Exploration of Parallelization Frameworks for Computational Finance

Raj B Krishnamurthy[1], **Ikubin Chin**[2] **and Anjil Chinnapatlolla**[3]
[1]Exploratory Systems Lab, IBM Systems and Technology Group, Poughkeepsie, NY, USA
[2]Business Analytics and Optimization Lab, IBM Software Group, Tokyo, Japan
[3]System Software Lab, IBM Systems and Technology Group, Bangalore, India

Abstract *- This paper presents a comparison of parallelization frameworks for efficient execution of computational finance workloads. We use a Value-at-Risk (VaR) workload to evaluate OpenCL and OpenMP parallelization frameworks on multi-core CPUs as opposed to GPUs. In addition, we study the impact of SMT on performance using GCC (4.4) and IBM XLC (11.01) compilers for both single-precision and double-precision codes. We use an 8-core, 4-way SMT IBM Power7 with Linux (RHEL 6.0, 2.6.32 kernel) to evaluate OpenCL and OpenMP. Using the IBM XLC compiler, 2-way SMT is able to provide over 30% average improvement as compared to 1 SMT thread per core, whereas, 4-way SMT is able to provide over 50% average improvement as compared to 1 SMT thread per core.*

Keywords: *HPC, Supercomputing, OpenMP, Finance, Multicore, OpenCL*

1 Introduction

The global financial crisis of 2008 has pointed to the lack of efficacy of financial models and systems to run those models efficiently. Smarter systems can help bridge that gap by providing programming frameworks that are easy to learn, use and deploy. Also these frameworks must support languages that are functionally portable and provide the maximum degree of performance portability. As systems are confronted with the diminishing returns of clock-speed improvements, they must turn to uncovering task and data parallelism in workloads to maintain performance growth. This is needed to efficiently use the increasing multitude of cores that are being provided on modern CPUs today. Unfortunately, parallel programming is hard and significant research in academia and industry is attempting to bridge the gap between programmability, portability and performance [12].

OpenMP [2] has emerged as the de-facto standard for programming SMP (Shared-Memory) machines using user-provided compiler directives. The standard has also been extended recently to support accelerators and thread affinity [2]. OpenCL [1] (Open Compute Language) is vying to become the de-facto standard for programming accelerators. Originally designed for targeting graphics chips and accelerators, it is also emerging as a programming framework for parallelizing workloads on multi-core CPUs. In this paper, we investigate how OpenCL may be used for programming multi-core CPUs as *opposed* to GPUs (Graphics Processing Units). We compare the productivity and performance of OpenCL with OpenMP for multi-core CPUs. We also investigate how SMT (Simultaneous Multithreading) i.e. sharing of a CPU's physical resources by use of multiple hardware contexts can improve performance. This paper investigates the use of SMT for computational finance workloads that tend to be compute bound. In this paper, we use the term SMT "degree" to indicate the number of SMT threads per core.

Computational finance workloads are diverse in their computational characteristics. They tend to be sensitive to square root, exponential, logarithmic and reciprocal functions. They are run in single precision (SP) or double precision (DP) modes. A common practice is to replicate sequential implementations of these workloads across a large CPU cluster. In this paper we investigate how parallelism may benefit computational finance workloads. We built a VaR (Value-at-Risk) workload based on [3] and interaction with users at various conferences [13, 14, 18]. The VaR workload is rich with elementary math and transcendental math functions. It also displays significant amount of task and data parallelism.

We begin by exploring the characteristics of computational finance system stacks for computational finance in Section 2. Section 3 describes the parallelization frameworks that we evaluate in this paper. Section 4 presents the system architectures on which the aforementioned parallelization frameworks will be run. Section 5 describes the workload and avenues for parallelization in this workload. Section 6 describes implementation issues and Section 7 provides performance results with detailed analysis. Section 8 details the impact of performance evaluation. Section 9 discusses related work. Section 10 concludes the paper with future work.

2 Computational Finance Systems

We participated in several conferences over the past few years and learnt that customers in the HPC finance community use Linux for their computational needs [13, 14, 18]. The use of x86 CPUs (which support 2-way SMT) is highly widespread in this community. We decided not to take this practice for granted and investigate higher degrees of SMT cardinality or "degree". Users usually run models on R,

Matlab [15] or Mathematica [16]. They then convert this to C/C++ when the logic and mathematics become stable. The conversion to C/C++ allows programmers to check for numerical stability and convergence. There is a tendency to run these single threaded C/C++ programs by brute-force replication across large number of cores. The C/C++ code tends to be sensitive to sqrt(), log(), inv() and exp() performance.

