OpenCL on shared memory multicore CPUs

Akhtar Ali, Usman Dastgeer and Christoph Kessler

Published in: Proc. Fifth Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-2012) at HiPEAC-2012 conference, Paris, France, Jan. 2012.

OpenCL for programming shared memory multicore CPUs

Akhtar Ali, Usman Dastgeer, and Christoph Kessler

PELAB, Dept. of Computer and Information Science,
Linköping University, Sweden
`akhal935@student.liu.se`
`{usman.dastgeer,christoph.kessler}@liu.se`

Abstract. Shared memory multicore processor technology is pervasive in mainstream computing. This new architecture challenges programmers to write code that scales over these many cores to exploit the full computational power of these machines. OpenMP and Intel Threading Building Blocks (TBB) are two of the popular frameworks used to program these architectures. Recently, OpenCL has been defined as a standard by Khronos group which focuses on programming a possibly heterogeneous set of processors with many cores such as CPU cores, GPUs, DSP processors.

In this work, we evaluate the effectiveness of OpenCL for programming multicore CPUs in a comparative case study with OpenMP and Intel TBB for five benchmark applications: matrix multiply, LU decomposition, 2D image convolution, Pi value approximation and image histogram generation. The evaluation includes the effect of compiler optimizations for different frameworks, OpenCL performance on different vendors' platforms and the performance gap between CPU-specific and GPU-specific OpenCL algorithms for execution on a modern GPU. Furthermore, a brief usability evaluation of the three frameworks is also presented.

1 Introduction

Shared memory multicore processor technology evolved in response to the physical limiting factors such as clock speed and heat/power problems with single processor technology. But this shift of hardware technology from single core to many cores required developers to write programs with parallelism to exploit all available cores. Parallel programming paradigms abstracting thread management activities evolved over time, such as OpenMP and Intel's TBB, to help developers divide single algorithm workload on different cores. Because of the specialized architecture of GPUs, which has higher arithmetic intensity and is thus more suitable for data parallel applications, the notion of general purpose GPU computing (GPGPU) has emerged at the same time. With growing focus on parallel computing, an environment of heterogeneous hardware architectures became inevitable and the need for a programming framework offering a uniform interface to these heterogeneous architectures was realized. Arisen from this need is the Open Computing Language (OpenCL) standard API that is based on ISO C99 standard. OpenCL abstracts the underlying hardware and enables developers to write code that is portable across different shared-memory architectures.

Individually, these frameworks have been studied extensively such as OpenMP in [2, 12], TBB in [6, 13] and OpenCL in [7, 8]. There is still lot of work focused only on CPU specific [9, 10] and GPU specific [1, 4, 5] frameworks comparisons. But there is comparatively little work evaluating OpenCL CPU performance with that of state-of-the-art CPU specific frameworks. Evaluating OpenCL on CPUs is becoming increasingly important as OpenCL implementations are available for CPUs by AMD and Intel. Due to its promise for code portability, OpenCL could provide the programming foundation for modern heterogeneous systems. An important study in this direction is [3] examining overhead, scalability and usability of five shared memory parallelism frameworks including OpenCL on a 2D/3D image registration application. In our work, we choose OpenMP and TBB as popular representatives of CPU specialized frameworks and compare these with OpenCL, on five benchmark applications to provide a wider base for comparison. 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, compiler support, programming effort and code size.
- OpenCL specific issues: such as performance implications of different vendor's OpenCL' implementations, CPU-specific vs GPU-specific optimizations for an OpenCL algorithm and performance implications of these optimizations.

2 Frameworks

2.1 OpenMP

Open Multi Processing (OpenMP) provides parallelizing facilities in C/C++ and Fortran by using preprocessor 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 [10]. Although there exists a commercial implementation of OpenMP on clusters [2], its prime target are shared memory architectures. It provides a very easy to use high level application programming interface for multi-threading computationally intensive data and task parallel programs. Most OpenMP implementations use thread pools on top of the fork/join model, which exist throughout execution of the program and therefore avoid the overhead of thread creation and destruction after each parallel region [11]. Its work sharing constructs such as `omp for` in C/C++ (and `omp do` in Fortran) provide data parallelism while `omp sections` [3] and the `omp task` construct in OpenMP 3.0 support task parallelism.

