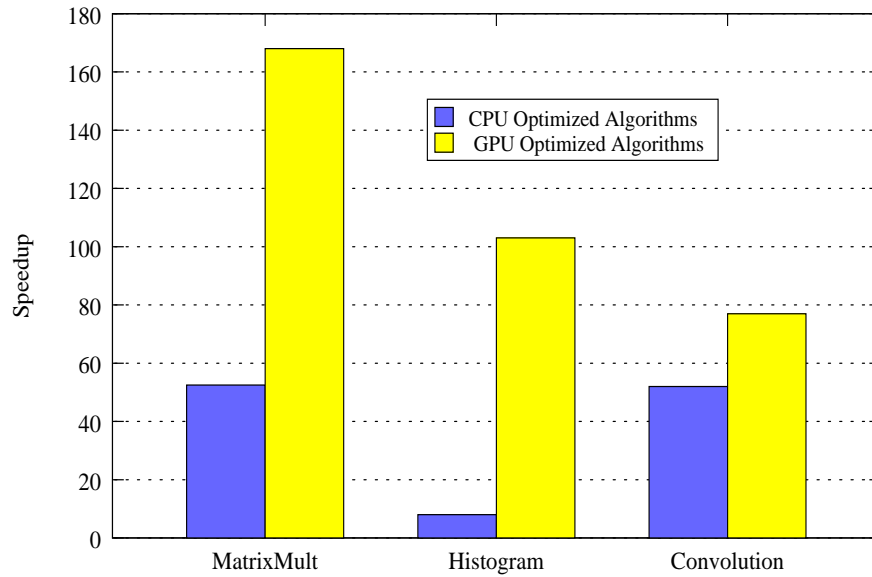


Figure 5.9: Performance comparison of CPU algorithms with GPU optimized algorithms run on a GPU

Figure 5.9 shows performance comparisons of two sets of OpenCL solutions. One set of OpenCL applications are written for CPU and the other is the same set of applications which are written and optimized to run on GPU, taken from the Nvidia SDK examples. Only the first three applications, namely, matrix multiply, histogram generation and convolution are taken in this comparison since no corresponding Nvidia optimized code for the same algorithms was found for LU decomposition and Pi value calculation. The speedups are calculated with respect to sequential CPU execution on a single core. It is clear that the GPU optimized algorithms outperform all our three applications which were primarily tuned for CPUs.

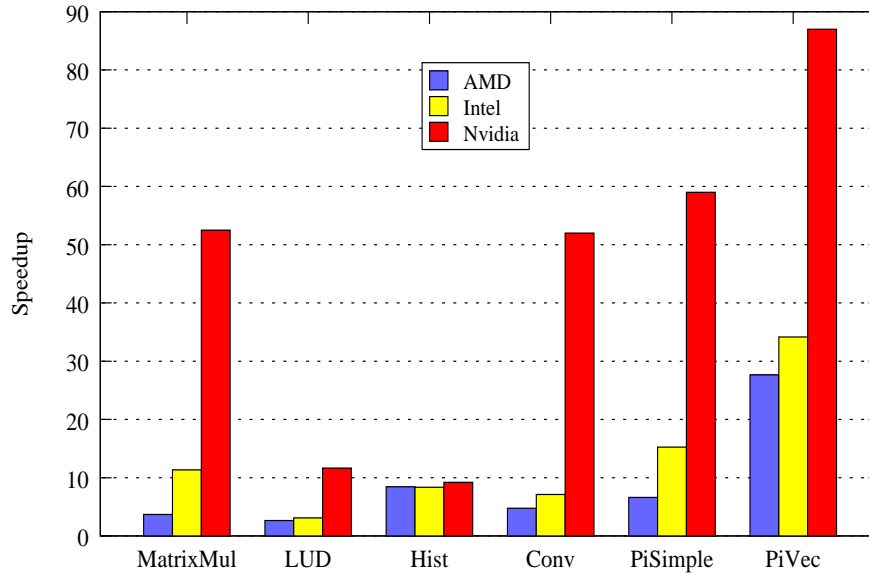


Figure 5.10: Performance comparison of CPU algorithms on AMD and Intel OpenCL platforms using Intel CPU and Nvidia OpenCL platform using Nvidia GPU

The results can be interpreted in two ways:

- There is a significant performance impact of architecture-specific optimizations. As illustrated by Figure 5.9, GPU optimized results outweigh our implementations by 3.3x in matrix multiply and 1.5x in convolution application. In case of histogram where the kernel is taken from AMD samples that is optimized for (AMD) CPUs, the GPU speedup is 12.3x.
- On the positive side, not only the code primarily written for CPU is portable, it also gives some performance improvements when executed on a GPU. For example, in case of matrix multiplication, the speedup is increased from 4 and 11 times (AMD and Intel platforms respectively) to 50 times when running the same implementation on a powerful GPU (see Figure 5.10). Similarly, there is the speedup elevation for LU decomposition from around 3 on CPU to 11 on GPU, from around 8 on CPU to 9 on GPU in histogram generation, from 5 and 7 on CPU (AMD, Intel) to around 50 on GPU in case of convolution, from 7 and 15 (AMD, Intel) to 60 on GPU in the Pi application. In the Pi vectorized kernel, the speedup is increased from around 28 and 34 on CPU (AMD, Intel respectively) to close to 90 on Nvidia GPU.

The GPU test shows multiple times magnified performance of our algorithms

than its CPU equivalent in Figure 5.10. The reason is clearly the difference of hardware architecture since GPUs offer better compute power and the notion of platform implicit optimizations for its hardware (since Nvidia OpenCL SDK used on Nvidia GPU). But it still suggests a reasonable performance of CPU optimized algorithms on GPUs.

Chapter 6

Usability Evaluation

Besides the performance and scalability issues, programmability plays also an important role in adopting a certain parallelism framework. Programmers can have higher productivity resulting in low cost if certain frameworks are chosen which are easier to learn and program with. To experience the programming complexity of the frameworks, all the five benchmark applications in our study were parallelized from reference C/C++ implementations using these three frameworks, i.e., OpenMP, Intel Threading Building Blocks and OpenCL. This made it possible to compare complexities involved and programmability of these frameworks. During parallelizing these applications, an empirical approach was taken to document the amount of time spent and how much of online support such as tutorials, code samples and community support was available. The number of lines of code is also recorded in each case.

