Assessing the Performance of Computational Engineering Codes

by

Omkar Deshmukh

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN – MADISON

2015

Approved:

Associate Professor Dan Negrut

Department of Mechanical Engineering, Department of Electrical and Computer Engineering (affiliated)

University of Wisconsin – Madison

Abstract

The goal of this thesis is threefold. First, it attempts to gauge the performance of different computational engineering libraries on different platforms. Given a numerical computing library, we compare the performance between all supported backends running same benchmark. The benchmarks run were the reduction, sorting, prefix scanning and SAXPY operation of vectors ranging from 10M to 25M elements in size. The second part consists of the use of profiling tools to understand code performance and the underlying hardware – software interplay. Finally, we discuss a performance tracking infrastructure that is instrumental in carrying out the process of benchmarking and analyzing the results in a reproducible manner. We describe this infrastructure in terms of its source control management, automation with makefiles and python scripts, and the use of a relational database management system that combine to enable the user to find out expeditiously performance metrics of various libraries on multiple architectures.

Acknowledgments

I am extremely grateful to my advisor, Associate Professor Dan Negrut, for his guidance and support in the past years. His abilities and work ethic continue to inspire me. I would also like to thank the other members of the committee, Professor Eftychios Sifakis and Professor Krishnan Suresh for their expertise. I am also grateful for the friendship and assistance of my colleagues in the Simulation-Based Engineering Laboratory without whose support, none of this would have been possible.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Background

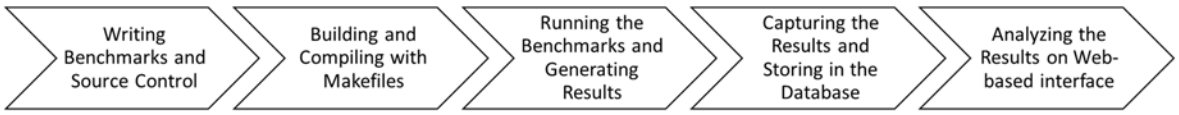
Advanced computing is at the core of computational engineering. The goal of this study is to understand the interplay between hardware and software used for advanced computing in Science and Engineering and use the findings to make informed decisions on how to better run simulation-based workloads. This work involves research on three fronts – software profiling, benchmarking state of the art hardware platforms, and creating infrastructure for performance benchmarking legacy purposes.

Software profiling is a form of program analysis that measures, for example, the space or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization. We worked with profilers such as Intel Vtune and Linux Perf, which operate on the principal of statistical sampling. The data generated by these tools help us gain insights into performance hotspots, threading, locks & waits, memory bandwidth utilization among many other parameters. For example, to understand the CPU behavior, we measure last level cache misses, branches taken and branch mispredictions, CPU cycles consumed, floating point unit utilization, etc. The performance of memory subsystems can be gauge by assessing the number of memory accesses and page fault rates. These insights are helpful in identifying potential performance tuning opportunities.

The process of benchmarking revolves around the idea of running a set of programs to assess the relative performance of advanced computing; i.e., parallel computing, systems we work on. We associate it with assessing the performance characteristics of a highly parallel processor and the software stack running on top of it. Thus the analysis involves running a micro-benchmarking suite on different sets of platforms with libraries optimized for linear algebra and vector operations and gathering the results for further analysis. As an example, one of the benchmarks runs reduction operation on vectors for floating point number of sizes up to 25 million entries.

As noted by Zhang [1], based on their size and complexity, benchmarks can be classified as microbenchmarks and macrobenchmarks. Microbenchmarks are helpful in measuring the performance of an underlying platform, whether hardware or software in terms primitive operations supported. They often consist of short sequences of code (kernels) that solve small and well-defined problems. Microbenchmarks are helpful in a detailed understanding of information about the fundamental computing costs of a system and are useful in comparing low-level operations of different systems.

The tools and techniques that allow us perform detailed analysis of performance profiles – such as comparing the performance of an accelerator across different numerical computing libraries, are built-on top of software infrastructure that makes heavy use of automation. The process of getting useful results out of large amounts of performance metrics involves a number of steps. The end-to-end automated workflow of this entire process can be described in the following stages:



For the purpose of source code management, we use Github which offers distributed revision control among all the contributors. Along with the source codes for benchmarks, it also keeps track of Makefiles used for compilation workflows. These Makefiles are used to generate platform specific benchmark executables built with specific libraries. These are run using automated scripts, which generate the results based on types of inputs provides. These results are stored in a relational database management system, which keeps track of all the data generated along with any associated metadata. The schema for this database is given below:

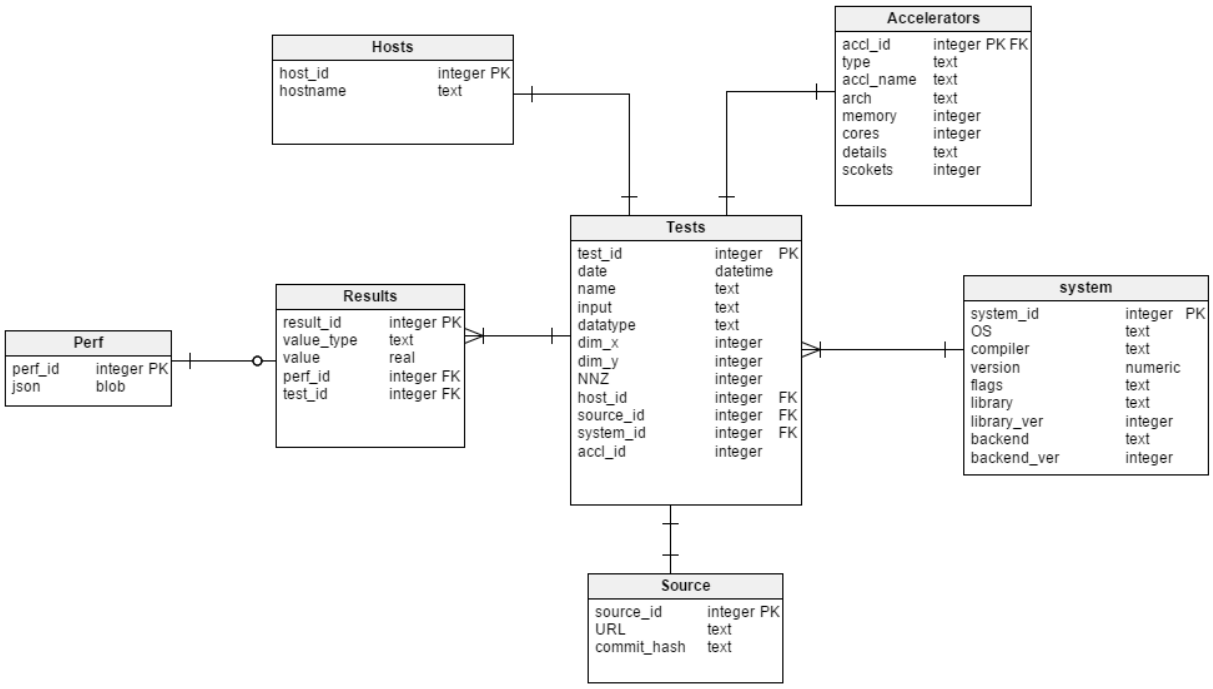


Figure 1. Database Schema

With this setup, we are able to capture important characteristics such as the timing results for specific inputs, the configuration of the system under state, the code that was used to run these tests etc. A typical scenario for running sparse matrix-vector multiplication (SpMV) operations involves storing in the *Tests* table the characteristics of input matrices and the timing metrics in *Results* table. The workflow also captures different parameters such as *Operating System* and *Compiler* used, details about the accelerators used, the particular version of the code from Github repository and stores them appropriate tables. This data can later be used for detailed analysis of hardware and software systems under test.

1.2 Organization

This thesis is organized as follows.

- In Chapter 2, we talk about some of the numerical computing libraries which are widely used in computational engineering. These include commercial offerings such as Intel's Math Kernel Library (MKL) as well open-source projects such as Thrust and VexCL.
- In Chapter 3, we describe benchmarks which were generated with the help of these libraries. These benchmarks stress most commonly used routines such as inclusive scan of vector of floating point numbers and SpMV.
- In Chapter 4, we discuss the hardware setup and testing methodology employed.

- In Chapter 5, we discuss the results of the micro-benchmarks used to evaluate the platforms and how the hardware architecture determines the results and performance.
- In Chapter 6, we talk about software profiling tools such as Intel VTune and Linux Perf in an attempt to explain some of the software bottlenecks and performance profiles we see with the benchmarks.
- In Chapter 7, we discuss the software infrastructure that has been built to automate the process of benchmarking and analyzing results. This infrastructure is also shown to help in building research practices which help carry out the entire process in structured manner. With constantly evolving hardware and software packages that are updated routinely, it enables us to reproduce and verify the results as and when needed. In particular, we talk about importance of source code managements and tools such as Github, Python for automated scripts and Relational Database Management Systems for archiving and analyzing large amounts of data generated during performance profiling of the benchmarks.

2. Numerical Computing Libraries

This section briefly describes the libraries that have been benchmarked as part of a systematic study that sought to understand what type of performance can be counted on when building engineering codes that require numerical algebra and/or utility support for operations such as sorting, prefix scans, etc.

2.1 VexCL

VexCL is a vector expression template library for OpenCL/CUDA [2]. It has been created for ease of development with C++ on Graphics Processing Units (GPU), also called General Purpose GPU (GPGPU) computing. VexCL strives to reduce the amount of boilerplate code needed to develop GPGPU applications. The library provides convenient and intuitive notation for vector arithmetic, reduction, sparse matrix-vector products, etc. It supports running the programs on multiple devices at once and also write-once-run-anywhere types of computations which work across platforms of different architectures. For the purpose of testing, we modified and worked with some of the benchmarks that are supplied as a part of VexCL distribution.

2.2 Thrust

Similar to VexCL, Thrust [3] is a C++ template library for CUDA inspired by the Standard Template Library (STL). It provides a rich collection of data parallel primitives such as scan, sort, and reduce, which can be composed together to implement complex algorithms with concise, readable source code. Describing the computation in terms of these high-level abstractions provides freedom to select the most efficient implementation automatically. As a result, Thrust can be utilized in rapid prototyping of CUDA applications, where programmer productivity matters most, as well as in production, where robustness and absolute performance are crucial.

2.3 MKL

Intel Math Kernel Library (MKL) [4] provides optimized math processing routines that leverage parallel execution to increase application performance and reduce development

time. Intel MKL includes highly vectorized and threaded linear algebra, fast Fourier transforms (FFT), vector math and statistics functions. We worked with version 13.0 which implements standard BLAS and LAPACK interfaces. Therefore, if the application relies on such functionality, one can re-link with Intel MKL to get potentially better performance on Intel and compatible architectures.