We decided to explore parallelization so that users could get speedups on their codes, execute models in a reduced amount of time and be able to trigger changes to their business process because of this newly attained efficiency. We decided to investigate OpenCL [1] and OpenMP [2]. OpenMP has been widespread in the HPC community where automatic thread parallelization is sought by using compiler directives. OpenCL has become popular in the HPC community as the medium for programming GPUs. In this paper, we investigate how OpenCL can be used for parallelizing codes across *CPUs* especially when they are enabled for SMT.

Based on our user investigations and findings, we came up with the following requirements for systems that are suited for computational finance –

1) Run Linux on HPC building blocks

2) Support throughput computing using large number of threads or cores

3) CPU should also be capable of high single threaded performance

4) CPU, compilers and runtimes are optimized for sqrt(), log(), inv() and exp() performance. Additionally, they must support elementary math functions (e.g. IBM MASS [4] and linear algebra libraries (e.g. ESSL [4] and ATLAS)) with high performance

5) Provide parallelization that is user-directed and performance portable across a given set of platforms by investigating OpenCL and OpenMP

6) Provide the capability to port codes across new generations of hardware

The workload-optimized stack structure for computational finance is shown in Figure 1.

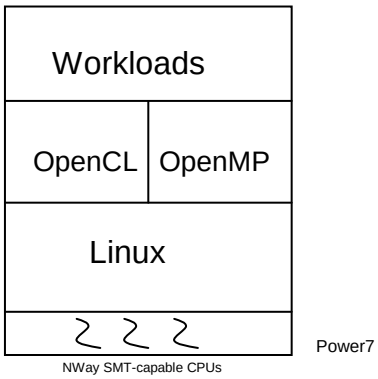

Figure 1. Initial Selection of Stack Components

3 Parallelization Frameworks

OpenCL (Open Compute Language) [1] is an open standard and is backed by a consortium of companies that has developed a framework for standardizing a high-performance computational language which is portable across a variety of platforms. Compute-intensive codes are packaged into a so-called OpenCL "kernel" that can be task- and data-parallelized. The OpenCL model is inspired from GPU programming (CUDA/OpenGL) and draws from the strengths of other parallel programming frameworks. The most fundamental action of an OpenCL kernel is to a "work item" and "work-items" are grouped into "workgroups". "Workgroups" are scheduled to hardware resources e.g. cores. The IBM OpenCL compiler is available on IBM developerWorks for POWER and x86 and this paper uses v0.3 [7]. The host CPU schedules kernels on the attached accelerator or CPU cores and must also ship data required by the kernels.

OpenMP [2] is a standard for parallel programming for shared memory machines and is backed by a consortium of companies. The user supplies OpenMP compiler directives to parallelize loops across resources. Operations like reduction are directly supported in the pragma. The standard is now being extended to support accelerators, thread affinity and embedded systems. IBM supports OpenMP pragmas in addition to its own pragmas for user-directed parallelization. We use OpenMP shipped with 2.6 Linux kernels for IBM Power7 for this paper. The OpenMP compiler generates pThread calls and pThreads are scheduled by the Operating System. We do not use any of the new OpenMP extensions for accelerators or thread affinity.

4 System Architecture

The system architecture used in this paper is a compute blade. This is essentially a multi-core system which is capable of large N-way SMT threads. We use the Power7 CPU [8] which has 8 cores and is capable of 4-way SMT for a total of 32 SMT threads. The CPU runs Linux (RHEL 6.0 with a 2.6.32 kernel). Both OpenCL and OpenMP may be parallelized across multiple CPUs. SMT threads are exposed

by the Operating System as additional "logical" CPUs. OpenCL kernels and OpenMP threads are scheduled across multiple SMT threads. We use GCC 4.4 and IBM XLC 11.01 for C/C++ code compilation. IBM XLCL v0.3 is used for OpenCL kernel compilation. We use the MASS [4] library (Mathematical Acceleration Subsystem) for optimized elementary and transcendental math functions with our C/C++ code. This is used as a high-performance alternative to the standard math library. In the next section, we describe the computational finance workload used to evaluate our system architecture.

5 Risk Analysis Workload

This section describes the risk analysis workload and its computational composition. We also highlight workload components where parallelism may be uncovered. The risk analysis workload uses the "value-at-risk" financial framework which will be described next. The workload has been built with user input and is based on the mathematics described in [3].