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. The directive based approach and support for incremental parallelism makes this model easy to use for parallelizing new as well as existing applications [12].

2.2 Intel TBB

Intel Threading Building Blocks (TBB) is Intel's technology for parallelism, comprising a template library for C++. Users have to specify logical parallelism and the TBB runtime library maps it into threads ensuring efficient usage of available resources [6]. 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. It also provides concurrent containers such as `concurrent_queue` and `concurrent_vector` that makes it even more suitable for data parallel programming.

In contrast to some 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 technique shows better results than the former one in complex situations. It also fits well into the task stealing scheduler [6]. OpenMP also allows dynamic scheduling but in simple cases when fixed amount of work is known ahead of time, static scheduling is recommended since it incurs less run-time overhead than dynamic scheduling.

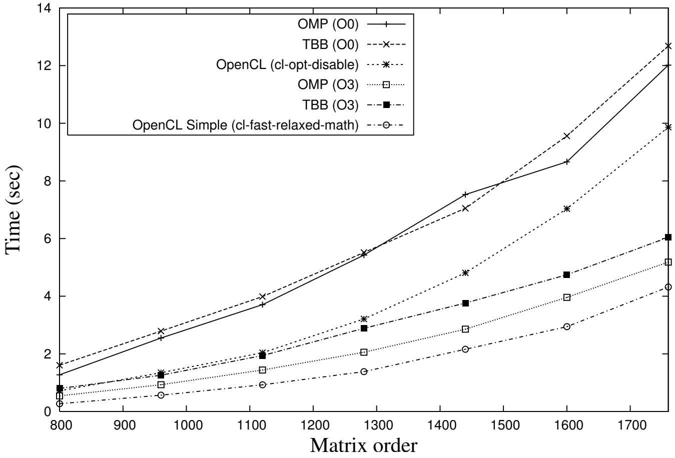
2.3 OpenCL

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 [3]. The OpenCL code is just-in-time (JIT) compiled during runtime which prevents dependencies of instructions set and supporting libraries. The dynamic compilation enables better utilization of the underlying device' latest software and hardware features such as SIMD capability of hardware [7].

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 [7].

The OpenCL memory model consists of host side memory and four types of memories on device side: global, constant, local and private. Every single element of the problem domain is called $work-item$ while some work-items can be grouped together to form a $work-group$. 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

Figure 1: Matrix Multiplication Performance of OMP, TBB, OpenCL



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. The host program running on the host CPU creates memory objects in global memory using OpenCL APIs while en-queuing memory commands operating on these memory objects which can be synchronized using command-enqueue barriers or context events [8]. AMD, Intel, IBM, Nvidia and Apple are some well-known vendors who have implemented the OpenCL standard.

3 Applications

Five benchmarks, namely matrix multiplication, LU factorization, 2D Gaussian image convolution, Pi value approximation and histogram generation are chosen for investigating parallelism frameworks. Matrix computations such as matrix multiplications are widely used in scientific computing [14]. Linear algebra computations are ubiquitous in science and engineering and have applications in e.g., graphics programming, artificial intelligence and data mining. LU factorization is a technique for solving linear equation systems. The algorithm used derives lower and upper triangles in the same input matrix by using the row-major order variant of the right-looking algorithm that accesses consecutive elements of the matrix, which improves performance due to cache effects.

Convolution and histogram generation are important building blocks of many image processing applications. Convolution is used to apply different kinds of effects such as blur, sharpen or emboss [15]. 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 doing only $n+n$ multiplications compared to $n \times n$ multiplications in the 2D case where n is the filter size in each dimension. Histograms are used to compare images or to modify their intensity spectrum [9]. For a simple gray-scale image, a histogram generation function maps the frequency of the intensity levels in an image to the gray-level range. Pi approximation computes the area under the curve $y = 4/(1 + x^2)$ between 0 and 1. The only input parameter is the precision control value.