6.1 Usability

Introducing parallelism using OpenMP pragmas was the simplest practice preserving the serial semantics compared to the other two frameworks. The only delicate issue during the application of OpenMP constructs besides making loops independent was choosing the right variables to be made shared, private or firstprivate etc. But OpenMP pragmas spread throughout the serial program makes readability slightly hard since it commits additional hierarchies to the nested loops thereby making the scope of every loop and construct somewhat more complex.

TBB needs parallelized portions of the program to be encapsulated into template classes with a medium effort and thus moved parallelized code out of the serial structure of the main program contributing to object oriented style and better program readability. The equivalent subtle issue with TBB as also pointed out in [14], compared to OpenMP variable sharing, was to select the right variables to be included in the

`parallel_for/parallel_reduce` object’s class definitions. Similarly, the effort for identifying the most suitable scheduling strategy out of static, dynamic and guided for OpenMP resembled the effort for experimenting with default auto-partitioner, simple-partitioner or affinity partitioner and grain-sizes in TBB. TBB, however, made the main program more manageable by moving looping structures to the template classes while the use of lambda expressions in the convolution application made the code relatively compact but rendered the looping structure almost like OpenMP.

OpenCL, on the other hand, following GPU programming fashion constitutes a quite different style of programming. It involved a fixed amount of code and effort to initialize the environment which was baseline and was reused in all applications so the successive implementations were easier in OpenCL. Since specifying workgroup sizes and use of local/private memories in matrix multiplication and LU factorization did not benefit well or even degraded performance on CPUs. This extra effort can be avoided while programming in OpenCL for CPUs if a high level of performance optimizations is not desired. Introducing explicit vectorization in the kernel, however, was worth the effort and it showed high performance both on CPU and GPU. Most of the optimization literature found is GPU specific and does not significantly improve performance on CPUs which makes OpenCL programming for CPUs comparatively easier than for GPUs.

A summary in Table 1 shows the time/effort as empirical representation of the combined time in weeks for learning and mapping each application into the corresponding framework by the user and LOC as the number of lines of code for each parallelized application using these models in our implementations. Notation +++ denotes the lowest while - - - denotes the highest time/effort. The first value in LOC of OpenCL represents the constant number of lines of code that acts as a boilerplate while the second value is variable for every application.

	OpenMP		Intel TBB		OpenCL	
	Time	LOC	Time	LOC	Time	LOC
<i>MatrixMultiply</i>	+	+	+	+	+	+
<i>LUFactorization</i>	+	+	-	-	-	-
<i>Convolution</i>	+	+	+	+	-	-
<i>Histogram</i>	+	+	-	-	-	-
<i>PiCalculation</i>	+	+	+	+	-	-

Table 6.1: Time/effort and number of lines of code for given benchmarks for the three frameworks

Chapter 7

Related Work

This chapter presents related previous studies and compares them with our work. Part of this work is also published in the “4th Swedish Workshop on Multicore Computing (MCC-2011)” at Linköping [2] and “5th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG’12)” [3] at Paris. Individually, the three considered frameworks have been studied extensively. As in OpenMP case, a performance comparison of OpenMP constructs is given in [23] depending upon different operating platforms and compilers. A study [6] proposes addition of a new OpenMP construct `onthreads` to the OpenMP work-sharing directives. It is used to specify a sub-team of threads from the current team to share the workload. Intel Threading Building Blocks is studied in [18, 24], and [30] even advocates the addition of TBB contents to the universities multi-core related curriculum asserting that it is a higher level templated threading library for C++. OpenCL is also used to implement the symmetric key cryptographic algorithm Advanced Encryption Standard (AES) in [12] that shows good performance, scalability and portability. Another study [17] applies OpenCL to program Intel Single-chip Cloud Computer (SCC) which is a homogeneous non-cache-coherent system and has 48 cores on a single chip. It is found that the OpenCL’s weak memory consistency model fits well with such an architecture.

A comparative study of different frameworks including OpenMP, TBB, Cilk++ and RapidMind frameworks is done in [19]. But this study only tests these frameworks on a histogram generation application and looks for any built-in support to handle race conditions in these frameworks. It concludes that OpenMP and TBB do not provide any support for automatically detecting and resolving race conditions. On the other hand, Cilk++ provides tools to detect race conditions and RapidMind is free of deadlocks and race conditions by design. In our study, therefore, we had to calculate a subhistogram in each thread and then merge these subhistograms in an OpenMP `critical` section and TBB’s `join` class method.

An extension of [19] study is [20] by the same authors. Here they also add OpenCL to the multi-core frameworks evaluation and apply them to a 2D/3D image registration application. This study also evaluates usability in addition to the performance and overhead. The overhead results of this study are somewhat in contrast with our study. It shows that OpenMP instead has the most overhead on a single and few cores while it scales well for an increased number of cores. We find no overhead for OpenMP on a single core. The reason behind this contrast could be compiler differences. The zero overhead in OpenMP execution on a single core in our case suggests that some OpenMP implementations have a special code path for a single thread that effectively adds a single branch. It can also be due to the suitability of different frameworks to different applications. The usability study of this work confirms our results and maintains that OpenCL requires the most familiarization time and mapping effort, TBB needs average time and effort while OpenMP is the most user-friendly out of these three.

A comparison of OpenMP and TBB on a medical imaging application is done in [14]. This work shows less OpenMP overhead and comparatively superior performance which matches with most of our results. It also affirms more usability of OpenMP compared to TBB.

A very interesting study is [27] which compares the programmability of OpenMP and OpenCL. It judges programmability in terms of *productivity*, *portability* and *performance*. The results of this work are very close to our results. It maintains that OpenMP is a winner with less lines of code and less multi-core programming effort when starting from a sequential code. OpenCL is more portable than OpenMP as we could also run the same code on both a multi-core CPU and GPU. This work also shows that OpenCL actually outperforms OpenMP in more cases on multi-core CPUs without much CPU specific optimizations. Furthermore, it affirms that the OpenCL performance is influenced by the platform: the Intel Platform usually outperforms the AMD one and uses platform specific optimizations like vectorization.

Chapter 8

Discussion and Conclusion

8.1 Discussion

We evaluate the applicability of the OpenCL programming model for multi-core CPUs on multiple benchmarks in comparison with two state-of-the-art CPU specialized frameworks, OpenMP and Intel Threading Building Blocks. The overall performance and scalability numbers show that OpenCL on CPU yields competitive performance when compared with the CPU specific frameworks. Enabling compiler optimizations had significant impacts on results particularly with OpenMP since it takes a compiler directives based approach to parallelism. The OpenCL compiler also had alternative auto-optimization options for tuning performance in comparison with the other frameworks which relied on `-Ox` compiler optimizations support. Enabling explicit vectorization in OpenCL kernels improved performance while the use of local/private memory did not yield major performance improvements.