5.1 VaR – Value-at-Risk

Value-at-Risk is the potential loss in value of a risky asset over a defined period of time for a given confidence interval. VaR is widely used in commercial and investment banks to capture the potential loss in a traded portfolio from adverse market movements for a given period, e.g. If the VaR of an asset is $ 100 million at one-week, 95% confidence interval, it means that there is a 5% chance that the value of the asset will drop more than $100 million over any given period of one week. There are several popular computational methods used to calculate VaR – (i) historical simulation, (ii) variance-covariance, (iii) Monte-carlo and (iv) Delta-gamma. This paper uses the Monte-Carlo method and will be described next.

5.2 Monte-Carlo VaR

A Monte-Carlo method is used to approximate the probability of outcomes by performing so-called "random walks". "Random walks" simulate multiple trials of a random variable. In the case of VaR, portfolio returns over a given period of days is computed and then sorted. The largest loss corresponding to the VaR period is reported as shown in Figure 2. There are number of ways of computing portfolio returns over a given period but we use the Black-Scholes model to price financial instruments. In this model, the price of a financial instrument consists of log(), inv(), sqrt() and exp() functions in its closed-form representation.

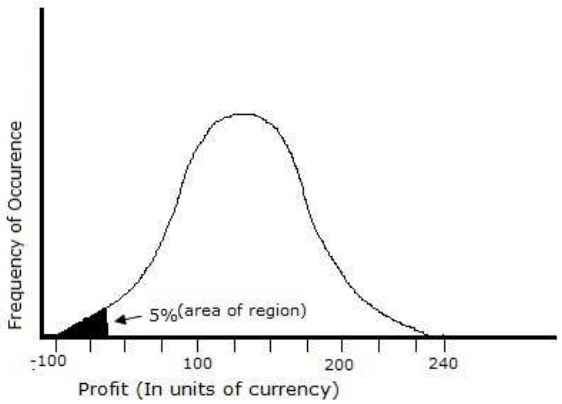

Figure 2. Value-At-Risk Distribution

5.3 Monte-Carlo VaR Parallelization

The Monte-Carlo VaR implementation pseudo code is below. As a first step, we provide the co-variance matrix which captures the portfolio details, the number of portfolios, number of Monte-Carlo simulations and the confidence interval. A number of parallelization strategies are possible. We could equally divide the portfolios across multiple cores and each of the portfolio groups could be computed concurrently. Alternatively, a single portfolio could be computed at a time and the simulations could be parallelized across multiple cores. We use the latter strategy. The values needed for random walks are computed using random number generation which can be computationally intensive. Each simulation prices a financial instrument using a closed form representation composed of log(), inv(), sqrt() and exp() functions. These returns (in currency units) are sorted and the partial distribution is sent to the root CPU for aggregation and merge-sorting. After all CPUs complete their computation, the root CPU forms a distribution similar to Figure 4. The VaR calculation picks the loss corresponding to the confidence interval from the sorted distribution, which is aggregated from all participating CPUs.

1. Supply covariance matrix, number of portfolios, number of simulations and confidence interval
2. For portfolio = 1…N
3. **do**
    1. Compute random numbers 1..M
    2. For simulations = 1 … M
    3. **do**

        Compute portfolio return by pricing financial instruments in the Black-Scholes Model

    End Loop

4. Sort and form partial distribution of returns vs frequency

5. Send partial distribution to root CPU

6. Root CPU Merge-Sorts partial distributions

7. Root CPU forms aggregate distribution

8. Root CPU extracts VaR value from aggregate distribution

End Loop

6 Implementation Issues

This section describes the implementation issues for OpenCL and OpenMP code. We used the Armadillo [17] library for implementing the VaR code, since it gave us function wrappers for BLAS and LAPACK libraries. In addition, it gave us matrix template classes. We used the cholesky decomposition and convolution operations as the principal linear algebra functions. Our elementary math functions included the erlang function, log(), inv(), sqrt(), rnd() and exp().

6.1 OpenCL

We encoded the entire Monte-Carlo simulation into an OpenCL kernel. We support both single precision and double precision calculations inside the kernel as a compile-time option. We did not pre-calculate random numbers but supply them to the kernel during execution. The IBM OpenCL runtime v0.3 reported upto 32 workgroups for the Power7 blade with 32 SMT threads across 8 cores with 4-way SMT. It also reported a global workgroup size of 1024. We inferred that a single workgroup is being scheduled to each SMT thread and used a local workgroup size of 32. The OpenCL program evaluates a single portfolio at a time and the simulations for a particular portfolio are mapped across the local workgroup size of 32 work-items in a given workgroup.