2.4 Blaze

Blaze [5] is an open-source, high-performance C++ math library for dense and sparse arithmetic. It offers high performance through the integration of BLAS libraries and manually tuned HPC math kernels. It provides shared memory parallelization with OpenMP backend by default. It also supports parallel execution by C++11 threads and Boost threads.

3. Benchmarked Tests

For the purpose of benchmarking, we identified some of the most commonly used routines in computational engineering. These tasks, which are discussed in detail below, are usually the hotspots in the program being executed – meaning they end up using the majority of system resources such as CPU clock cycles, inter-node bandwidth and main memory. Working with microbenchmarks is different than profiling: Here we attempt to measure the performance in terms of small bits of code as opposed to that of the entire application. Nonetheless, these are very helpful in understanding the potential of the hardware-software combo employed and how close one can get to the maximum theoretical performance offered. Given below are the routines that were at the core of these microbenchmarks.

3.1 Reduction

A reduction algorithm uses a binary operation to reduce an input sequence to a single value. For example, the sum of an array of numbers is obtained by reducing the array using a plus "+" operation. Similarly, the maximum of an array is obtained by reducing with an operator that takes two inputs and returns the maximum. The number of floating operation in SUM based reduction operation is equal to the size of the vector.

3.2 Prefix scan

Parallel prefix-sums, or scan operations, are important building blocks in many parallel algorithms such as stream compaction and radix sort. In an inclusive or prefix-scan each element of the output is the corresponding partial sum of the input range. For example, data[2] = data[0] + data[1] + data[2]. An exclusive scan is similar, but shifted by one place to the right. The number of floating point operations in this case are dependent on type of algorithm implemented, which may vary from one library to another. Hence, we measure the performance in terms of number of entries or keys scanned per second.

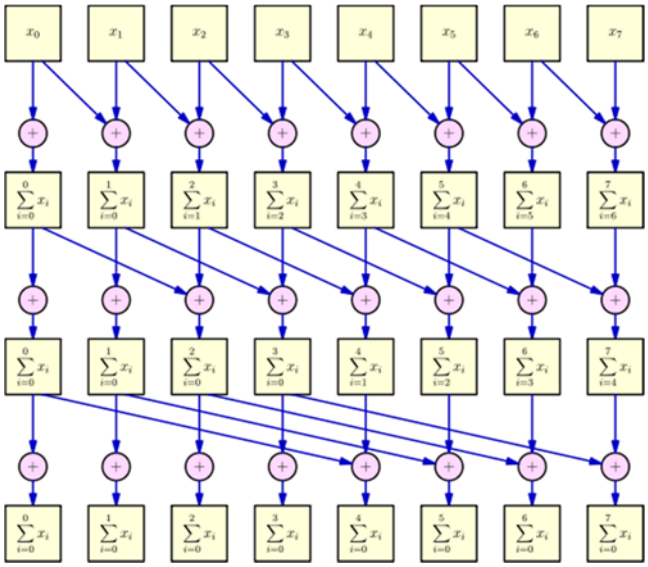


Figure 2. Prefix sum applied to an eight-element array [6]

3.3 Sort

Similar to prefix-scan, scan takes a vector as an input and returns a sorted version of it. For example, Sort sorts the elements in [first, last) into ascending order, meaning that if i and j are any two valid iterators in [first, last) such that i precedes j, then *j is not less than *i. In case of sort as well, the number of floating point operations in this case are dependent on type of algorithm implemented. Hence, here too, we measure the performance in terms of number of entries or keys scanned per second.

3.4 SAXPY

SAXPY adds a scalar multiple of a real vector to another real vector. SAXPY stands for "Single-Precision A·X plus Y". It is a function in the standard Basic Linear Algebra Subroutines (BLAS) library. SAXPY is a combination of scalar multiplication and vector addition: it takes as input two vectors of 32-bit floats X and Y with N elements each, and a scalar value A. It multiplies each element X[i] by A and adds the result to Y[i]. The result is another vector which returned as the result or overwritten in one of the inputs depending upon the implementation.

$$\begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_n \end{bmatrix} \leftarrow \alpha \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_n \end{bmatrix}$$

Figure 3. Single-precision aX+Y

3.5 SPMV

Sparse matrix-vector multiplication is a kernel operation widely used in iterative linear solvers. SpMV of the form y=Ax is a widely used computational kernel existing in many scientific applications. The input matrix A is sparse. The input vector x and the output vector y are dense. The floating point operations for an SpMV operation is equal to twice the non-zero elements in input matrix A.

$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{bmatrix} \cdot \begin{bmatrix} 2 \\ 5 \\ 1 \\ 8 \end{bmatrix} = \begin{bmatrix} 4 \\ 47 \\ 5 \\ 68 \end{bmatrix}$$

Figure 4. Sparse matrix vector multiplication [8]

4. Testing

The testing involves compiling individual benchmarks with required libraries and running them on different accelerators. The process of compilation is simplified with the help of makefiles which help automate the entire process. One such makefile and its contents are given below:

```
CC              := g++
CFLAGS          := -O3 -std=gnu++11
INC_BOOST       := /usr/local/boost/1.53.0/include/
LIB_BOOST       := /usr/local/boost/1.53.0/lib/
INC_CL          := /opt/AMDAPPSDK-2.9-1/include/
LIB_CL          := /opt/AMDAPPSDK-2.9-1/lib/x86_64/

all: ocl

ocl:
     $(CC) $(CFLAGS) -I$(INC_CL) -I$(INC_BOOST) -o ocl.out
vector.cpp -L$(LIB_CL) -lOpenCL -L$(LIB_BOOST) -
lboost_program_options -lboost_system -lboost_thread

bench:
     $(CC) $(CFLAGS) -I$(INC_CL) -I$(INC_BOOST) -o bench.out
benchmark.cpp -L$(LIB_CL) -lOpenCL -L$(LIB_BOOST) -
lboost_program_options -lboost_system -lboost_thread

clean:
     rm -fv *.out
```

With the help of makefiles such as these, we can cut down on time and effort involved compiling and running the benchmarks. It also helps that a single makefile can be used to compile benchmarks on different platforms, with little to no changes. These platforms are briefly discussed next.

4.1 AMD Opteron 6274

Opteron 6274 is AMD's x86 server and workstation processor. The unit under test had four x AMD Opteron 6274 2.2GHz 16 core processors with 128GB DDR3 ECC RAM. It supports quad-channel memory that delivers up to 102.4 GB/s memory bandwidth per CPU socket. The four sockets were interlinked via Mellanox ConnectX-2 MHQH19B-

XTR Infiniband HCA. Each of the 64 cores had a 256-bit FPU. The 16 cores on each socket shared 16MB L3 LLC. The cores ran at the base frequency of 2.2 GHz [9].

4.2 Intel Core i7-5960X

Intel's i7-5960X is part of extreme processor line based on Haswell-E microarchitecture. It is a desktop oriented processor with eight cores. The CPU has 4 memory channels and maximum memory bandwidth of of 68 GB/s. The system under test had 32GB DDR4 RAM and tests were run with HTT (hyper-threading technology) support enabled. The processor supports SSE4.2 and AVX 2.0, has 20MB L3 LLC and runs at a base frequency of 3 GHz [10].

4.3 Intel Xeon Processor E5-2690 v2

Intel's E5-2690 v2 s is an Ivy Bridge-EP processor that is part of Xeon-branded server chips. The system under test had two such CPUs, with Mellanox ConnectX-2 MHQH19B-XTR Infiniband HCA interlink. Each E5-2600 V2 has 10 physical cores, and can handle up to 20 threads via HTT. The number of memory channels available per CPU is 4 which can provide a maximum bandwidth of 59.7 GB/s. The instruction set supports 256-bit floating point arithmetic with AVX. Each physical core has access to 256KB L2 cache and the chip has a total of 25MB L3 LLC. The system used had 64GB DDR3 ECC RAM [11].

4.4 Intel Xeon Phi Coprocessor 5110P

Xeon Phi is Intel's coprocessor based on its Many-Integrated-Core (MIC) architecture. It has 60 cores which run at the frequency of 1.053 GHz and support simultaneous multi-threading with up to 240 threads. The number of memory channels is 16 which combined

offer a maximum bandwidth of 320 GB/s. It has 8 GB GDDR5 memory and PCIe x16 interconnect. It has a 512-bit wide vector engine for floating point arithmetic. The cores have 32 KB each of L1 Instruction and Data cache and 512 KB L2 cache per core [12].

4.5 NVidia Tesla K40c

NVidia's K40c GPU is based on the Kepler architecture and comes equipped with 12GB of GDDR5 memory. It is designed for workstations and servers, and connects to host systems via a PCI Express Gen3 x16 interface. It has 2880 scalar processors [13].

4.6 NVidia Tesla K20Xm

The Tesla K20Xm is a server grade GPU built on the 28 nm process, based on Kepler architecture. Tesla K20Xm is connected to the rest of the system using a PCIe 3.0 x16 interface. It has 6GB of GDDR5 memory onboard. It has 2688 scalar processors [14].

4.7 NVidia GeForce GTX 770

The GTX 770 is a desktop grade GPU built on the Kepler architecture. It has 4GB of GDDR5 global memory and 1536 scalar processors. The bus interface to the host is PCI Express 3.0 [15].

4.8 AMD A10-7850K

The A10 is an integrated system with 12 on-chip compute cores - four CPU cores and eight GPU stream processors which can run a total of 512 GPU threads simultaneously. The CPU and GPU cores make use of Heterogeneous System Architecture (HSA) allowing them to share 16GB of DDR3 between them. The GPU is based on Radeon R7 graphics GCN architecture [16].