Figure 2: LU Decomposition Performance of OMP, TBB, OpenCL

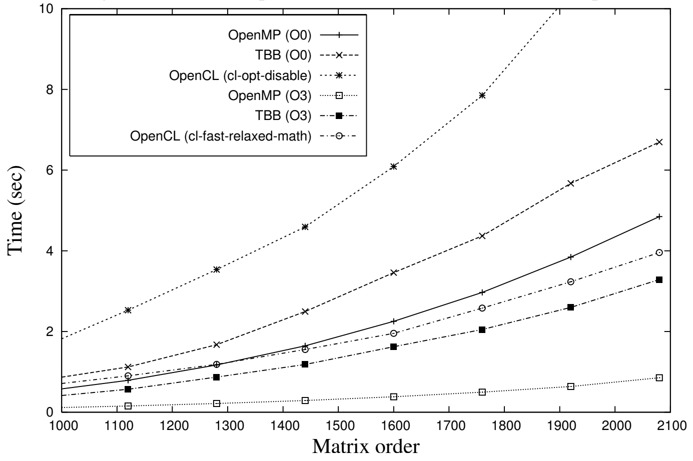
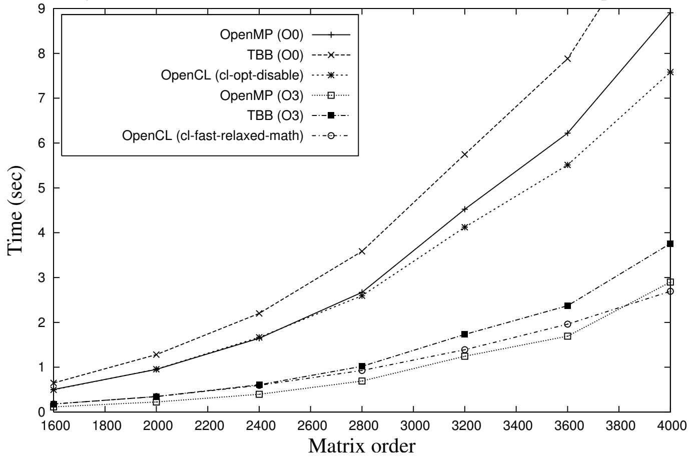


Figure 3: Gaussian Convolution Performance of OMP, TBB, OpenCL



4 Evaluation

Our test applications are parallelized from the sequential C/C++ implementations by applying standard CPU specific optimizations available in each framework to experience the programming complexity of these frameworks. For OpenCL implementations, optimizations such as vectorization and AoS (Array of Structures) data storage format are applied to improve performance on CPUs. 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.

4.1 Performance and compiler optimizations

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. 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 executions with *disabled* compiler optimizations, option `-O0` is used during the program compilation. For OpenCL dynamic compilation, a comparable effect is achieved by passing the `-cl-opt-disable` compiler option. For executions with *aggressive* compiler optimizations, 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 loops blocking to improve cache usage. For OpenCL dynamic compilation, similar effect is achieved by passing the `-cl-fast-relaxed-math`[2] option.In the following, we discuss effects of compiler optimizations for different benchmarks.

Figure 1 shows that OpenCL outperforms the other two in matrix multiplication at both disabled and aggressive optimization levels while OpenMP beats TBB when aggressive compiler optimizations are enabled.

Figure 2 shows that OpenMP yields 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 image 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 aggressive optimization level, which means that OpenMP benefited more from compiler optimization than TBB as shown in Figure 2.

For 2D image convolution, 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 3. It also demonstrates that the performance gap among the three frameworks narrows with compiler optimizations.