When the same OpenCL code was tested on the Intel CPU using different vendor-dependent platforms, the Intel platform outperformed that by AMD in four out of the five tested applications with a good margin. This shows that Intel’s OpenCL implementation has better implicit optimizations for Intel CPUs. OpenCL code written for CPUs could successfully run on the Nvidia platform with Tesla GPUs. This confirms that OpenCL guarantees code portability and correctness of execution across different platforms and hardware architectures. The performance of CPU specific code on GPU reasonably surpassed its CPU performance but it did not utilize the full capacity of highly data parallel GPUs as GPU optimized algorithms did since there is a richer set of optimization options for GPUs in OpenCL. This implies reduced portability of OpenCL performance between CPUs and GPUs. The implementation experience of the three frameworks shows slightly more effort on behalf of Intel TBB compared to OpenMP while OpenCL needed the most time and effort since complete code had to be restructured. Since there are limited performance tuning options in OpenCL for CPUs, it makes

CPU programming in OpenCL comparatively simpler than for GPUs. This has the advantage of easy integration of GPUs as accelerators to CPUs in a multicore computing environment without code changes given that performance expectations on behalf of GPUs are moderate.

8.2 Conclusion

We have seen that OpenCL gave competitive performance at the cost of major restructuring of the code when compared to CPU specialized state-of-the-art frameworks, OpenMP and Intel TBB. OpenMP gives good CPU performance with low programming complexity and code changes than Intel Threading Building Blocks. Intel TBB, on the other hand, made code more readable following an object-oriented programming style which plays an important role when it comes to huge and complex applications. Thus OpenMP can be used to parallelize existing applications while Intel TBB is a good candidate for developing new multicore applications.

This competitive multicore CPU performance of OpenCL at the cost of adopting a different programming style is worth the effort since OpenCL offers code portability across a number of different architectures. This has significant implications in future high performance computing with the addition of data parallel multicore GPUs entering into mainstream computing. The considered Intel and Nvidia platforms outperformed AMD when tested on the Intel CPUs and Nvidia GPUs respectively implying OpenCL platforms doing implicit optimizations for their own architectures. We also have seen that enabling compiler optimizations had significant impact on overall performance numbers. Therefore, we hope that future versions of OpenCL compilers will take care of the architecture specific optimizations detecting the underlying architecture differences at runtime and will reduce this performance gap and increase performance portability.

8.3 Future Work

More applications could be added to the set of benchmarks to extend the applicability of results. Applications evaluated here test data parallelism, it is also worth the effort to see how much task parallelism can be exploited using these frameworks. Having a wider base of applications, it could be interesting to recognize certain patterns which are suited for specific frameworks, both from performance and usability perspective.

We have seen that OpenCL implementations from different vendors provided different performance numbers. Intel SDK outperformed AMD running on Intel CPU in most cases. This implies architecture specific auto-optimizations support in these implementations. These results could be consolidated when these applications are run on an AMD CPU in multicore

CPU tests. Similarly, in case of GPUs, tests could also be run on an AMD GPU to compare the performance of Nvidia and AMD OpenCL platforms.

We saw that CPU specific OpenCL algorithms showed performance portability to some extent when run on modern GPU. The reverse tests, i.e., GPU specific OpenCL algorithms, can also be tested on CPUs to complement performance portability issues.

Appendices

Appendix A

Matrix Multiplication

A.1 OpenMP

```

1 void mat_omp(float* mat1, float* mat2, float* mat3)
2 {
3     omp_set_num_threads(nr_threads);
4     int i, j, k;
5     // warm-up run (Code removed since it is given below anyway)
6
7     // loop runs
8     double omp_time = 0.0;
9     tick_count omp1, omp2;
10    for(int iter=0; iter<AVG_ITERATIONS; iter++)
11    {
12        memset(mat3, 0, size*size*sizeof(float));
13        omp1 = tick_count::now();
14        #pragma omp parallel shared (mat1, mat2, mat3) private (i, j, k)
15        {
16            #pragma omp for schedule (static)
17            for(i=0; i<size; i++)
18            {
19                for(j=0; j<size; j++)
20                {
21                    for(k=0; k<size; k++)
22                    mat3[i*size + j] += mat1[i*size + k] * mat2[k*size + j];
23                }
24            }
25        }
26        omp2 = tick_count::now();
27        omp_time += (omp2 - omp1).seconds();
28    }
29
30    omp_time /= (double)AVG_ITERATIONS;
31
32    cout<<"Elapsed_avg_time_for_OpenMP:_"<<omp_time<<endl;
33    ompCollector.addData(size, omp_time);
34 }

```

A.2 TBB

```

1
2 class mat_mul
3 {
4     float* a;
5     float* b;
6     float* c;
7 public:
8     mat_mul(float* mat1, float* mat2, float* mat3) :
9         a(mat1), b(mat2), c(mat3) {}
10
11     void operator()(blocked_range<int>& r) const
12     {
13         int last = r.end();
14         int i, j, k;
15         for (k = r.begin(); k != last; ++k)
16         {
17             for (i = 0; i < size; ++i)
18             {
19                 for (j = 0; j < size; ++j)
20                 {
21                     c[i*size + j] += a[i*size + k] * b[k*size + j];
22                 }
23             }
24         }
25     }
26 };
27
28 // Invoking the function object
29 void mat_tbb(float* mat1, float* mat2, float* mat3)
30 {
31     // selecting number of threads
32     task_scheduler_init TBBinit(nr_threads);
33     // warm-up run
34     parallel_for(blocked_range<int>(0,size), mat_mul(mat1, mat2, mat3));
35     // loop runs
36     double tbb_time = 0.0;
37     tick_count tbb1, tbb2;
38     for(int i=0; i<AVG_ITERATIONS; i++)
39     {
40         memset(mat3, 0, size*size*sizeof(float));
41         tbb1 = tick_count::now();
42         parallel_for(blocked_range<int>(0,size), mat_mul(mat1, mat2, mat3));
43         tbb2 = tick_count::now();
44
45         tbb_time += (tbb2 - tbb1).seconds();
46     }
47
48     tbb_time /= (double) AVG_ITERATIONS;
49
50     cout<<"Elapsed_avg_time_for_TBB---: " <<tbb_time<<endl;
51     tbbCollector.addData(size, tbb_time);
52 }