5. Results

5.1 Reduction

The number of floating operation in SUM based reduction operation is equal to the size of the vector. As compared to other benchmarks, reduction test requires only one memory access per element of the vector. The number of floating operations in SUM based reduction operation is equal to the size of the vector. The organization of libraries-backends-platforms below shows libraries and backends with which benchmarks were compiling and the hardware platform they were run on:

Library	Thrust		VexCL	
Backend	OpenMP	CUDA	OpenCL	CUDA
AMD A10-7850K			☑	
GTX 770		☑	☑	☑
Tesla K20Xm		☑	☑	☑
Tesla K40c		☑		☑
Xeon Phi	☑		☑	
Xeon E5-2690v2	☑		☑	
i7-5960X	☑		☑	
Opteron	☑		☑	

Table 1. Organization of the Reduction Benchmark

CPU results

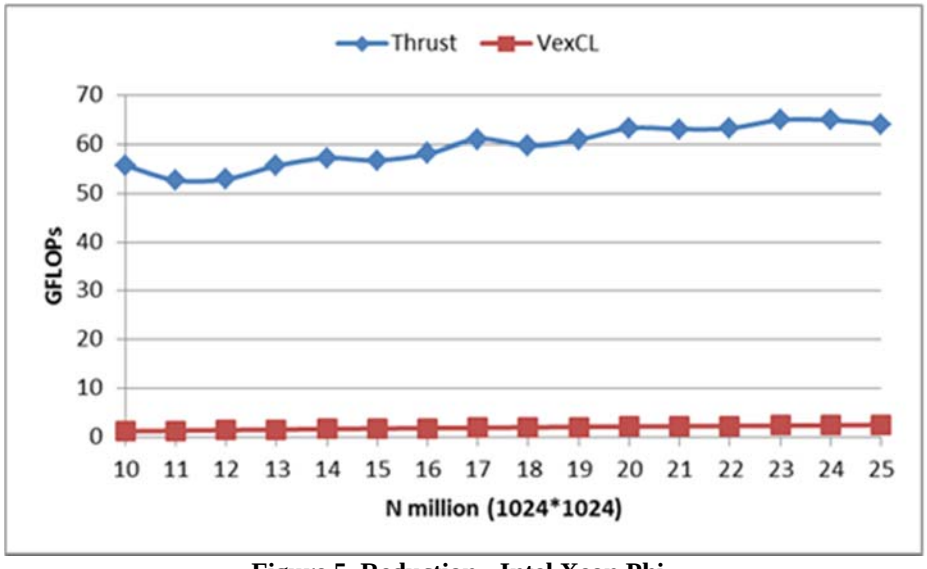


Figure 5. Reduction - Intel Xeon Phi

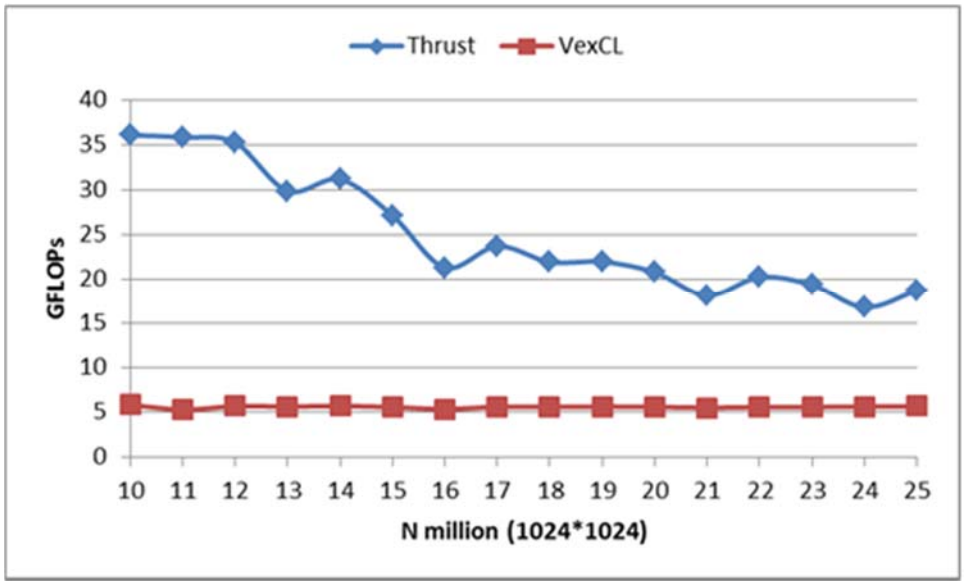


Figure 6. Reduction - Intel Xeon E5-2690v2

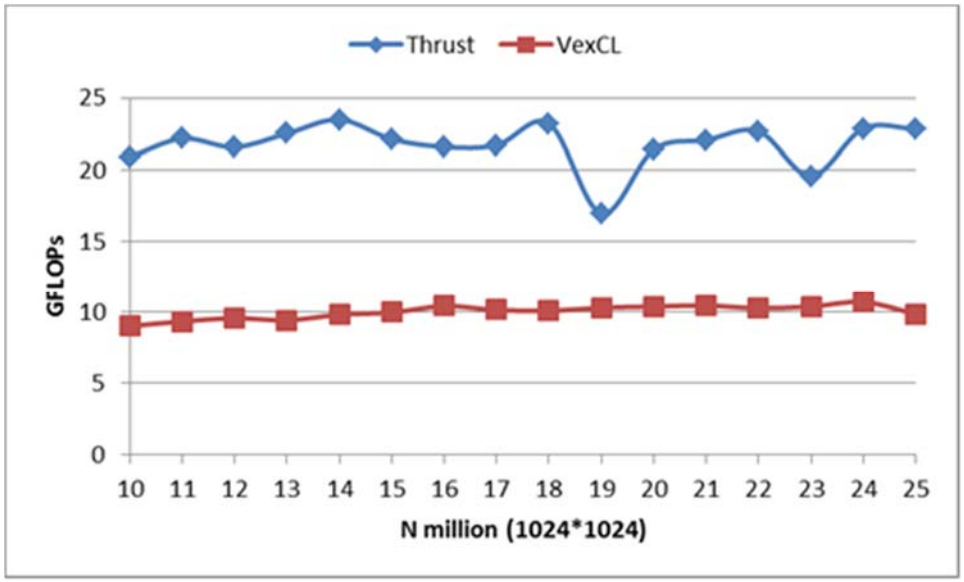


Figure 7. Reduction - Intel i7-5960X

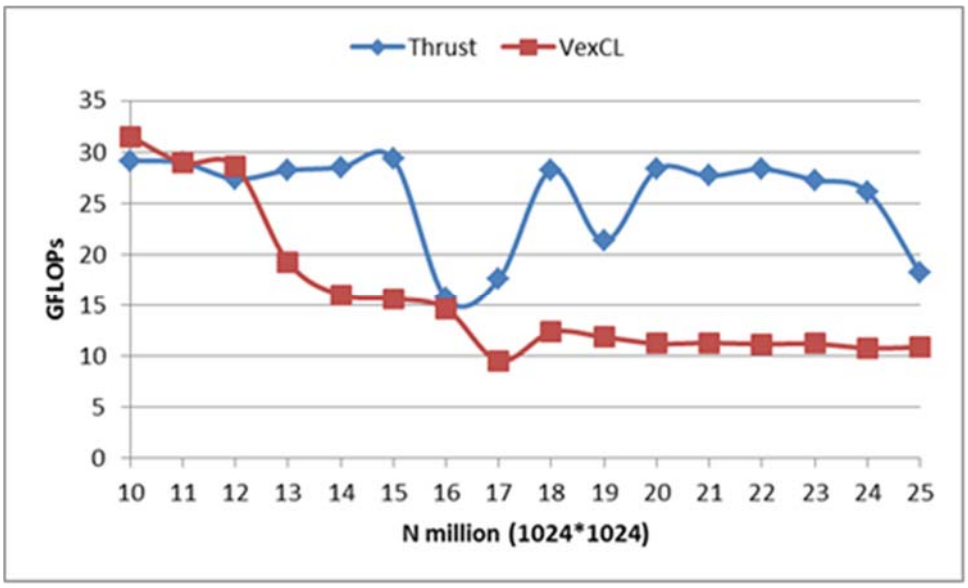


Figure 8. Reduction - AMD Opteron 6274

GPU results

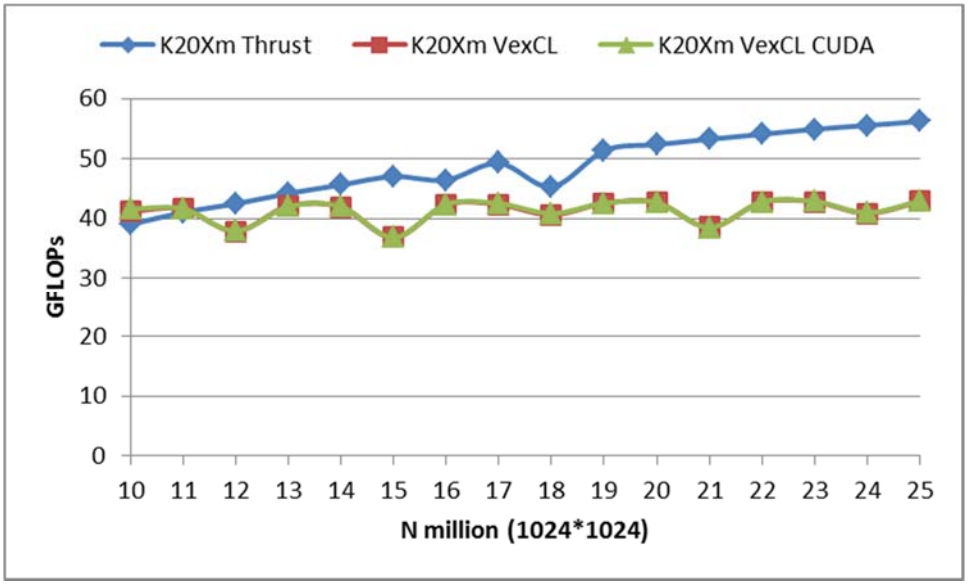


Figure 9. Reduction - NVidia Tesla K20Xm

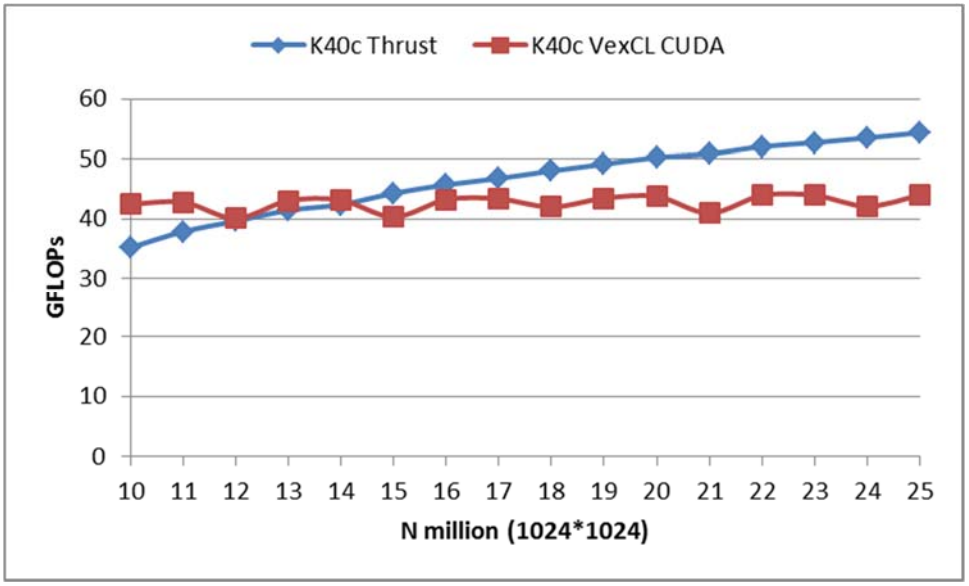


Figure 10. Reduction - NVidia Tesla K40c

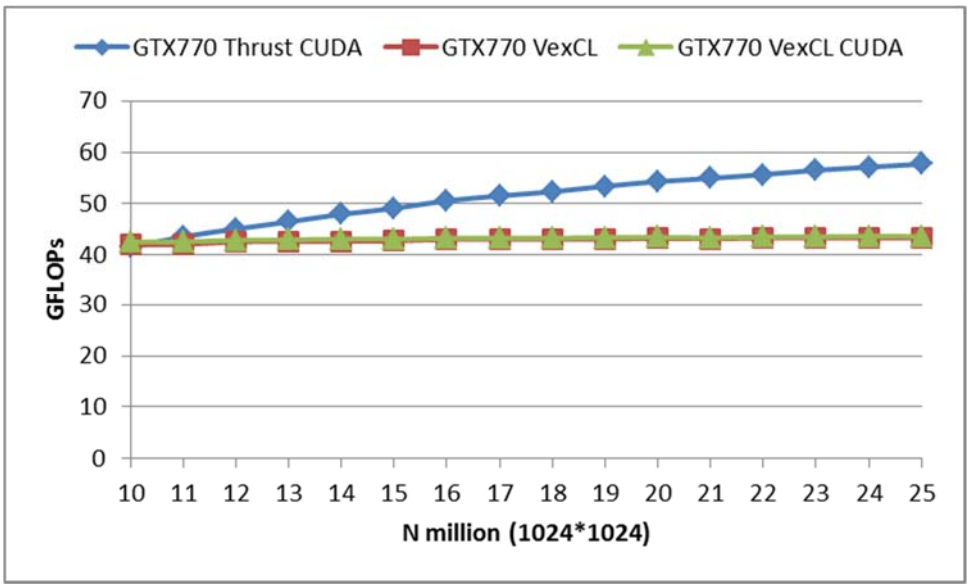


Figure 11. Reduction - NVidia GTX 770

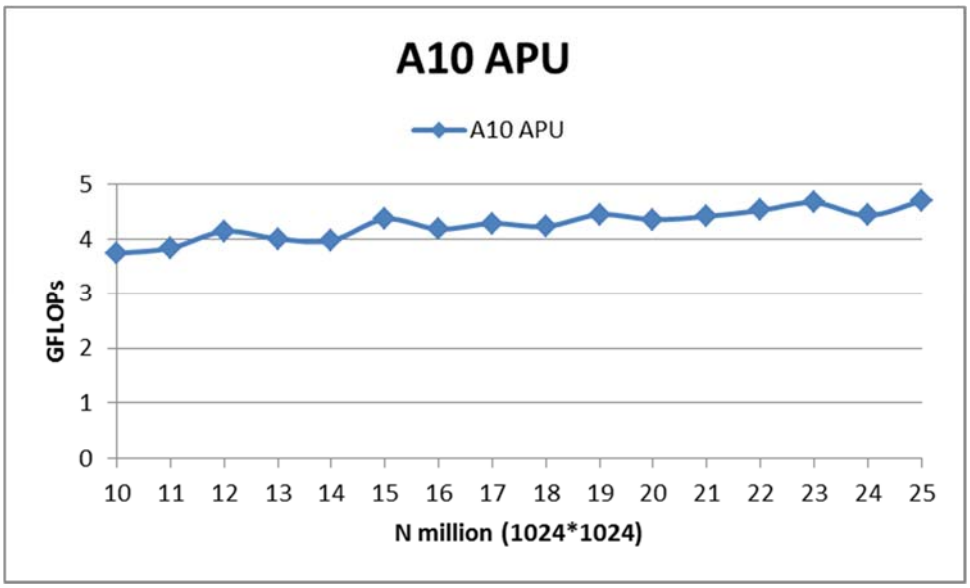


Figure 12. Reduction - AMD A10-7850K

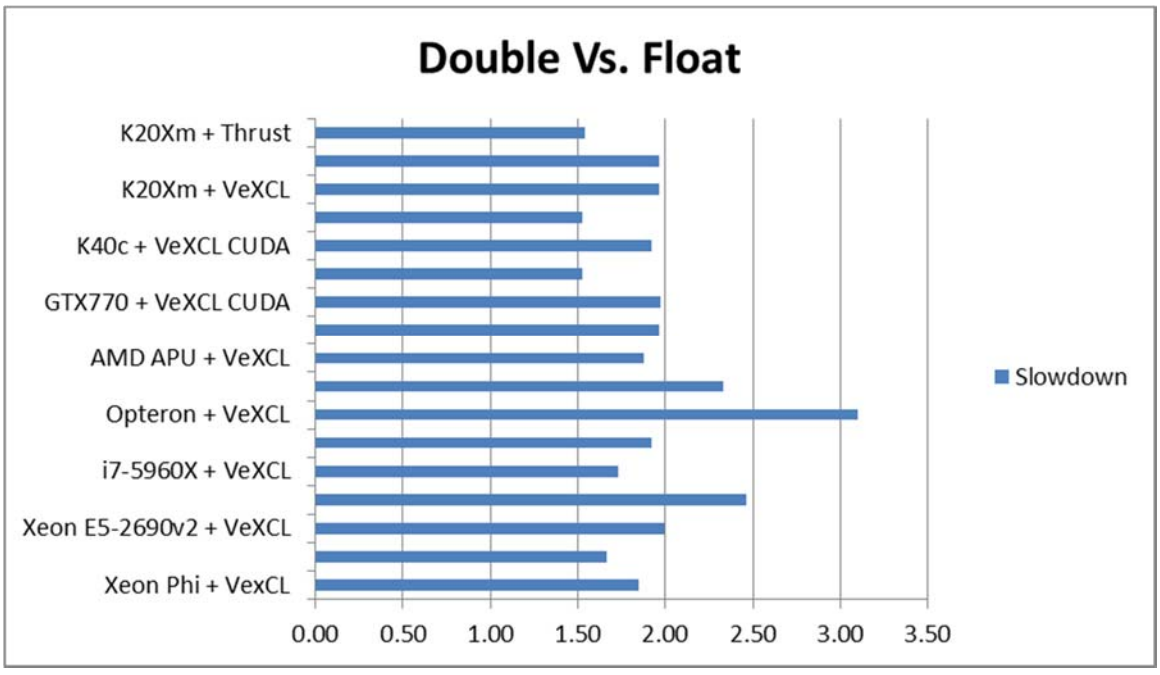