Pi value approximation uses reduction in which TBB presents better performance than OpenMP with no compiler optimizations while OpenCL shows the best performance as Figure 4 indicates. The graph clearly demonstrates that an explicitly vectorized OpenCL kernel significantly beats a simple OpenCL kernel and other models in speedup. With the Intel compiler's default optimization

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

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

level, there is a narrowed gap between OpenMP and TBB performances while the OpenCL kernel is still way faster. This shows that enabling explicit vectorization in OpenCL enhances speedup on CPUs.

Histogram generation is also a reduction application and is run at default compiler optimization level. Its performance graph is shown in Figure 5 where TBB shows slightly low performance while OpenMP and OpenCL match well for the whole range of matrix size. This performance behavior of the frameworks is similar to the compiler optimized version of convolution.

Figure 4: Comparing PI Performance of OMP, TBB, OpenCL

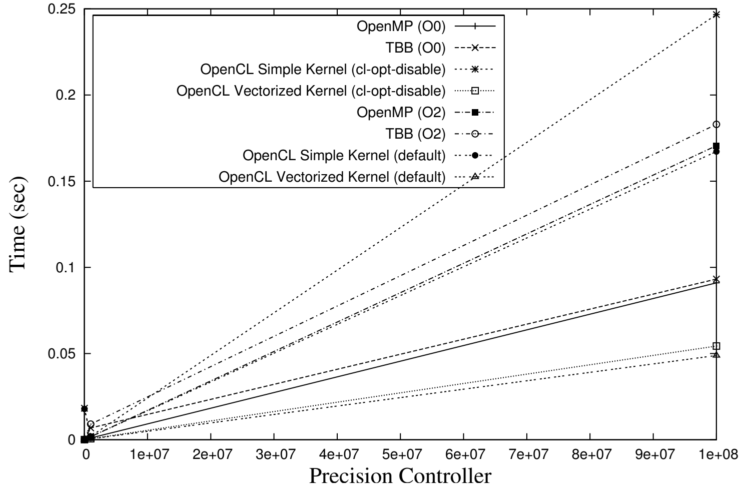
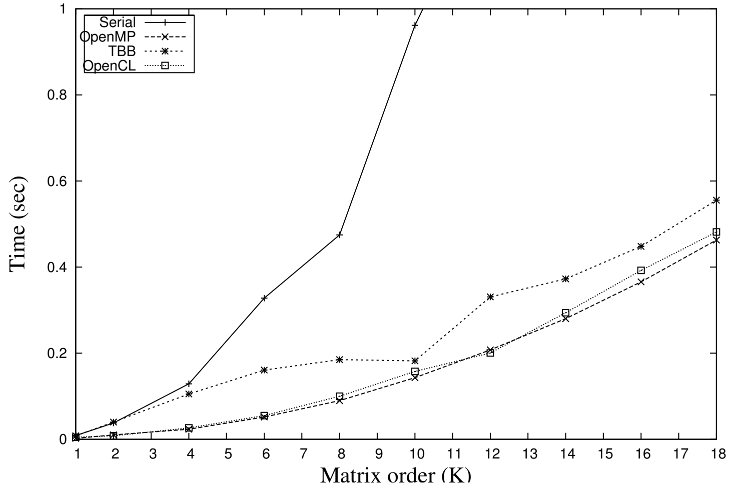


Figure 5: Histogram Generation Performance of OMP, TBB, OpenCL



4.2 Scalability Evaluation

OpenCL involves the highest overhead in LU factorization and shows performance improvement after five cores as shown in Figure 6. TBB incurs comparatively low overhead for a small number of cores but scales almost like OpenCL for a high number of cores. OpenMP beats TBB and OpenCL significantly and scales best for all numbers of cores, incurring zero overhead for a single core.

Figure 7 illustrates that TBB and OpenCL incur some overhead on a single core but all the three frameworks demonstrate identical performance when multiple cores are used. OpenMP with loops statically scheduled shows no overhead for one core and scales better for all number of cores compared to dynamically scheduled loops in TBB. 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. OpenCL shows some overhead degradation on single core but it catches up well with OpenMP compared to TBB when two or more cores are used.