```

A.3 OpenCL

```

1  /* MatrixMul kernels */
2
3
4  /*-----Simple kernel-----*/
5  __kernel void MatrixMul_simple(__global float* A, __global float* B,
6                                __global float* C, int widthA, int widthB)
7  {
8      int IDx = get_global_id(0);
9      int IDy = get_global_id(1);
10     float sum = 0.0f;
11     for (int i = 0; i < widthA; ++i)
12     {
13         float tempA = A[IDy*widthA + i];
14         float tempB = B[i*widthB + IDx];
15         sum += tempA * tempB;
16     }
17     C[IDy*widthA + IDx] = sum;
18 }
19
20 /*-----Optimized kernel-----*/
21 __kernel void MatrixMul_opt(__global float* A, __global float* B, __global
22                             float* C,
23                             int heightA, int widthB, __local float* Bwork)
24 {
25     int j, k;
26     int IDx = get_global_id(0);
27     int nglob = get_global_size(0);
28     int iloc = get_local_id(0);
29     int nloc = get_local_size(0);
30     float sum;
31     int heightC = heightA;
32     int widthC = widthB;
33     int common_dim = widthB;
34     for(j = 0; j < widthC; j++)
35     {
36         for(k = iloc; k < common_dim; k+=nloc)
37         Bwork[k] = B[k*common_dim + j];
38         barrier(CLK_LOCAL_MEM_FENCE);
39         sum = 0.0f;
40         for (k = 0; k < common_dim; k++)
41         {
42             float tempA = A[IDx*heightC + k];
43             float tempB = Bwork[k];
44             sum += tempA * tempB;
45         }
46         C[IDx*heightC + j] = sum;
47     }

```

Appendix B

LU Decomposition

B.1 OpenMP

```

1 void lud_omp_comb(float* matrix_omp2)
2 {
3     int i, j, k;
4     float* o2 = new float[size*size];
5     for(int ii=0; ii<size*size; ii++)
6         o2[ii] = matrix_omp2[ii];
7
8     // warm-up run (code removed)
9
10    // Looping runs
11    i = j = k = 0;
12    tick_count omp1, omp2;
13    double omp_comb = 0.0;
14    for(int iter=0; iter<AVG_ITERATIONS; iter++)
15    {
16        // Reloading inputs
17        for(int ii=0; ii<size*size; ii++)
18            o2[ii] = matrix_omp2[ii];
19
20        omp1 = tick_count::now();
21        for(k=0; k<size-1; k++)
22        {
23            #pragma omp parallel for shared(o2) private(i,j) schedule(static)
24            for(i=k+1; i<size; i++)
25            {
26                o2[i*size + k] = o2[i*size + k] / o2[k*size + k];
27                for(j=k+1; j<size; j++)
28                    o2[i*size + j] = o2[i*size + j] - o2[i*size + k]*o2[k*size + j];
29            }
30        }
31        omp2 = tick_count::now();
32
33        omp_comb += (omp2 - omp1).seconds();
34    }
35

```

```

36 // Copying results to original matrix
37 for(int ii=0; ii<size*size; ii++)
38     matrix_omp2[ii] = o2[ii];
39
40 delete[] o2;
41
42 omp_comb /= (double)AVG_ITERATIONS;
43 cout<<"Elapsed_avg_time_for_OpenMP_combined_loops="<<omp_comb<<endl;
44 omp2Collector.addData(size, omp_comb);
45 }

```

B.2 TBB

```

1
2 class lud_class
3 {
4     float* my_a;
5     const int n;
6 public:
7     lud_class(float* arr, int num) : my_a(arr), n(num) {}
8     void operator()(const blocked_range<int>& r) const
9     {
10         float* a = my_a;
11         int begin = r.begin();
12         int end = r.end();
13         float factor;
14
15         for(int k= begin; k!=end-1; k++)
16         {
17             for(int i=k+1; i!=n; ++i)
18             {
19                 factor = a[i*n + k]/a[k*n + k];
20                 for(int j=k+1; j!=n; ++j )
21                     a[i*n + j] = a[i*n + j] - factor * a[k*n + j];
22             }
23         }
24     }
25 };
26
27 // Invoking function object
28 void lud_tbb(float* matrix_tbb)
29 {
30     task_scheduler_init TBBinit(nrthreads);
31     float* tbb = new float[size*size];
32     for(int i=0; i<size*size; i++)
33         tbb[i] = matrix_tbb[i];
34
35     // warm-up run
36     parallel_for(blocked_range<int>(0, size), lud_class(tbb, size));
37
38     for(int i=0; i<size*size; i++)
39         tbb[i] = matrix_tbb[i];
40
41     // Looping runs

```

```

42 double tbb_time = 0.0;
43 tick_count tbb1, tbb2;
44 for(int iter=0; iter<AVG_ITERATIONS; iter++)
45 {
46     // Reloading inputs
47     for(int i=0; i<size*size; i++)
48         tbb[i] = matrix_tbb[i];
49
50     tbb1 = tick_count::now();
51     parallel_for(blocked_range<int>(0, size), lud_class(tbb, size));
52     tbb2 = tick_count::now();
53
54     tbb_time += (tbb2-tbb1).seconds();
55 }
56 // Copying results to original matrix
57 for(int i=0; i<size*size; i++)
58     matrix_tbb[i] = tbb[i];
59
60 delete[] tbb;
61
62 tbb_time /= (double)AVG_ITERATIONS;
63 cout<<"Elapsed_avg_time_for_TBB_outer_loops="<<tbb_time<<endl;
64 tbbCollector.addData(size, tbb_time);
65 }

```

B.3 OpenCL

```

1
2 __kernel void lud_opt(__global float4* matrix, uint u, int k)
3 {
4     int gid = get_global_id(0);
5     int lid = get_local_id(0);
6     int n = get_local_size(0);
7     int group_id = get_group_id(0);
8     int xx = lid * VECTOR_SIZE;
9     int xx1 = xx + 1;
10    int xx2 = xx + 2;
11    int xx3 = xx + 3;
12
13    __local float ratio;
14
15    if(group_id > k)
16    {
17        if(k == xx)
18            ratio = matrix[group_id*n + lid].x / matrix[xx*n + lid].x;
19        if(k == xx1)
20            ratio = matrix[group_id*n + lid].y / matrix[xx1*n + lid].y;
21        if(k == xx2)
22            ratio = matrix[group_id*n + lid].z / matrix[xx2*n + lid].z;
23        if(k == xx3)
24            ratio = matrix[group_id*n + lid].w / matrix[xx3*n + lid].w;
25    }
26
27    if(group_id > k)