Figure 13. Reduction - Double Vs. Single Precision Slowdown

As seen from the results above, Thrust offered consistently higher performance than VexCL when the benchmarks were run on CPUs. However, the performance gap narrowed when the same benchmarks were run on GPUs. This tells us that, even though VexCL is a competent library, it's implementation on CPU platforms has scope for improcement. In terms of pure performance, NVidia GPUs proved to be most capable devices, even though their performance was two to three magnitudes lower than the advertised peak. AMD's A10 APU, owing to the fact that it has lower resources, was not able to match up to the same performance.

5.2 Prefix Scan

In an inclusive or prefix-scan each element of the output is the corresponding partial sum of the input range. For example, data[2] = data[0] + data[1] + data[2]. An exclusive scan is similar, but shifted by one place to the right. Since the access pattern is regular, the execution helps from hardware prefetcher which latches onto sequential access pattern. The metric measured was rate of keys or elements scanned - i.e. keys/sec. The element can be a tuple of key-values or individual elements, in which case the scan is performed on the element itself. The organization of libraries-backends-platforms looks as follows:

Library	**Thrust**		**VexCL**	
Backend	**OpenMP**	**CUDA**	**OpenCL**	**CUDA**
AMD A10-7850K			☑	
GTX 770			☑	☑
Tesla K20Xm		☑	☑	☑
Tesla K40c		☑		☑
Xeon Phi	☑		☑	
Xeon E5-2690v2	☑		☑	
i7-5960X	☑		☑	
Opteron	☑		☑	