6.2 OpenMP

For the OpenMP implementation, we compute a single portfolio at a time and parallelize the simulations across multiple SMT threads of the 8-way Power7 system. A single OpenMP pragma was needed to parallelize the simulations.

7 Performance Evaluation

We now describe the performance evaluation of the risk analysis workload for OpenCL and OpenMP implementations. We measure speedup and execution time against the number of SMT threads available on the system. We have an 8-way Power7 blade which is capable of supporting 32 SMT threads. The Risk Analysis Workload (VaR) is evaluated with a portfolio size of 10,000 and with 112,640 simlulations. We measured each data-point three times and used the average in the plots below to eliminate any OS-jitter effects. We configured the Linux OS scheduler to schedule user-level threads on SMT hardware contexts in an even fashion so load is evenly balanced, e.g 11 threads across 8 cores means that 1 SMT thread is scheduled on each core, while 3 cores have an additional SMT thread running. Each user-level software thread is mapped to a single SMT thread.

7.1 Single Precision Floating Point (SP)

Figure 3 shows the execution-time performance of OpenMP (with standard math library, also called OpenMP-std-math), OpenCL and OpenMP-with-MASS implementations using the GCC compiler with –O3. MASS is the IBM Mathematical Acceleration Subsystem [4] library which is optimized for transcendental and elementary math operations. The OpenCL compiler (v0.3, XLCL) was used to compile the OpenCL kernel and then GCC was used to link and bind all object modules. The performance of the code with the standard math library performs worse than the OpenCL and OpenMP-with-MASS implementations.

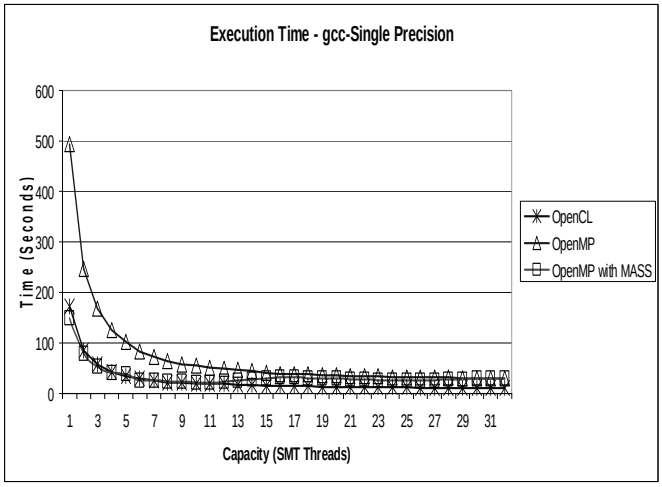

Figure 3. Execution Time vs SMT Threads (GCC, SP)

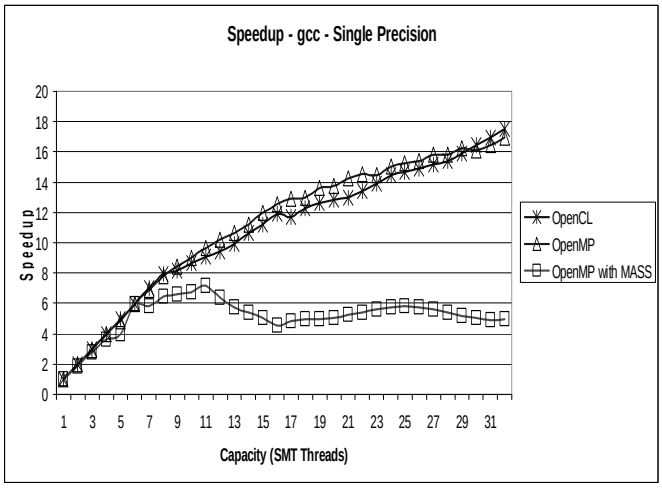

Figure 4. Speedup vs SMT Threads (GCC, SP)