```



```

28     {
29         if(xx >= k)
30         {
31             matrix[group_id*n + lid] -= matrix[k*n + lid] * ratio;
32         }
33         else if(xx1 >= k)
34         {
35             matrix[group_id*n + lid].y -= matrix[k*n + lid].y * ratio;
36             matrix[group_id*n + lid].z -= matrix[k*n + lid].z * ratio;
37             matrix[group_id*n + lid].w -= matrix[k*n + lid].w * ratio;
38             if(lid*VECTOR_SIZE > k)
39                 matrix[group_id*n + lid] -= matrix[k*n + lid] * ratio;
40         }
41         else if(xx2 >= k)
42         {
43             matrix[group_id*n + lid].z -= matrix[k*n + lid].z * ratio;
44             matrix[group_id*n + lid].w -= matrix[k*n + lid].w * ratio;
45             if(lid*VECTOR_SIZE > k)
46                 matrix[group_id*n + lid] -= matrix[k*n + lid] * ratio;
47         }
48         else if(xx3 >= k)
49         {
50             matrix[group_id*n + lid].w -= matrix[k*n + lid].w * ratio;
51             if(lid*VECTOR_SIZE > k)
52                 matrix[group_id*n + lid] -= matrix[k*n + lid] * ratio;
53         }
54
55         if(xx == k)
56             matrix[group_id*n + lid].x = ratio;
57         if(xx1 == k)
58             matrix[group_id*n + lid].y = ratio;
59         if(xx2 == k)
60             matrix[group_id*n + lid].z = ratio;
61         if(xx3 == k)
62             matrix[group_id*n + lid].w = ratio;
63     }
64 }
```

Appendix C

Convolution

C.1 OpenMP

```

1  /*---OpenMP Gaussian Blur Convolution---*/
2  void conv_omp( image_t* src, image_t* temp, image_t* dst, float* filterW, int
    filterR, int imageW, int imageH)
3  {
4      omp_set_num_threads(nrthreads);
5      // warm-up run (code removed)
6      // loop runs
7      tick_count omp_t1 = tick_count::now();
8      for(int i = 0; i < AVG_ITERATIONS; i++)
9      {
10         convRow_omp(src, temp, filterW, filterR, imageW, imageH);
11         convCol_omp(temp, dst, filterW, filterR, imageW, imageH);
12     }
13     tick_count omp_t2 = tick_count::now();
14
15     double time_omp = (double) (omp_t2 - omp_t1).seconds()/(double)(
        AVG_ITERATIONS);
16     cout<<"\tElapsed_Avg_OpenMP_Time=\t"<<time_omp<<endl;
17     ompCollector.addData(imageW, time_omp);
18 }
19 void convRow_omp( image_t* src, image_t* dst, float* filterW, int filterR,
    int imageW, int imageH)
20 {
21     int x, y, k, d;
22     float sum;
23     #pragma omp parallel for default(shared) private(x, y, k, d, sum)
24     for(y = 0; y < imageH; y++)
25     {
26         for(x = 0; x < imageW; x++)
27         {
28             sum = 0;
29             for(k = -filterR; k <= filterR; k++)
30             {
31                 d = x + k;
32                 if(d >= 0 && d < imageW)

```

```

33     sum += src[y * imageW + d] * filterW[filterR - k];
34     }
35     dst[y * imageW + x] = sum;
36 }
37 }
38 }
39 void convCol_omp( image_t* src, image_t* dst, float* filterW, int filterR,
    int imageW, int imageH)
40 {
41     int x, y, k, d;
42     float sum;
43     #pragma omp parallel for default(shared) private(x, y, k, d, sum)
44     for(y = 0; y < imageH; y++)
45     {
46         for(x = 0; x < imageW; x++)
47         {
48             sum = 0;
49             for(k = -filterR; k <= filterR; k++)
50             {
51                 d = y + k;
52                 if(d >= 0 && d < imageH)
53                     sum += src[d * imageW + x] * filterW[filterR - k];
54             }
55             dst[y * imageW + x] = sum;
56         }
57     }
58 }

```

C.2 TBB

```

1  /*---TBB Gaussian Blur Convolution---*/
2  void conv_tbb( image_t* src, image_t* temp, image_t* dst, float* filterW, int
    filterR, int imageW, int imageH)
3  {
4      task_scheduler_init TBBinit(nrthreads);
5      // warm-up run (code removed)
6      // loop runs
7      tick_count tbb_t1 = tick_count::now();
8      for(int i = 0; i < AVG_ITERATIONS; i++)
9      {
10         convRow_tbb(src, temp, filterW, filterR, imageW, imageH);
11         convCol_tbb(temp, dst, filterW, filterR, imageW, imageH);
12     }
13     tick_count tbb_t2 = tick_count::now();
14
15     double time_tbb = (double) (tbb_t2 - tbb_t1).seconds() / (double) (
        AVG_ITERATIONS);
16     cout<<"\tElapsed_Avg_TBB---Time_="<<time_tbb<<endl;
17     tbbCollector.addData(imageW, time_tbb);
18 }
19 void convRow_tbb( image_t* src, image_t* dst, float* filterW, int filterR,
    int imageW, int imageH)
20 {
21     parallel_for(0, imageH, 1, [=](int y) {

```

```

22     for(int x = 0; x < imageW; x++)
23     {
24         float sum = 0;
25         for(int k = -filterR; k <= filterR; k++)
26         {
27             int d = x + k;
28             if(d >= 0 && d < imageW)
29                 sum += src[y * imageW + d] * filterW[filterR - k];
30         }
31         dst[y * imageW + x] = sum;
32     }
33 });
34 }
35 void convCol_tbb( image_t* src, image_t* dst, float* filterW, int filterR,
36                  int imageW, int imageH)
37 {
38     parallel_for(0, imageH, 1, [=](int y) {
39         for(int x = 0; x < imageW; x++)
40         {
41             float sum = 0;
42             for(int k = -filterR; k <= filterR; k++)
43             {
44                 int d = y + k;
45                 if(d >= 0 && d < imageH)
46                     sum += src[d * imageW + x] * filterW[filterR - k];
47             }
48             dst[y * imageW + x] = sum;
49         }
50     });
51 }