Table 2. Organization of the Prefix Scan Benchmark

CPU results

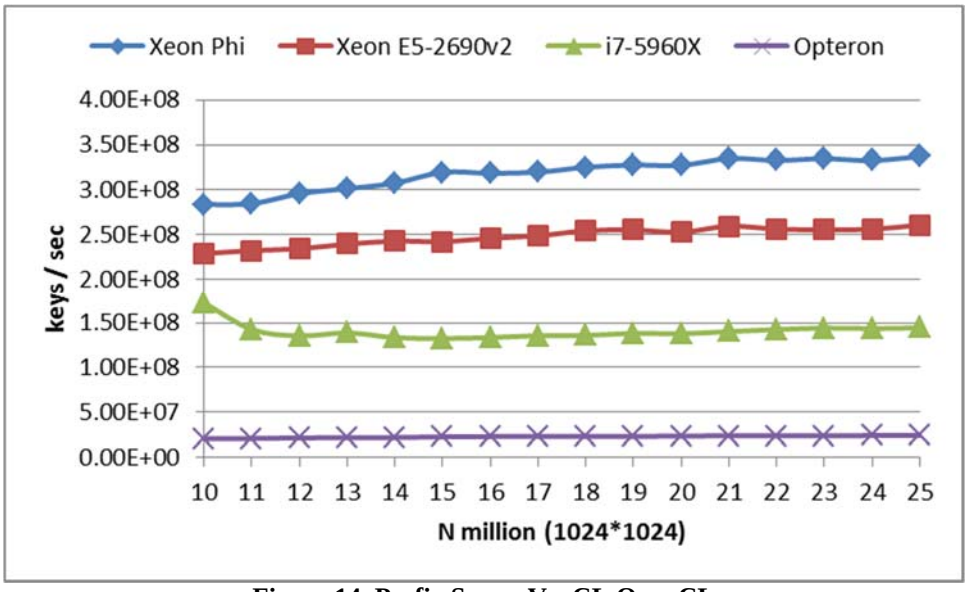


Figure 14. Prefix Scan - VexCL OpenCL

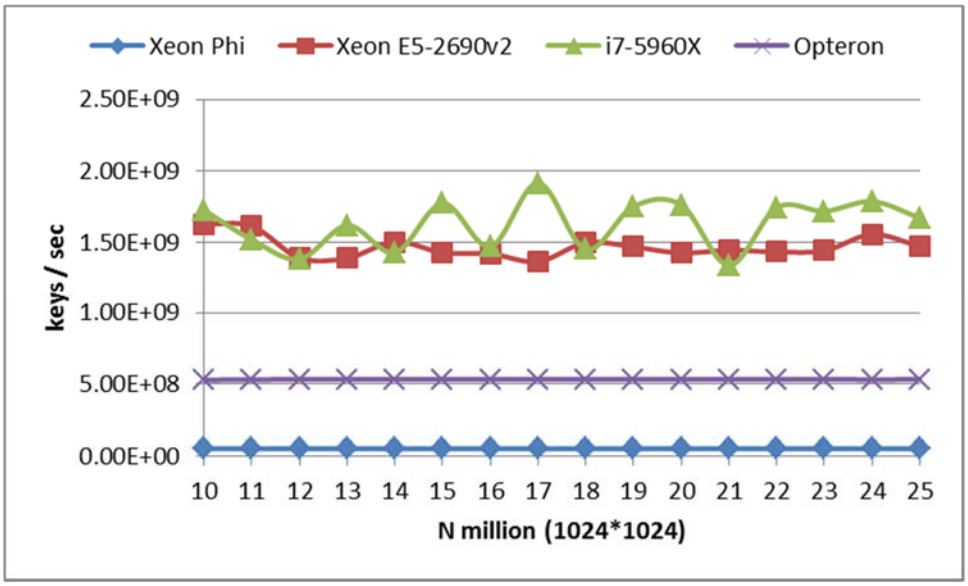


Figure 15. Prefix Scan - Thrust OpenMP

GPU results

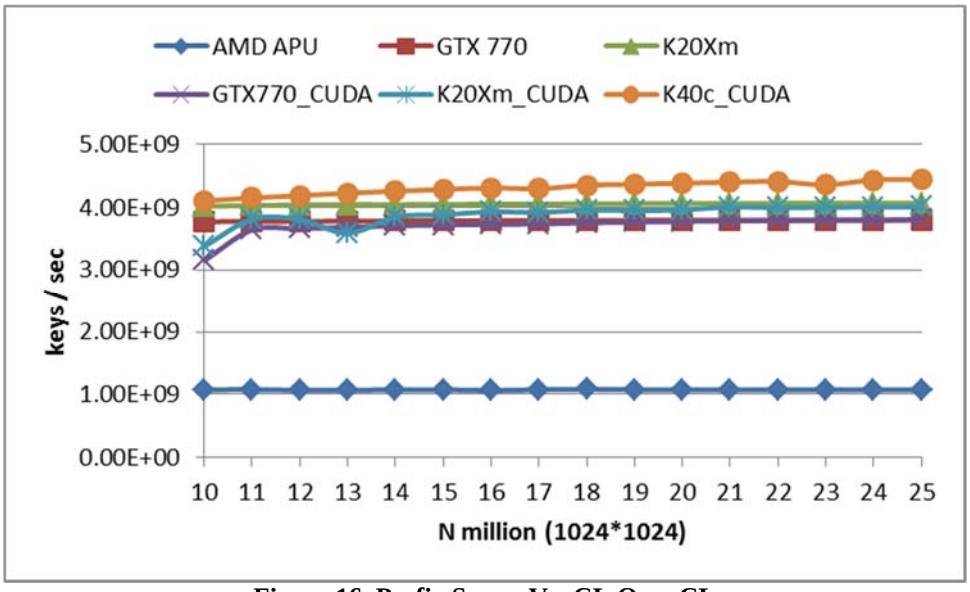


Figure 16. Prefix Scan - VexCL OpenCL

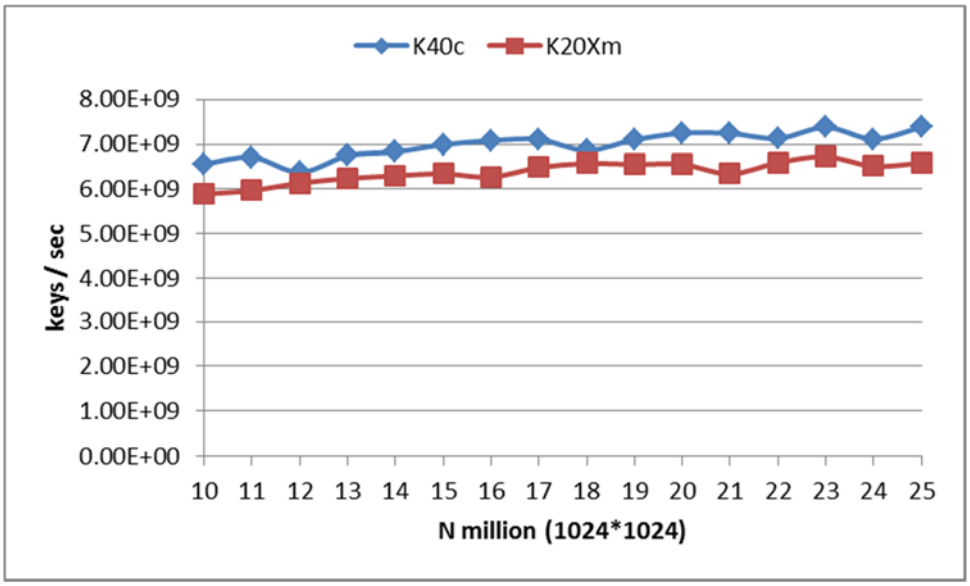


Figure 17. Prefix Scan - Thrust CUDA

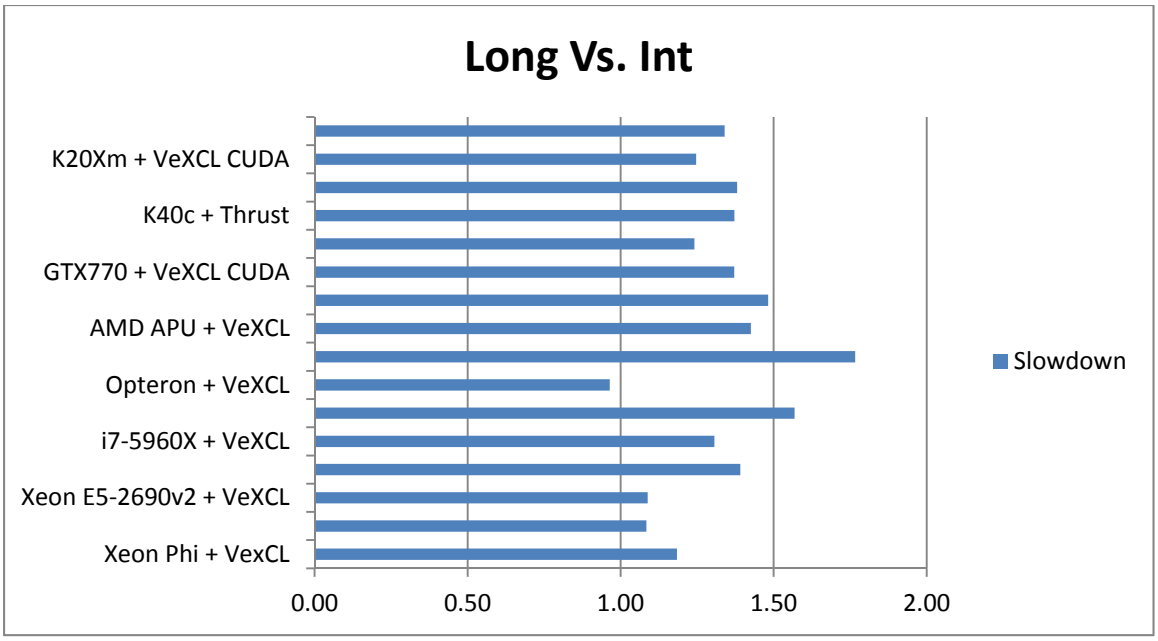


Figure 18. Prefix Scan - Long Vs. Integer Slowdown

As seen from the results above, the performance of Intel Xeon Phi was largely dependent on library being use – the highest performing benchmark being written with VexCL. Despite higher number of CPU core count, AMD Opteron system was slower than Intel based systems with lesser CPU cores.

5.3 Sort

Similar to the prefix scan, the performance for the sorting benchmark is measured in terms of keys/sec. However, the process of sorting random integers involved random accesses across the length of the vector. Further, this access pattern is dependent on sorting algorithm implements. As a result, sorting is much more taxing on memory subsystems and often not bound by compute capabilities of CPU. The organization of libraries-backends-platforms below shows libraries and backends with which benchmarks were compiling and the hardware platform they were run on:

Library	Thrust		VexCL	
Backend	**OpenMP**	**CUDA**	**OpenCL**	**CUDA**
AMD A10-7850K			☑	
GTX 770			☑	☑
Tesla K20Xm		☑	☑	☑
Tesla K40c		☑		☑
Xeon Phi	☑		☑	
Xeon E5-2690v2	☑		☑	
i7-5960X	☑		☑	
Opteron	☑		☑	