The OpenMP-with-MASS implementation tracks the OpenCL compiler performance until 11 SMT threads and then starts diverging. The OS scheduler schedules 8 SMT threads across 8 cores and 3 threads across 3 cores, so 3 cores have 2 SMT

threads running. We ran the "perf stat" profiler and found that IPC decreases and cache misses of the GCC-generated code increases beyond 11 SMT threads. We attribute this to resource conflicts due to OpenMP thread data sharing using the highly-optimized MASS library. Enhanced MASS optimizations leads to reduced number of pipeline stalls and cache misses. Thus, they do not benefit from SMT. Note that OpenMP with the standard math library does not display this behavior as it does not provide efficient optimizations. OpenCL code compilation follows a two-step process and the IBM Power OpenCL compiler is able to do a better job of compacting thread and data state to avoid resource conflicts. 4-way SMT on Power7 for this workload is able to provide a 50% reduction in elapsed time at 32 SMT threads (four per core) over 8 SMT threads (1 per core) for the OpenCL and OpenMP-std-math codes.

The speedup curves in Figure 4 show OpenMP-with-std-math and OpenCL codes providing near-linear speedups but the OpenMP-with-MASS codes flattening after 11 SMT threads. This is easily attributed to enhanced optimization with MASS leading to higher floating point pipeline utilization that does not benefit by SMT. Additionally, the single thread performance of OpenMP-with-MASS is higher than OpenMP-with-std-math and slightly higher than OpenCL.

We used the IBM XLC compiler to compile our codes and found that the OpenMP-with-MASS implementation does much better than the corresponding GCC-generated code of Figure 3. In fact, the OpenMP-with-MASS implementation equals or betters the OpenCL implementation using the IBM XLC compiler. The execution times are shown in Figure 5 and the speedups are shown in Figure 6. The speedup curves of OpenCL are close to linear than OpenMP-with-MASS because the OpenMP implementation does better at execution time of 1 SMT thread but not at higher thread counts, as the enhanced optimization leads to optimal resource utilization of the floating-point pipeline and cache.

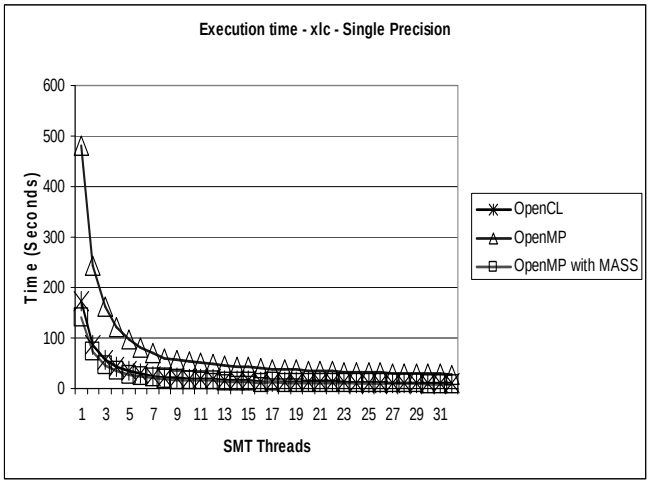

Figure 5. Execution Time vs SMT Threads (XLC, SP)

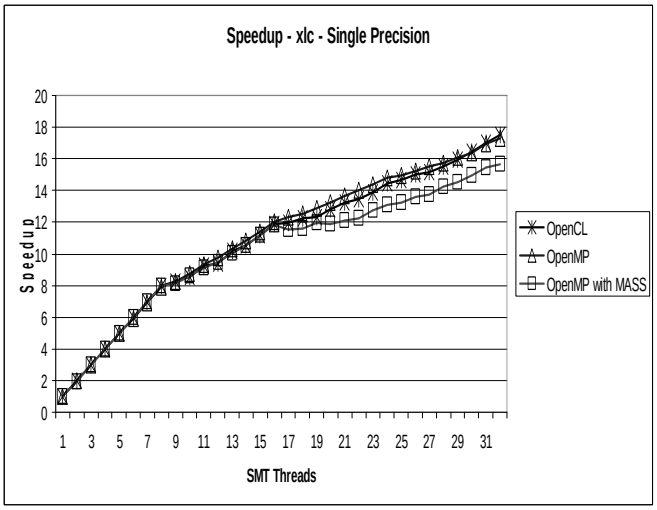

Figure 6. Speedup vs SMT Threads (XLC, SP)