```

C.3 OpenCL

```

1
2 /*---Row-wise convolution---*/
3 __kernel void row_convolution(__global float* Src, __global float* Dst, int
4                               imageW, int imageH, __constant float* filterW, int filterR)
5 {
6     int IDx = get_global_id(0);
7     int IDy = get_global_id(1);
8
9     int k, d;
10    float sum = 0.0f;
11
12    for(k = -filterR; k <= filterR; k++)
13    {
14        d = IDx + k;
15        if(d >= 0 && d < imageW)
16            sum += Src[IDy*imageW + d] * filterW[filterR - k];
17    }
18    Dst[IDy * imageW + IDx] = sum;
19 }
20 /*---Column-wise convolution---*/

```

```

21 __kernel void col_convolution(__global float* Src, __global float* Dst, int
    imageW, int imageH, __global float* filterW, int filterR)
22 {
23     int IDx = get_global_id(0);
24     int IDy = get_global_id(1);
25
26     int k, d;
27     float sum = 0.0f;
28
29     for(k = -filterR; k <= filterR; k++)
30     {
31         d = IDy + k;
32         if(d >= 0 && d < imageH)
33             sum += Src[d*imageW + IDx] * filterW[filterR - k];
34     }
35     Dst[IDy * imageW + IDx] = sum;
36 }

```

Appendix D

Histogram Generation

D.1 OpenMP

```
1 void hist_omp(long int* image, int* hist)
2 {
3     int x, y;
4
5     int hist_tmp[INTENSITY_LEVELS];
6     memset(hist_tmp, 0x0, INTENSITY_LEVELS*sizeof(int));
7
8     #pragma omp parallel default(none) shared(image, hist) private(x, y)
9         firstprivate(*HEIGHT, WIDTH,* hist_tmp)
10     {
11         /* calculate one histogram per thread */
12         #pragma omp for
13         for(y=0; y<HEIGHT; y++)
14         {
15             for(x=0; x<WIDTH; x++)
16             {
17                 int image_value = image[x + y*WIDTH];
18                 hist_tmp[image_value] += 1;
19             }
20         }
21
22         /* merge the subhistograms */
23         #pragma omp critical
24         for(int i=0; i<INTENSITY_LEVELS; i++)
25         {
26             hist[i] += hist_tmp[i];
27         }
28     }
```

D.2 TBB

```

1
2 /*-----TBB functor class-----*/
3 class histogram_tbb
4 {
5     long int* my_image;
6     int my_width , my_height;
7 public:
8     int* my_hist;
9
10    histogram_tbb(long int* image, int* hist, int width = WIDTH , int height =
        HEIGHT) : my_image(image), my_hist(hist), my_width(width), my_height(
        height)
11    {
12        memset(my_hist , 0x0, sizeof(int) * INTENSITY_LEVELS);
13    }
14
15    histogram_tbb(histogram_tbb& other, split) : my_image(other.my_image),
16        my_hist(other.my_hist), my_width(other.my_width), my_height(
        other.my_height)
17    {
18        /* scalable_allocator allocates and frees memory in a way that scales
19           with the number of processors */
20        my_hist = (int*) scalable_malloc(sizeof(int) * INTENSITY_LEVELS);
21        memset(my_hist , 0x0, sizeof(int) * INTENSITY_LEVELS);
22    }
23
24    /* calculate one histogram per thread */
25    void operator()(const blocked_range<int>& r)
26    {
27        int width = my_width;
28        int height = my_height;
29        long int* image = my_image;
30
31        for (int y=r.begin(); y!=r.end(); ++y)
32        {
33            for (int x=0; x<width; x++)
34            {
35                int image_value = image[x + y*width];
36                my_hist[image_value] += 1;
37            }
38        }
39    }
40
41    /* merge the subhistograms */
42    void join(const histogram_tbb& other)
43    {
44        for (int i=0; i<INTENSITY_LEVELS; i++)
45        {
46            my_hist[i] += other.my_hist[i];
47        }
48        scalable_free(other.my_hist);
49    }
50 };
51
52 // Invoking function object
53 void hist_tbb(long int* image, int* hist)

```

```

54 {
55     histogram_tbb hist_funcior(image, hist, WIDTH , HEIGHT);
56     parallel_reduce(blocked_range <int>(0, HEIGHT), hist_funcior,
57                     auto_partitioner());

```

D.3 OpenCL

```

1  #define LINEAR_MEM_ACCESS
2  #pragma OPENCL EXTENSION cl_khr_byte_addressable_store : enable
3
4  #define BIN_SIZE 256
5
6  __kernel void histogram256(__global const uint* data,
7                             __local uchar* sharedArray,
8                             __global uint* binResult)
9  {
10     size_t localId = get_local_id(0);
11     size_t globalId = get_global_id(0);
12     size_t groupId = get_group_id(0);
13     size_t groupId = get_group_id(0);
14     size_t groupId = get_group_id(0);
15     size_t groupId = get_group_id(0);
16     size_t groupId = get_group_id(0);
17     size_t groupId = get_group_id(0);
18
19     /* initialize shared array to zero */
20     for(int i = 0; i < BIN_SIZE; ++i)
21         sharedArray[localId * BIN_SIZE + i] = 0;
22
23     barrier(CLK_LOCAL_MEM_FENCE);
24
25     /* calculate thread-histograms */
26     for(int i = 0; i < BIN_SIZE; ++i)
27     {
28         #ifdef LINEAR_MEM_ACCESS
29             uint value = data[groupId * groupId * BIN_SIZE + i * groupId +
30                             localId];
31         #else
32             uint value = data[globalId * BIN_SIZE + i];
33         #endif LINEAR_MEM_ACCESS
34         sharedArray[localId * BIN_SIZE + value]++;
35     }
36
37     barrier(CLK_LOCAL_MEM_FENCE);
38
39     /* merge all thread-histograms into block-histogram */
40     /* All block histograms are merged to final histogram in the host*/
41     for(int i = 0; i < BIN_SIZE / groupId; ++i)
42     {
43         uint binCount = 0;
44         for(int j = 0; j < groupId; ++j)
45             binCount += sharedArray[j * BIN_SIZE + i * groupId + localId];
46
47         binResult[groupId * BIN_SIZE + i * groupId + localId] = binCount;
48     }
49 }

```


Appendix E

Pi Calculation

E.1 OpenMP

```

1
2 void calcPi_omp(double& pi, const long N)
3 {
4     omp_set_num_threads(nrthreads);
5     // warm-up run (code removed)
6
7     // looping run
8     tick_count t1 = tick_count::now();
9     for(int i = 0; i < ITERATIONS; i++)
10    {
11        my_pi = 0.0;
12        w = 1.0 / N;
13        #pragma omp parallel for default(shared) reduction(+ : my_pi) schedule(static
14        )
15        for(long i = 0; i < N; i++)
16        {
17            double local = (i + 0.5) * w;
18            my_pi += 4.0 / (1.0 + local * local);
19        }
20        pi = my_pi * w;
21    }
22    tick_count t2 = tick_count::now();
23    double omp_time = (t2 - t1).seconds() / (double) ITERATIONS;
24    cout<<"Elapsed_pi_omp_time_="<<omp_time<<endl;
25    ompCollector.addData(N, omp_time);
26 }