Table 3. Organization of the Sort Benchmark

CPU results

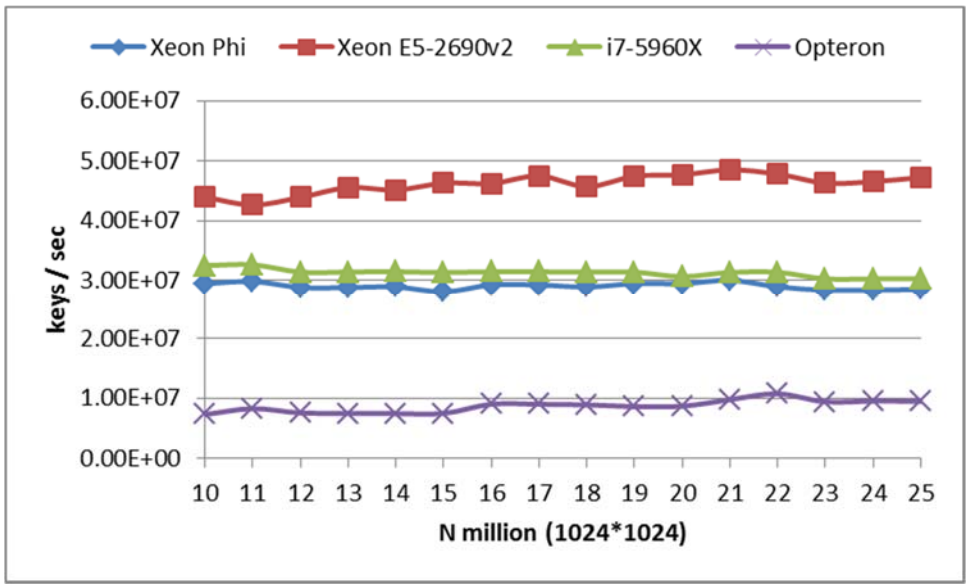


Figure 19. Sorting - VexCL OpenCL

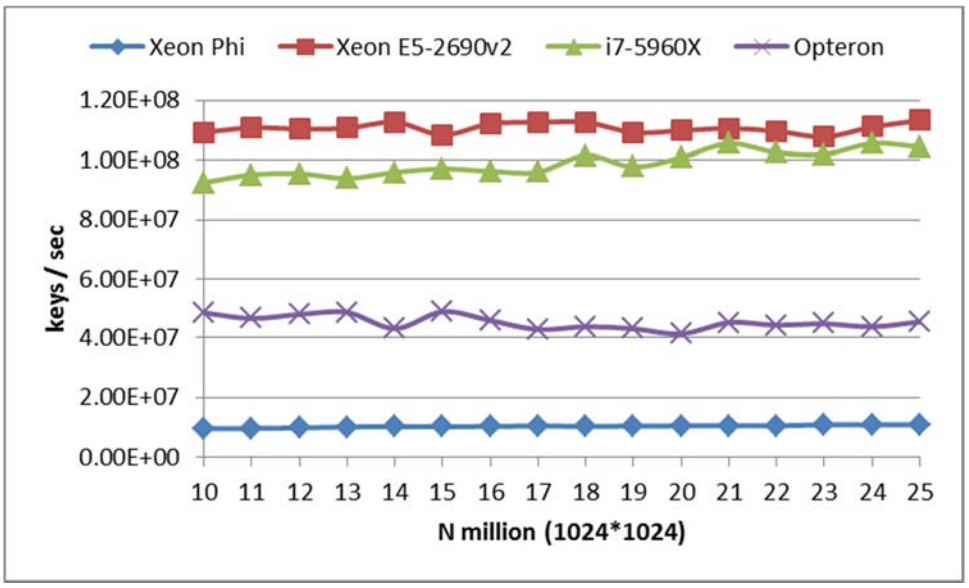


Figure 20. Sorting - Thrust OpenMP

GPU results

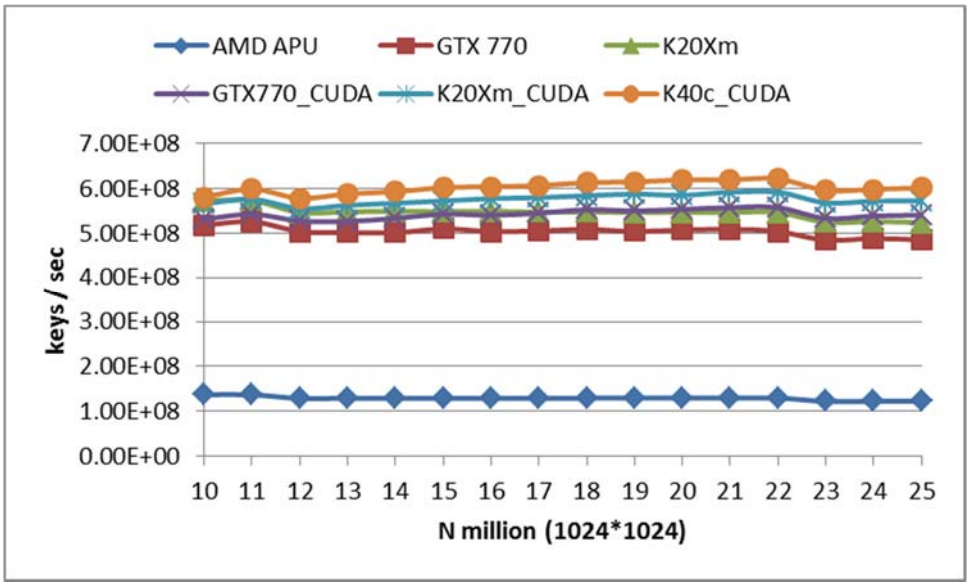


Figure 21. Sorting - VexCL OpenCL

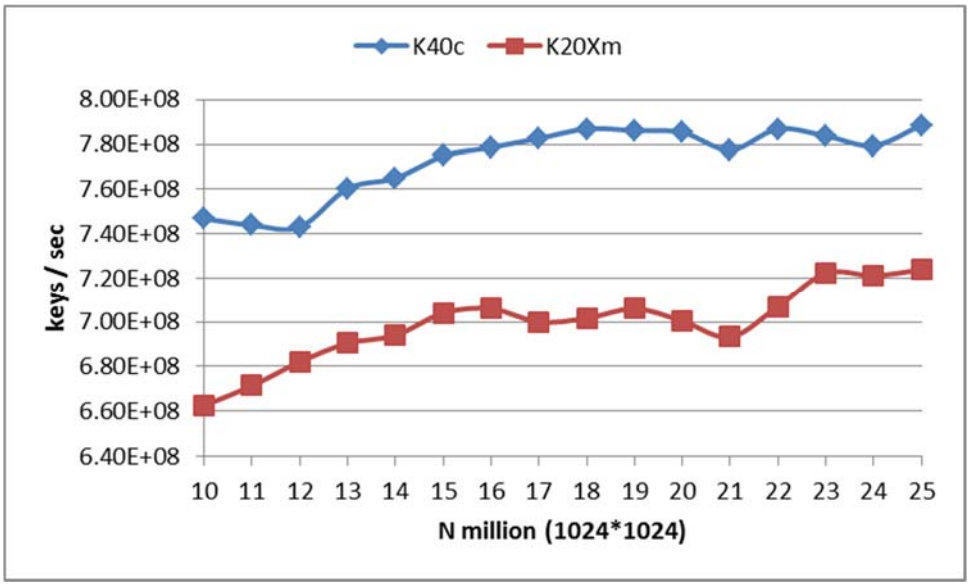


Figure 22. Sorting - Thrust CUDA

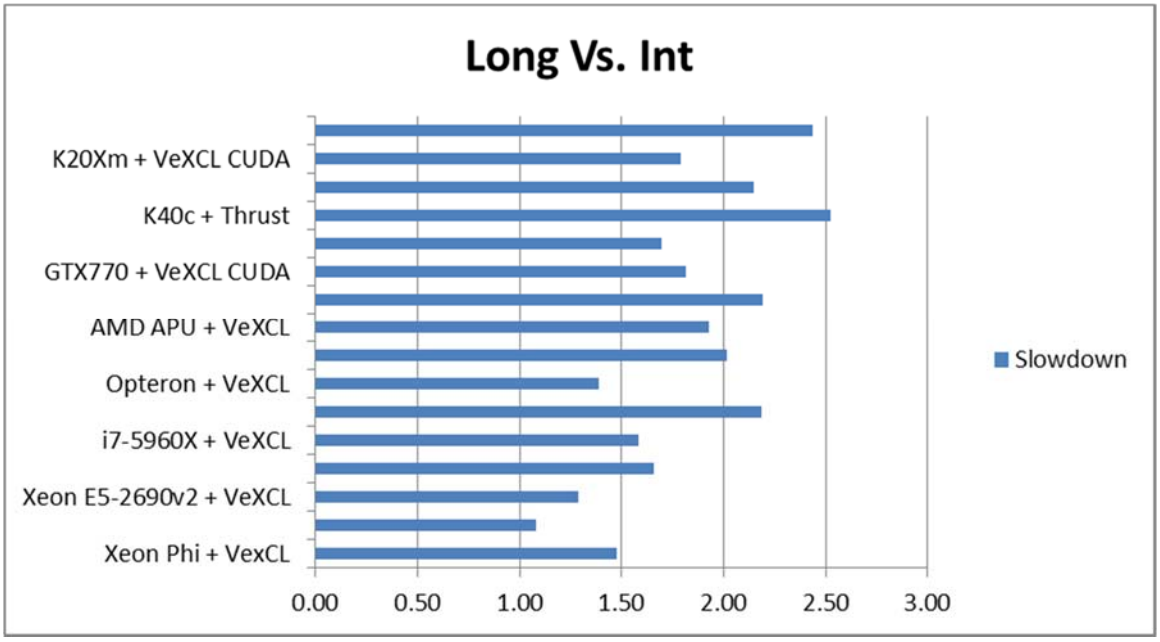


Figure 23. Sorting - Long VS. Integer Slowdown

As seen from the results above, sorting operating is much more I/O intensive than prefix scan, which results into soring rate that is about an order of magnitude lower than scanning rate. This results into flat performance profiles which indicate that the soring operation is bound by the memory bandwidth limits even at smaller input sizes.

5.4 SAXPY

SAXPY adds a scalar multiple of a real vector to another real vector. SAXPY stands for "Single-Precision A·X plus Y".

$$X = aX + Y$$

The result is another vector which, depending upon the implementation is returned as the result or overwritten in one of the inputs. Though similar to reduction operation, SAXPY involves three times as many memory access per element of output vector – two reads from input vectors X and Y and one write of the result of summation. Hence, for the platforms which are not bound by bandwidth, the FLOPS profile looks similar to reduction benchmark. The organization of libraries-backends-platforms looks as follows:

Library	**Thrust**		**VexCL**		**Blaze**			**MKL**
Backend	**OpenMP**	**CUDA**	**OpenCL**	**CUDA**	**OpenMP**	**CPP**	**Boost**	**OpenMP**
APU			☑					
GTX770			☑	☑				
Tesla K20Xm		☑	☑	☑				
Tesla K40c		☑		☑				
Xeon Phi	☑		☑		☑			☑
Xeon E5-2690v2	☑		☑		☑	☑	☑	☑
i7-5960X	☑		☑		☑	☑	☑	.
Opteron	☑		☑		☑	☑	☑	