7.2 Double Precision (DP)

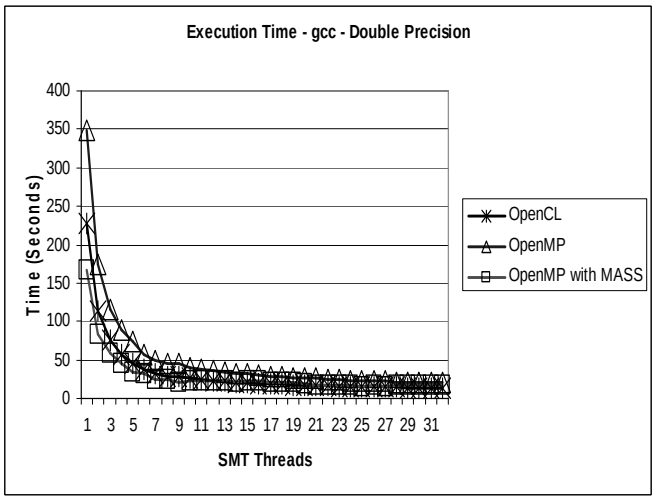

Figure 7. Execution Time vs SMT Threads (GCC, DP)

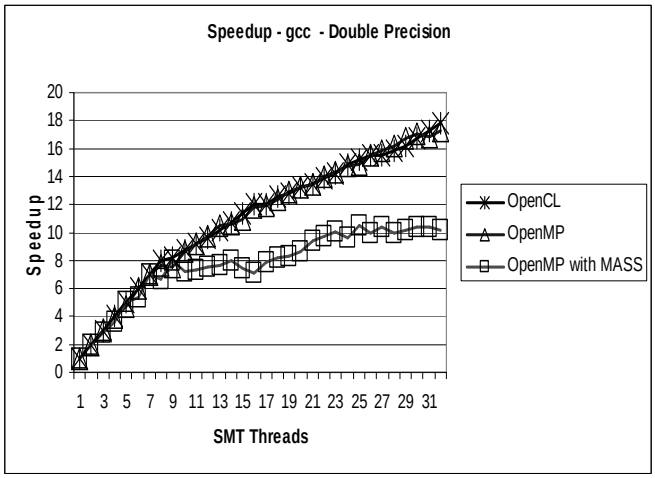

Figure 8. Speedup vs SMT Threads (GCC, DP)

Most codes in computational finance are run with double precision floating point and we discuss the results with GCC- and XLC-generated code. Figure 7 and Figure 8 show GCC-generated code performance for OpenCL, OpenMP-std-math and OpenMP-with-MASS. The OpenMP-with-MASS code shows similar behavior as seen for the single precision case where the execution time elongates slightly at 12 SMT threads due to resource conflicts. In Figure 8, OpenCL and OpenMP-with-std-math show close to linear performance but OpenMP-with-MASS flattens after 8-10 SMT threads. This is attributed to the higher performance of OpenMP-with-MASS at 1 SMT thread. Additionally, the use of MASS leads to efficient code that utilizes floating point and cache resources efficiently, leading to poor speedups when physical cores are shared with SMT.

Figures 9 and 10 show the performance of XLC-generated code. Both OpenMP-std-math and OpenMP-with-MASS performance exceeds the performance of the OpenCL code using the IBM XL compiler. This is attributed to better scheduling of code and resources in double precision mode by the XLC compiler leading to better utilization of the floating-point pipeline. If Figure 9 is compared with Figure 5, where Figure 5 is XLC-generated code in single precision mode, OpenMP-with-std-math in double precision does better than corresponding single precision code. In comparison, XLC-generated OpenCL code in double precision does worse than single precision code. We attribute the better performance of OpenMP-with-std-math code in double precision to better utilization of the floating point pipeline and instruction scheduling by the XLC compiler. In the case of OpenCL and OpenMP-with-MASS, both see reduced performance for DP code as the CPU can usually sustain more SP operation throughput than DP operation throughput. For DP code, OpenMP-with-MASS does much better than OpenCL as the XL compiler for C/C++ is able to do a better job of instruction scheduling and floating-point utilization than the XL compiler for OpenCL.