```

E.2 TBB

```

1 class classPi
2 {
3     double my_w;

```

```

4 public:
5     double my_pi;
6     classPi(double W, double value = 0.0) : my_w(W), my_pi(value) { }
7     classPi(classPi& other, split) : my_w(other.my_w), my_pi(0.0) { }
8     void operator() (const blocked_range<long>& r)
9     {
10         double pi = my_pi;
11         double w = my_w;
12         int end = r.end();
13         for(long i = r.begin(); i != end; i++)
14         {
15             double local = (i + 0.5) * w;
16             pi += 4.0 / (1.0 + local * local);
17         }
18         my_pi = pi;
19     }
20
21     void join(const classPi& other) { my_pi += other.my_pi; }
22
23 };
24
25 // Invoking function object
26 void calcPi_tbb(double& pi, const long N)
27 {
28     task_scheduler_init TBBinit(nrthreads);
29     // warm-up run (code removed)
30
31     // looping run
32     tick_count t1 = tick_count::now();
33     for(int i = 0; i < ITERATIONS; i++)
34     {
35         w = 1.0 / N;
36         classPi myClass(w);
37         parallel_reduce(blocked_range<long>(0, N), myClass, auto_partitioner())
38         ;
39         pi = myClass.my_pi * w;
40     }
41
42     tick_count t2 = tick_count::now();
43     double tbb_time = (t2 - t1).seconds() / (double) ITERATIONS;
44     cout<<"Elapsed_pi_tbb_time="<<tbb_time<<endl;
45     tbbCollector.addData(N, tbb_time);
46 }

```

E.3 OpenCL

```

1 #ifdef KHR_DP_EXTENSION
2 #pragma OPENCL EXTENSION cl_khr_fp64 : enable
3 #else
4 #pragma OPENCL EXTENSION cl_amd_fp64 : enable
5 #endif
6
7 #define VECTOR_SIZE 4
8

```

```

9  __kernel void pi_calculation(__global double* pi, uint iters)
10 {
11     uint IDx = get_global_id(0);
12     uint n_work_items = get_global_size(0);
13
14     double w = 1.0 / (double) (n_work_items*iters);
15     double temp = 0.0;
16     double local_num = 0.0;
17     for(uint i = IDx * n_work_items ; i < (IDx * n_work_items + iters); i++)
18     {
19         local_num = (i + 0.5) * w;
20         temp += 4.0 / (1.0 + local_num * local_num);
21     }
22     pi[IDx] = temp;
23 }
24
25 __kernel void pi_calculation_vec(__global double4* pi, uint iters)
26 {
27     uint IDx = get_global_id(0);
28     uint n_vec_items = get_global_size(0);
29     uint n_work_items = n_vec_items * VECTOR_SIZE;
30
31     double w = 1.0 / (double) (n_work_items * iters);
32     double4 temp = (double4) (0.0, 0.0, 0.0, 0.0);
33     double4 num = (double4) (0.0, 0.0, 0.0, 0.0);
34
35     uint start = IDx * n_work_items * VECTOR_SIZE;
36     uint end = (IDx * n_work_items * VECTOR_SIZE) + (iters * VECTOR_SIZE);
37
38     for(uint i = start ; i < end; i += VECTOR_SIZE)
39     {
40         num = ((i+0.5)*w, (i+1+0.5)*w, (i+2+0.5)*w, (i+3+0.5)*w);
41         temp += 4.0 / (1.0 + num * num);
42     }
43     pi[IDx] = temp;
44 }
45
46 __kernel void reduce(__global double* pi)
47 {
48     uint IDx = get_global_id(0);
49     size_t size_x = get_global_size(0);
50     pi[0] += pi[IDx];
51 }


```

Bibliography

- [1] A. Ali, L. Johnsson, J. Subhlok. Scheduling FFT Computation on SMP and Multi-core Systems. *21st International Conference on Supercomputing, ICS, June 2007.*
- [2] A. Ali, U. Dastgeer and C. Kessler. OpenCL for programming shared memory multicore CPUs. *4th Swedish Workshop on Multicore Computing (MCC'11), November 2011.*
- [3] A. Ali, U. Dastgeer and C. Kessler. OpenCL for programming shared memory multicore CPUs. *Proceedings of the 5th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG'12) at HiPEAC-2012, February 2012.*
- [4] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. *Proceedings of Supercomputing '90, pp. 2-11, 1990.*
- [5] A. Buchau, S.M. Tsafak, W. Hafla, W.M. Rucker. Parallelization of a Fast Multipole Boundary Element Method with Cluster OpenMP. *IEEE Transactions on Magnetics, vol.44, no.6, pp. 1338-1341, June 2008.*
- [6] B. Chapman, L. Huang. Enhancing OpenMP and Its Implementation for Programming Multicore Systems. *Parallel Computing: Architectures, Algorithms and Applications, pp. 3-18, 2007.*
- [7] J. Chen, K. Ji, Z. Shi, W. Liu. Implementation of Block Algorithm for LU Factorization. *WRI World Congress on Computer Science and Information Engineering, Volume 2, pp. 569-573, April 2009.*
- [8] J.B.T. Correa, D.O. Penha, C.A.P.S. Martins. Analysis of performance optimization techniques on digital image convolution. *6th World Multiconference on Systemics, Cybernetics and Informatics, Volume 9, pp. 45-50, July 2002.*
- [9] L. F. Cupertino, A. P. Singulani, C. P. da Silva, M. A. C. Pacheco. LU Decomposition on GPUs: The Impact of Memory Access. *22nd International Symposium on Computer Architecture and High Performance Computing Workshops, pp. 19-24, October 2010.*
- [10] A. Dorta, C. Rodriguez, F. Sande and A. Escribano. The OpenMP Source Code Repository. *Proceedings-13th Euromicro Conference on Parallel, Distributed and Network-based Processing, pp. 244-50, February 2005.*
- [11] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, J. Dongarra. From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming. *Technical report, Department of Computer Science, UTK, Knoxville Tennessee, September 2010.*