Table 4. Organization of the SAXPY Benchmark

CPU results

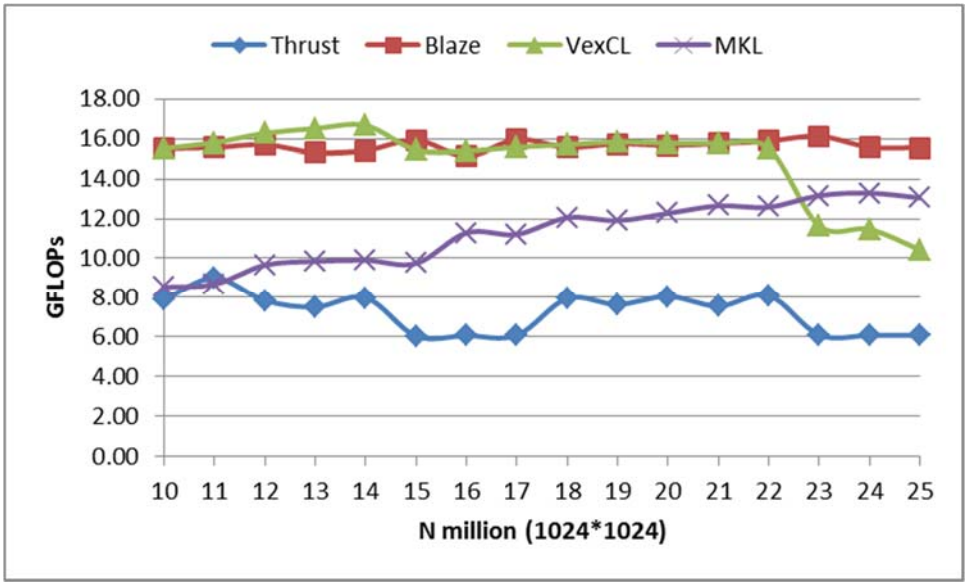


Figure 24. SAXPY - Intel Xeon Phi

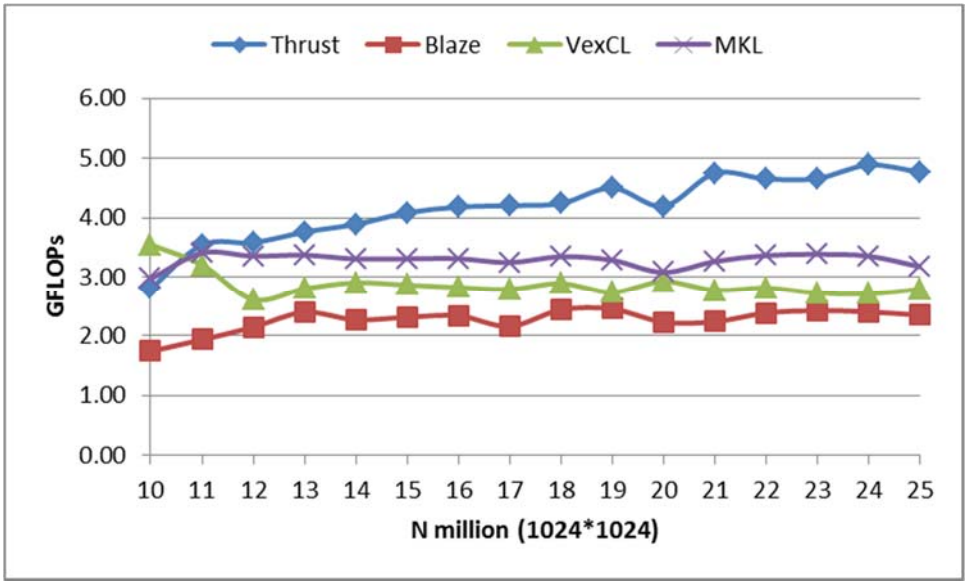


Figure 25. SAXPY - Intel Xeon E5-2690v2

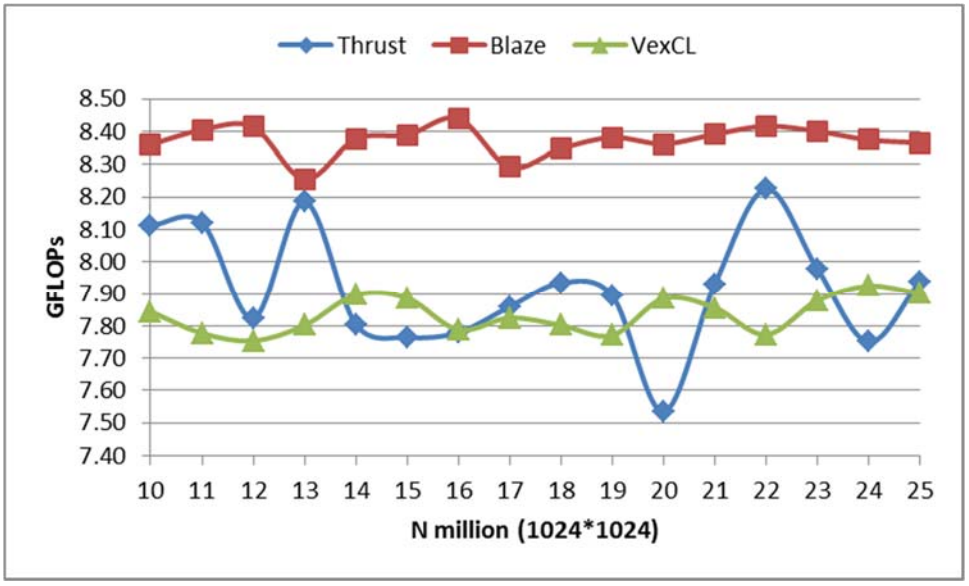


Figure 26. Intel i7-5960X

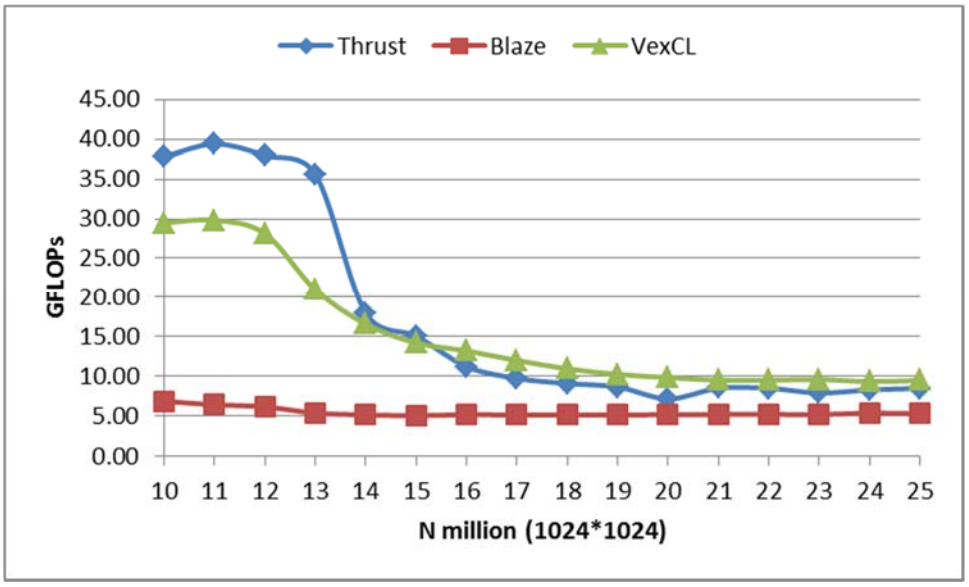


Figure 27. AMD Opteron 6274

GPU results

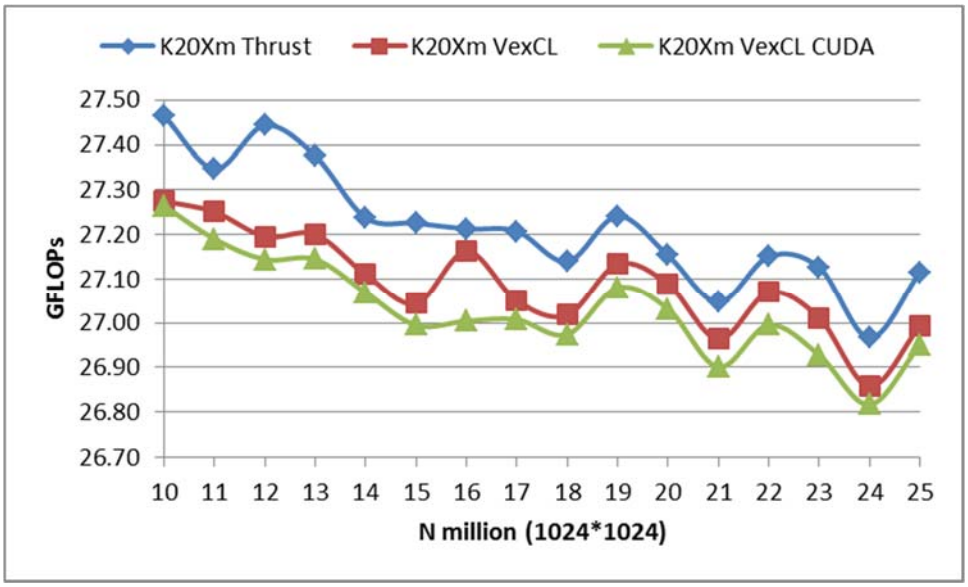


Figure 28. SAXPY - NVidia Tesla K20Xm

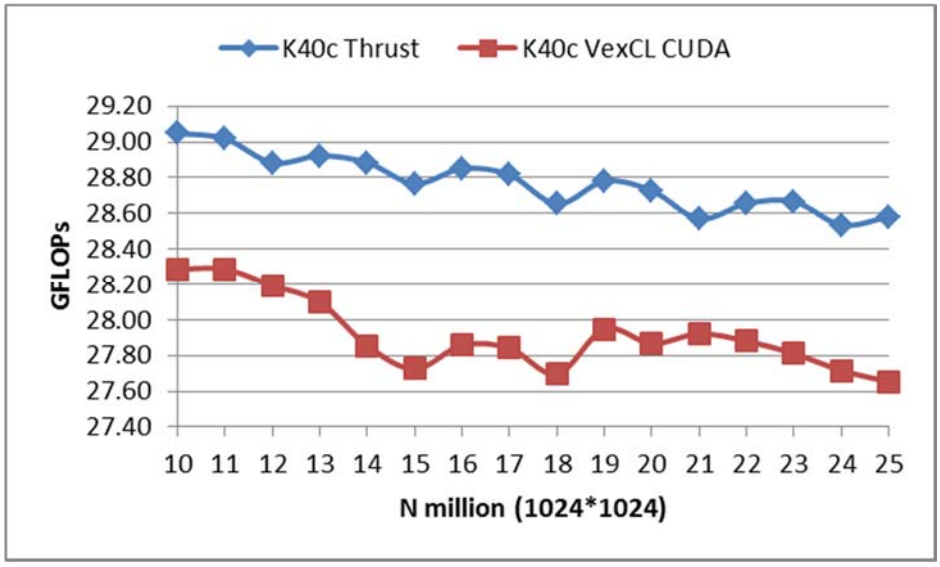


Figure 29. SAXPY - NVidia Tesla K40c

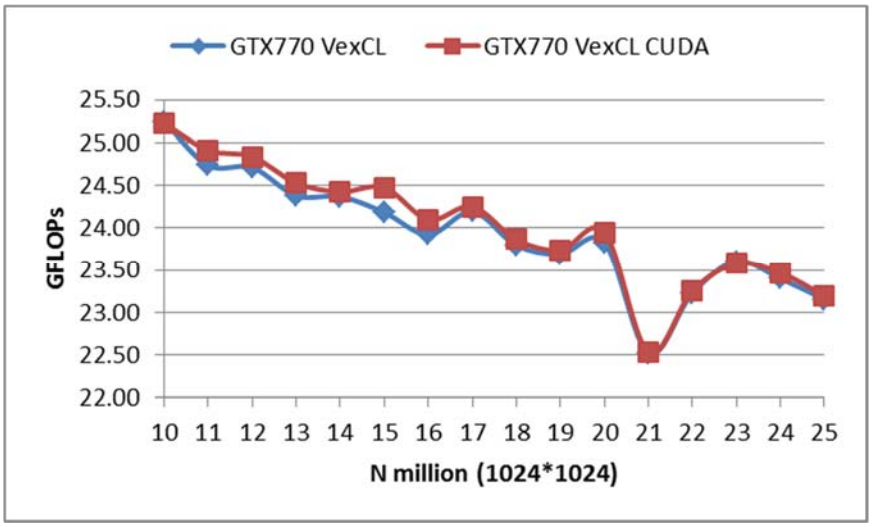


Figure 30. SAXPY - NVidia GTX 770

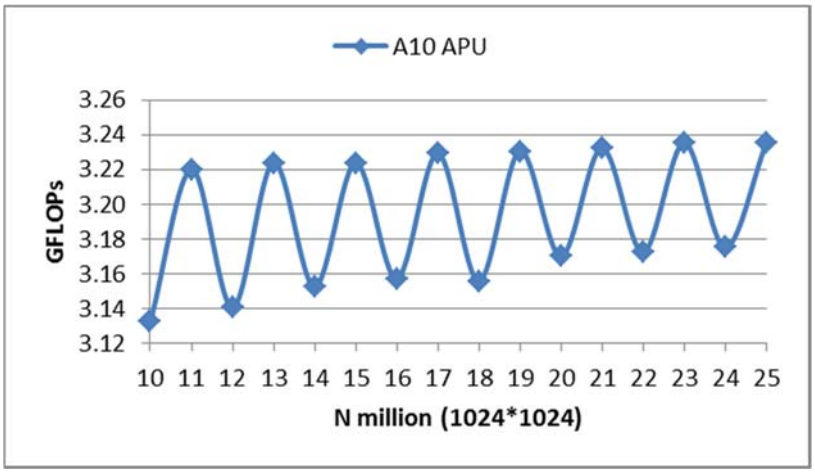


Figure 31. SAXPY - AMD A10-7850K

Blaze backends comparison

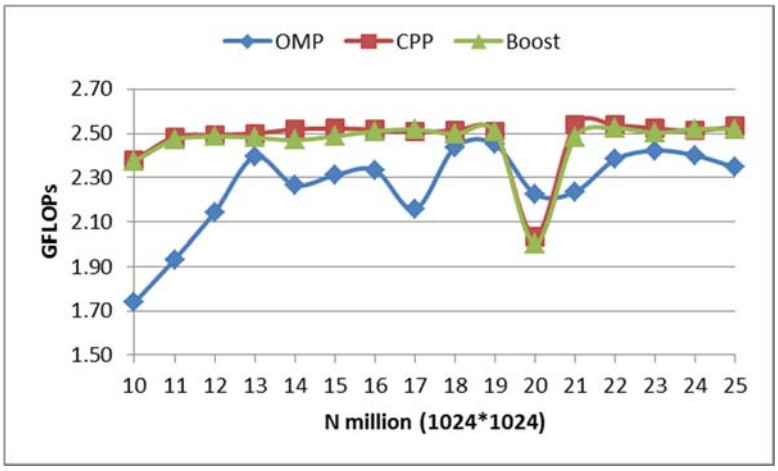


Figure 32. SAXPY - Intel Xeon E5-2690v2

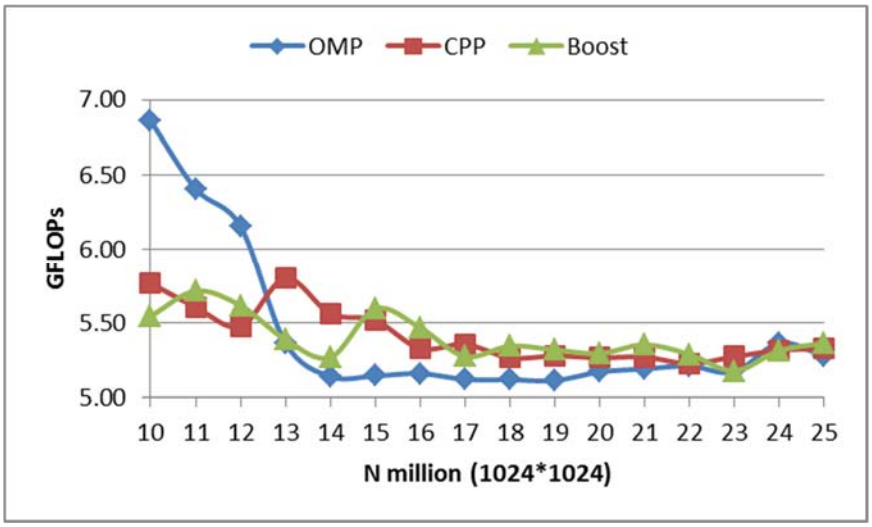


Figure 33. SAXPY - AMD Opteron 6274

As seen from the results above, all the hardware platforms except for Intel Xeon Phi and Intel Xeon E5-2690v2, hit the memory bandwidth limit between the specified input ranges. Hence, there is dip in the performance as size grows and the SAXPY operation becomes memory bound. The GFLOPs rate for Intel Xeon Phi and Intel Xeon E5-2690v2 continues to scale up, indicating that memory interface will saturate at sizes larger than those specified here. As for the Blaze library, the native support for threads in C++11 proved to offer higher performance than OpenMP backend. Although in terms of absolute performance, the gap is marginal.

6 Profiling

Software profiling is a type of dynamic program analysis that involves the execution of the program that the user studies to determine, for example, memory usage, time complexity, the usage of particular instructions, or the frequency and duration of function calls [17]. Profiling is achieved by instrumenting either the program source code or its binary executable form using profiler tools such as Linux Perf or Intel VTune. Profilers

may use a number of different techniques, such as event-based, statistical, instrumented, and simulation methods. They can be used to generate different types of profiles such as:

- Flat profile: Flat profilers compute the average call times, from the calls, and do not break down the call times based on the callee or the context
- Call graphs: It shows the call times, frequencies of the functions, and also the call-chains involved based on the callee.
- Input sensitivity: Input-sensitive profilers add a further dimension to flat or call-graph profilers by relating performance measures to features of the input workloads, such as input size or input values.

In this study, we worked with Linux Perf, Intel VTune and OProf.

6.1 Linux Perf

The Linux perf command can instrument CPU performance counters, tracepoints, kprobes, and uprobes (dynamic tracing). It is capable of lightweight profiling. It is included in the Linux kernel, under tools/perf, and is frequently updated and enhanced. It works with performance counters which are CPU hardware registers that count hardware events such as instructions executed, cache-misses experienced, or branches mispredicted. They form a basis for profiling applications to trace dynamic control flow and identify hotspots. Perf provides generalized abstractions over hardware specific capabilities. Among others, it provides per task, per CPU and per-workload counters, sampling on top of these and source code event annotation.

As shown in the example below, a perf record command can be used to trace the page faults that occurred, which fires when an I/O request is issued to service the page fetch from disk to main memory. Options included were `-a` to trace all CPUs, and `-g` to capture call graphs (stack traces). Trace data is written to a perf.data file, and tracing ended when `Ctrl-C` is hit or when executable finishes running. A summary of the perf.data file can be printed using `perf report`, which builds a tree from the stack traces, coalescing common paths and showing percentages for each path. Here is an example of one such `perf report`:

```