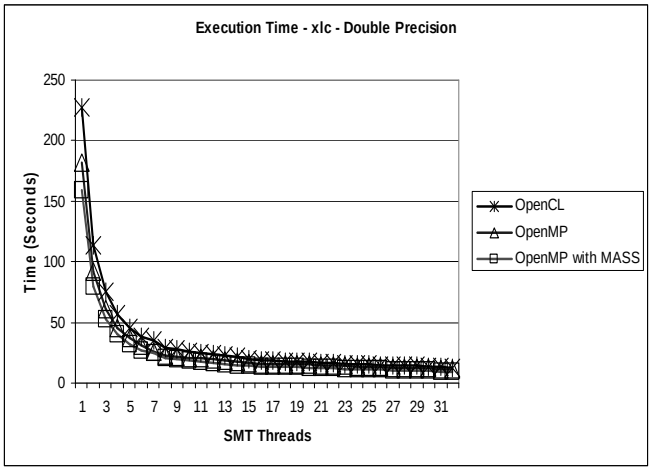

Figure 9. Execution Time vs SMT Threads (XLC, DP)

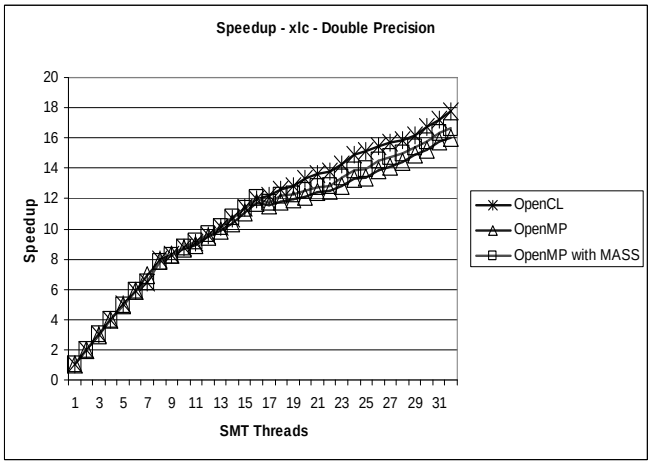

Figure 10. Speedup vs SMT Threads (XLC, DP)

The speedup curves of Figure 10 show OpenCL code showing close to linear performance while the OpenMP code flattening around the 16 SMT thread region. This region involves two SMT threads sharing a physical core. The sublinear performance in this region is easily attributed to resource and cache conflicts.

7.3 Impact of SMT

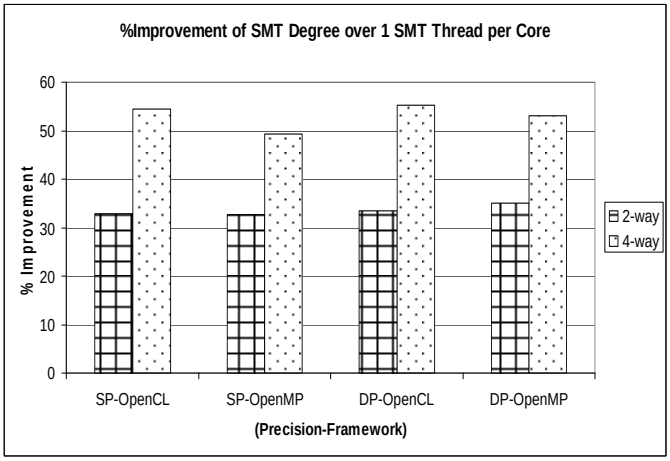

Figure 11. Impact of SMT Degree (XLC)

Figure 11 shows the impact of SMT degree for the VaR workload using the XLC compiler. 2-way SMT gives an average of 33.4% improvement across OpenCL and OpenMP while 4-way SMT gives an improvement of 53% across OpenCL and OpenMP. For the 2-way improvement calculation, 16 SMT threads on the 8-core system (2 threads/core) are compared to 8 SMT threads (1 thread/core). For the 4-way improvement calculation, 32 SMT threads on the 8-core system (4 threads per core) are compared to 8 SMT threads (1 thread/core). All comparisons are based on execution times of the VaR (Value-at-Risk) workload for different SMT thread counts using the XLC compiler.

8 Post-Evaluation Analysis

For the homogeneous multi-core architecture, the learning curve for OpenCL can be steep. For OpenMP, we simply had to use a single-line OpenMP pragma to parallelize the entire program, although, we could only match OpenCL performance using the right choice of IBM compiler toolchains and libraries. With OpenCL, we can run the OpenCL kernel unchanged from a CPU to an accelerator like a GPU. OpenCL allows portable interface with accelerators but OpenMP provides ease of use with high-level OpenMP directives. We were able to match OpenMP and OpenCL performance using the right set of compiler tool chains and libraries. The stack of Figure 1 should be updated to reflect the use of IBM XL C/C++ compilers and MASS libraries. For programming homogeneous multicores, OpenMP with XLC and MASS seems to emerge as a formidable framework. It provides better productivity and can match OpenCL performance.