These results somewhat conform with [10] but are in contrast with [3] 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 was well suited in convolution and LU decomposition algorithms since these applications have a fixed number of loop iterations with equal distribution of work across different loop iterations.

Figure 6: LUD Performance Scaling

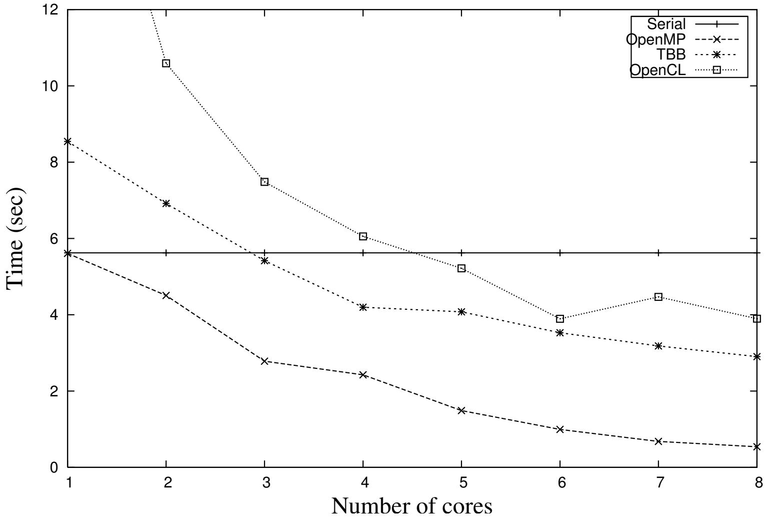
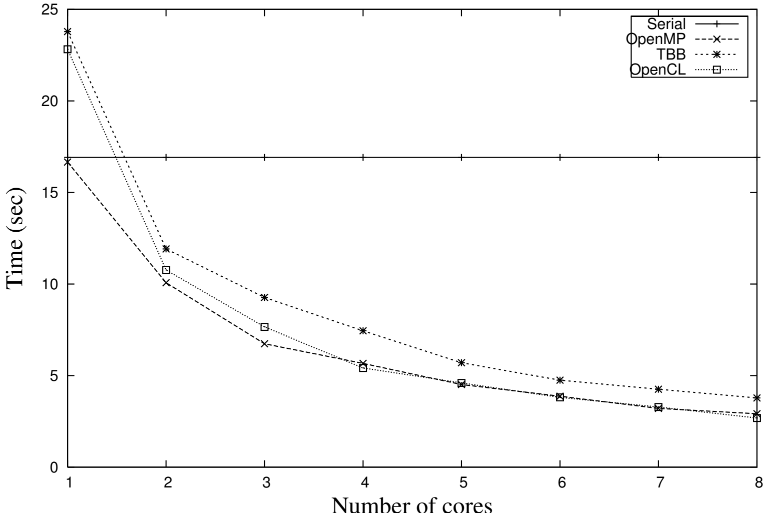


Figure 7: Image Convolution Performance Scaling



4.3 OpenCL Platforms Evaluation

In this section, we evaluate two OpenCL implementations available for multicore CPUs: AMD OpenCL SDK 2.5 and Intel OpenCL SDK LINUX 1.1, experimented on a system with Intel Xeon CPU E5520 hosting 16 cores, each running at 2.27 GHz, and a compiler gcc version-4.6.1. Interestingly Intel's OpenCL outperformed AMD in the three out of four tested applications with a clear margin in performance test on CPUs as shown by Figure 8. This behavior suggests that Intel enables its CPU specific auto-optimizations namely, auto-vectorization, using its JIT OpenCL C compiler. 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 8, the vectorized kernel increased Intel's speedup by 125% and that of AMD by around 320%. This significant relative difference for AMD shows that the Intel JIT OpenCL C compiler was already exploiting auto-vectorization to some extent on Intel CPUs when it was not explicitly programmed in the kernel.

4.4 Code and performance portability

Experiments in this section are performed on an Nvidia Tesla M2050 GPU with Nvidia OpenCL driver 280.13 which implements the OpenCL 1.1 specification.