- [12] J. Gervasi, D. Russo, F. Vella. The AES implantation based on OpenCL for multi-/many core architecture. *International Conference on Computational Science and Its Applications*, pp.129-134, 2010.
- [13] K. Karimi, N. Dickson and F. Hamze. A Performance Comparison of CUDA and OpenCL. *CoRR*, 2010.
- [14] P. Kegel, M. Schellmann, and S. Gorlatch. Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-cores. *Euro-Par 2009. Parallel Processing-15th International Euro-Par Conference*, pp.654-65, August 2009.
- [15] D. Kim, V. Lee, and Y. Chen. Image Processing on Multicore x86 Architectures. *Signal Processing Magazine, IEEE*, pp. 97-107, March 2010.
- [16] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa and H. Kobayashi. Evaluating Performance and Portability of OpenCL Programs. *Fifth International Workshop on Automatic Performance Tuning*, June 2010.
- [17] J. Lee, J. Kim, S. Seo. An OpenCL Framework for Homogeneous Manycores with No Hardware Cache Coherence. *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 56-67, October 2011.
- [18] J. Ma, K. Li, L. Zhang. Parallel Floyd-Warshall algorithm based on TBB. *2nd IEEE International Conference on Information Management and Engineering (ICIME 2010)*, pp. 429-433, April 2010.
- [19] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Comparison of Parallelization Frameworks for Shared Memory Multi-Core Architectures. *Proceedings of the Embedded World Conference, Nuremberg, Germany, March 2010*.
- [20] R. Membarth , F. Hannig , J. Teich, M. Körner, and W. Eckert. Frameworks for Multi-core Architectures: A Comprehensive Evaluation using 2D/3D Image Registration. *Architecture of Computing Systems, 24th International Conference*, pp. 62-73, February 2011.
- [21] P. Michailidis, K Margaritis. Performance Models for Matrix Computations on Multicore Processors using OpenMP. *The 11th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pp. 375-80, December 2010.
- [22] V. Podlozhnyuk. Image Convolution with CUDA. *NVIDIA white paper*, pp. 2-21, June 2007.
- [23] A. Prabhakar, V. Getov and B. Chapman. Performance Comparisons of Basic OpenMP Constructs. *High Performance Computing-4th International Symposium, ISHPC 2002. Proceedings*, pp. 413-24, May 2002.
- [24] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O Reilly Media, Inc. 2007.
- [25] E. E. Santos. Parallel Complexity of Matrix Multiplication. *The Journal of Supercomputing, Volume 25*, pp. 155-175, June 2003.
- [26] S. Seo, G. Jo and J. Lee. Performance characterization of the NAS Parallel Benchmarks in OpenCL. *IEEE International Symposium on Workload Characterization (IISWC)*, November 2011.
- [27] J. Shen, J. Fang, A. L. Varbanescu , and H. Sips. OpenCL vs. OpenMP: A Programmability Debate.

- [28] J. Shen, J. Fang, H. Sips, and A. L. Varbanescu. Performance Traps in OpenCL for CPUs. *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 38-45, February-March 2013.
- [29] J. Stone, D. Gohara and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Co-published by the IEEE CS and the AIP*, 2010.
- [30] J. Yang, H. Zheng, Y. Xie, J. Wang, N. Bao. Adding TBB Contents to The Multi-core Related Curriculums. *First International Conference on Intelligent Networks and Intelligent Systems*, pp. 685-688, November 2008.
- [31] Intel Threading Building Blocks Documentation, User Guide, Containers. http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm 15:41, May 6, 2013.
- [32] <http://www.daniweb.com/software-development/csharp/code/355645/optimizing-matrix-multiplication> 15:07, November 10, 2012.
- [33] http://www.cs.hmc.edu/~geoff/classes/hmc.cs105.201001/slides/class11_cache.ppt Slides 27-36, 15:07, November 10, 2012.

 Avdelning, Institution Division, Department Institutionen för datavetenskap, Dept. of Computer and Information Science 581 83 Linköping		Datum Date 2013-06-20
Språk Language <input type="checkbox"/> Svenska/Swedish <input checked="" type="checkbox"/> Engelska/English <input type="checkbox"/> _____	Rapporttyp Report category <input type="checkbox"/> Licentiatavhandling <input checked="" type="checkbox"/> Examensarbete <input type="checkbox"/> C-uppsats <input type="checkbox"/> D-uppsats <input type="checkbox"/> Övrig rapport <input type="checkbox"/> _____	ISBN ISBN ISRN LIU-IDA/LITH-EX-A-13/039-SE Serietitel och serienummer ISSN Title of series, numbering - _____ Linköping Studies in Science and Technology Thesis No. 039
URL för elektronisk version http://www.liu.diva-portal.org		
Titel Title Comparative study of parallel programming models for multicore computing Författare Author Akhtar Ali		
Sammanfattning Abstract <p>Shared memory multi-core processor technology has seen a drastic development with faster and increasing number of processors per chip. This new architecture challenges computer programmers to write code that scales over these many cores to exploit full computational power of these machines. Shared-memory parallel programming paradigms such as OpenMP and Intel Threading Building Blocks (TBB) are two recognized models that offer higher level of abstraction, shields programmers from low level details of thread management and scales computation over all available resources. At the same time, need for high performance power-efficient computing is compelling developers to exploit GPGPU computing due to GPU's massive computational power and comparatively faster multi-core growth. This trend leads to systems with heterogeneous architectures containing multicore CPUs and one or more programmable accelerators such as programmable GPUs. There exist different programming models to program these architectures and code written for one architecture is often not portable to another architecture. OpenCL is a relatively new industry standard framework, defined by Khronos group, which addresses the portability issue. It offers a portable interface to exploit the computational power of a heterogeneous set of processors such as CPUs, GPUs, DSP processors and other accelerators.</p> <p>In this work, we evaluate the effectiveness of OpenCL for programming multi-core CPUs in a comparative case study with two CPU specific stable frameworks, OpenMP and Intel TBB, for five benchmark applications namely matrix multiply, LU decomposition, image convolution, Pi value approximation and image histogram generation. The evaluation includes a performance comparison of the three frameworks and a study of the relative effects of applying compiler optimizations on performance numbers. OpenCL performance on two vendor-dependent platforms Intel and AMD, is also evaluated. Then the same OpenCL code is ported to a modern GPU and its code correctness and performance portability is investigated. Finally, usability experience of coding using the three multi-core frameworks is presented.</p>		
Nyckelord Keywords OpenCL, OpenMP, TBB, Multicore frameworks		