OpenCL provides a portable way to program multicores and accelerators. For example, one may prototype programs on a multicore and then launch these on CPU-GPU clusters. OpenCL is a low-level mechanism to exact parallelism and performance. OpenMP relies on the compiler and so the right compiler tool chain and libraries is needed to match OpenCL performance. OpenACC [5] is a newly emerging standard that stipulates compiler directives for use with accelerators and provides best of both worlds – OpenMP and OpenCL. The results of this paper reveal that 4-way SMT provided by the POWER architecture can lead to additional benefits by reducing execution time. In other words, if application code optimizations and compiler action are sub-optimal, SMT degree can be used to get better processor pipeline utilization and enhanced execution time. SMT is beneficial for the VaR workload because SMT threads are able to share instructions and data.

9 Related Work

[9, 10] focus on random number generation and their implementation complexity on the IBM Cell BE accelerator for value-at-risk calculations. They do not focus on understanding scaling issues and implications of programming models and frameworks. Similarly, [11] focuses on market risk calculations on GPUs (Graphics Processing Units). They also do not focus on scaling or use of parallelization frameworks. This paper uses CPUs for market risk calculations and investigates the use of SMT to increase processor utilization, while analyzing scaling.

10 Conclusion and Future Work

We evaluate OpenCL and OpenMP for both single precision and double precision VaR codes. We find that although OpenCL has a steeper learning curve, it provides good performance without use of additional libraries. OpenMP is based on user-directed parallelism but requires the right compiler toolchain (XLC) and library combination (MASS) to match OpenCL performance. The use of OpenCL and OpenMP parallelization frameworks can lead to better utilization of POWER cores using SMT. We find that SMT degree does help execution. We are currently evaluating a library that will allow OpenCL kernels to be dispatched onto attached blades from a large SMP system. This will allow computation to be scheduled close to data than moving data to the computation. We are also addressing how OpenCL kernels can be accelerated by explicit use of data parallelism. OpenCL allows portability of kernels across CPUs and accelerators. The new OpenMP directives [2] are also attempting to make OpenMP easy to program accelerators. OpenACC [5] is also headed in the direction of using compiler directives to program accelerators. As the race to find “Utopia” for parallelization frameworks continues, the systems community must constantly evaluate workloads against new system architectures. This will help determine which workload classes benefit from new and emerging parallelization frameworks.

11 References

[1] http://www.khronos.org/opencl/

[2] http://openmp.org/wp/

[3] Benninga, S and Wiener, Z. Value-at-Risk (VaR). Mathematica in Education and Research. Vol. 7, No. 4, 1998.

[4] http://www-01.ibm.com/software/awdtools/mass/linux/

[5] http://www.openacc-standard.org/

[6] Black, Fischer; Myron Scholes (1973). "The Pricing of Options and Corporate Liabilities". Journal of Political Economy 81 (3): 637–654. doi:10.1086/260062.

[7] https://www.ibm.com/developerworks/community/groups/service/html/communityview?communityUuid=80367538-d04a-47cb-9463-428643140bf1

[8] http://www-03.ibm.com/systems/power/advantages/power.html

[9] V. Agarwal, L.-K. Liu, and D.A. Bader, ``Financial Modeling on the Cell Broadband Engine," *22nd IEEE International Parallel and Distributed Processing Symposium* (IPDPS), Miami, FL, April 14-18, 2008

[10] D.A. Bader, `` On the Design of Fast Pseudo-Random Number Generators for the Cell Broadband Engine and an Application to Risk Analysis," *37th* International Conference on Parallel Processing, Portland, OR, Sept 8-12, 2008

[11] Matthew Dixon, Jike Chong, and Kurt Keutzer. 2009. Acceleration of market value-at-risk estimation. In *Proceedings of the 2nd Workshop on High Performance Computational Finance* (WHPCF '09).

[12] Krste Asanovic, “The Landscape of Parallel Computing Research: A View from Berkeley”, Technical Report No. UCB/EECS-2006-183, University of California at Berkeley, Berkeley, California.

[13] http://www.flaggmgmt.com/hpc/

[14] http://www.flaggmgmt.com/linux/

[15] http://www.mathworks.com/products/matlab/

[16] http://www.wolfram.com/mathematica/

[17] http://arma.sourceforge.net/ (C++ Linear Algebra Library)

[18] www.sifma.org/ (Securities and Financial Markets Association)