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

Although the code is portable, the performance portability is highly unlikely as optimizations for an OpenCL implementation on CPUs and GPUs are quite different. We have seen during our experiments that most OpenCL implementations designed specifically for GPUs actually performed poorly on the multicore

CPUs, sometimes even worse than a sequential implementation. In the following, we check the performance portability by comparing performance of OpenCL algorithms that are primarily written for CPU execution with optimized GPU implementations of the same applications on the GPU. The GPU-optimized implementations are taken from NVIDIA OpenCL SDK code samples. The intent is to see the extent of performance portability when single OpenCL code base written for CPUs is naively transported to another common architecture (GPUs).

Figure 9 shows performance comparisons of OpenCL implementations written for CPU and GPU execution on the GPU. The speedups are calculated with respect to sequential CPU execution on a single core. The results can be interpreted in two ways:

- The performance portability is by far sub-optimal to this point. Still, there is a significant performance impact of architecture-specific optimizations. For Histogram, the difference is up to 15 times which is quite substantial.
- On the positive note, not only the code is portable, but it also gives some performance improvements when executed on a GPU. For applications such as image convolution, the speedups increased from 5 times to 50 times when running the same implementation on a powerful GPU (see Figure 8 and Figure 9).

	MM		LU		CONV		PI		Histogram	
	T/E	LOC	T/E	LOC	T/E	LOC	T/E	LOC	T/E	LOC
OpenMP	+++	45	++	60	+++	70	+++	32	+	28
TBB	++	45	- - -	59	+	53	+	42	-	58
OpenCL	++	120/108	- -	120/225	-	120/186	-	120/128	- - -	120/150

Table 1: Time/effort (T/E) and number of lines of code for given bench- marks for the three frameworks. MM(Matrix Multiplication), LU(LU Factorization), CONV(Image convolution). [+++ : Least time/effort - - - : Most time/effort]

Figure 8: OpenCL vendors platforms evaluation on CPUs

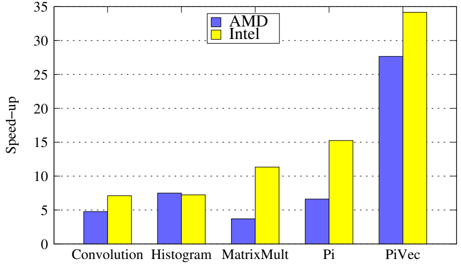
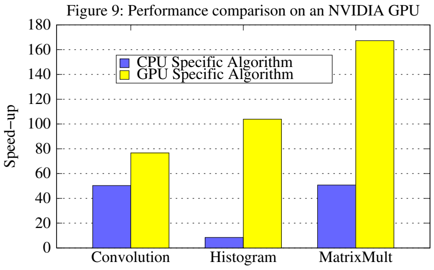


Figure 9: Performance comparison on an NVIDIA GPU



4.5 Usability Evaluation

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 affects readability since they commit additional hierarchies to the nested loops thereby making the scope of every loop and construct somewhat more complex.

TBB needs paralleled portions of a program to be encapsulated in template classes which required medium effort. It moves parallel 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 [10], compared to OpenMP variable sharing, was to select the right variables to be included in `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. However, TBB 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 the GPU programming fashion constitutes quite a different style of programming. It involves a fixed amount of code and effort to initialize the environment which was the 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. Introducing explicit vectorization in the kernel, however, was worth the effort. 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 number of lines of code that is fixed for all algorithms (initializing OpenCL platform, devices, contexts, queues etc.) while the second value shows remaining OpenCL lines of code and is variable for every application.

5 Discussion and Conclusion

We evaluated the applicability of the OpenCL programming model for multicore CPUs on multiple benchmarks in comparison with two state-of-the-art CPU specialized frameworks: OpenMP and Intel TBB. The overall performance and scalability numbers showed that OpenCL on CPU yields competitive performance when compared with the CPU specific frameworks. Enabling compiler optimizations had significant impacts on performance particularly with OpenMP since it takes a compiler directive based approach to parallelism. OpenCL compiler has optimization options for tuning performance similar to other frameworks. Enabling explicit vectorization in OpenCL kernels improved performance while the use of local/private memory did not yield major performance improvements.

On Intel CPUs, the Intel platform outperformed AMD in four out of five tested applications. This shows that Intel's OpenCL has better implicit optimizations for Intel CPUs. OpenCL code written for CPUs could successfully run on an Nvidia platform with Tesla GPUs which shows OpenCL code portability and correctness of execution across different platforms and hardware architectures. However, the performance portability is still by far sub-optimal. The performance of CPU specific code on GPU reasonably surpassed its CPU performance but it did not utilize the capacity of highly data parallel GPUs as GPU optimized algorithms did. This is mainly due to substantial architectural differences between CPU and GPU which makes it difficult to write one algorithm that can efficiently execute on each platform. Furthermore, the implementation experience with all three frameworks is documented with a usability evaluation.

The comparative multicore CPU performance of OpenCL at the cost of adopting a different programming style may be worth the effort since OpenCL offers code portability across a number of different architectures. This may have significant implications in future high performance computing with the addition of data parallel multicore GPUs entering into mainstream computing. Having a single code-base that can execute on variety of hardware architectures although giving sub-optimal performance can still become desirable due to cost issues associated with maintaining separate code-bases. In future, we aim at testing our OpenCL code on AMD CPUs and AMD GPUs to complement our study for comparison of OpenCL performance on vendor-dependent platforms and their implicit optimizations for their own hardware architectures.

Acknowledgements: Funded by EU FP7, project PEPPHER, grant #248481 (www.peppher.eu).

References

1. P. Du et al. *From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming.* Technical report, Dept. of Comp. Science, UTK. 2010.
2. B. Chapman, L. Huang. *Enhancing OpenMP and Its Implementation for Programming Multicore Systems.* Parallel Computing: Architectures, Algorithms and Applications, 2007.
3. R. Membarth et al. *Frameworks for Multi-core Architectures: A Comprehensive Evaluation using 2D/3D Image Registration.* Architecture of Computing Systems, 24th Int. Conf. 2011.
4. K. Komatsu et al. *Evaluating Performance and Portability of OpenCL Programs.* 5th Intl. workshop on Automatic Performance Tuning. 2010.
5. K. Karimi et al. *A Performance Comparison of CUDA and OpenCL.* CoRR, 2010.
6. J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism.* O Reilly Media, Inc. 2007.
7. J. Stone et al. *OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems.* Co-published by the IEEE CS and the AIP, 2010.
8. J. Gervasi et al. *The AES implantation based on OpenCL for multi/many core architecture.* Intl. Conf. on Computational Science and Its Applications, 2010.
9. R. Membarth et al. *Comparison of Parallelization Frameworks for Shared Memory Multi-Core Architectures.* Proc. of the Embedded World Conference, Nuremberg, Germany, 2010.
10. P. Kegel et al. *Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-cores.* Euro-Par 2009. Parallel Processing-15th 15th Int. Euro-Par Conference, 2009.
11. A. Ali, L. Johnsson, J. Subhlok. *Scheduling FFT Computation on SMP and Multicore Systems.* 21st Intl. Conf. on Supercomputing, ICS, 2007.
12. A. Prabhakar et al. *Performance Comparisons of Basic OpenMP Constructs.* ISHPC 2002.
13. J. Ma et al. *Parallel Floyd-Warshall algorithm based on TBB.* 2nd IEEE Int. Conf. on Inf. Management and Engineering, 2010.
14. P. Michailidis, K Margaritis. *Performance Models for Matrix Computations on Multicore Processors using OpenMP.* The 11th Intl. Conf. on Parallel and Distributed Computing, Applications and Technologies, 2010.
15. D. Kim et al. *Image Processing on Multicore x86 Architectures.* Signal Processing Magazine, IEEE, 2010.