# ========
# captured on: Mon Nov 10 22:35:41 2014
# hostname : euler16
# os release : 2.6.32-358.6.2.el6.x86_64
# perf version : 2.6.32-358.23.2.el6.x86_64.debug
# arch : x86_64
# nrcpus online : 64
# nrcpus avail : 64
# cpudesc : AMD Opteron(TM) Processor 6274
# cpuid : AuthenticAMD,21,1,2
# total memory : 132151156 kB
# cmdline : /usr/bin/perf record -a -e page-faults -o rs_16.faults -- ./dress
# event : name = page-faults, type = 1, config = 0x2, config1 = 0x0, config2 = 0x0, excl_usr = 0, excl_kern = 0, excl_host = 0, ex
# HEADER_CPU_TOPOLOGY info available, use -I to display
# HEADER_NUMA_TOPOLOGY info available, use -I to display
# ========
#
# Samples: 11K of event 'page-faults'
# Event count (approx.): 189401
#
# Overhead,Command,Shared Object,Symbol
45.91%,     dress,libc-2.12.so              ,[.] _int_malloc
9.51%,     dress,libChronoEngine_Parallel.so,[.] thrust::detail::enable_if_non_const_reference_or_tuple_of_iterator_references<thru
6.85%,     dress,libChronoEngine_Parallel.so,[.] thrust::detail::enable_if_non_const_reference_or_tuple_of_iterator_references<thru
5.85%,     dress,libChronoEngine_Parallel.so,[.] void thrust::system::cpp::detail::assign_value<thrust::system::cpp::detail::tag, t
5.78%,     dress,libc-2.12.so              ,[.] _wordcopy_fwd_aligned
4.33%,     dress,libChronoEngine_Parallel.so,[.] void thrust::detail::host_generate_functor<thrust::detail::fill_functor<real3> >::
4.14%,     dress,dress                     ,[.] void thrust::detail::host_generate_functor<thrust::detail::fill_functor<float> >::
```

The `perf report` output shows that of 189,401 events that were traced (page faults), 46% of them were from a malloc command. These were issued directly from the executable `dress` and came from standard C library. Walking down the stacks, about

half of these 32% were from the calls in Chrono, a middleware library for mechanical simulation developed by colleagues in the Simulation-Based Engineering Lab [18].

Perf instruments by sampling events and writing event data to a kernel buffer, which is read at an asynchronous rate by the `perf` command and written to the `perf.data` file. This file is then read by the perf report or perf script commands. When using the sampling mode with perf record, capture files can quickly become hundreds of MB. It depends on the rate of the event tracing: the more frequent, the higher the overhead and larger the `perf.data` size.

6.2 Intel VTune

Intel's VTune Amplifier is a performance profiling tool for C, C++, and Fortran code that can identify where in the code time is being spent in both serial and threaded applications. For threaded applications it can also determine the amount of concurrency and identify bottlenecks created by synchronization primitives. VTune Amplifier uses dynamic instrumentation and thus does not require use of Intel compilers or the use of any special compiler flags.

The analysis types include: hotspots, concurrency and locks and waits. Hotspots will profile the code's execution to determine which functions are consuming the most time and thus are targets for optimization. The hotspots analysis includes timing information from all threads and from sub-processes. The concurrency analysis assesses how well a threaded application takes advantage of multi-core hardware and identifies functions and times during execution where available CPU cores aren't fully utilized. The locks and

waits analysis adds the ability to identify synchronization points that contribute to underutilization of CPUs. VTune Amplifier uses sampling to gather profile information and should only incur a 5% execution-time overhead.

6.3 OProfile

Similar to Perf and VTune, OProfile [19] is a system-wide statistical profiling tool for Linux. OProfile can profile an entire system or its parts, from interrupt routines or drivers to user-space processes. It has low overhead. The workflow with OProfile is slightly different when compared to other tools. It draws on user-space tools such as `opcontrol, opreport and opgropf`, and is typically used as follows:

```
opcontrol ---start

 run <example>

 opcontrol --dump

 opreport -l <example> > <outputfile>

 opcontrol --stop (stops collecting the data)

 opcontrol --shutdown (Stops the demon)

 opcontrol --reset (clears the profile data which was stored in
the sample file given)
```

7. Software Setup for the Performance Database

The end-to-end process of benchmarking can be viewed as a pipeline of tasks, in which the individual stages are linked with each other. Therefore, it becomes essential to have a well-designed software setup that helps tie all the pieces together.

7.1 Github structure of the source repository

The process of benchmarking starts with writing the programs to be used on multiple hardware systems, potentially by multiple users. For the purpose of source control management, we use Github to setup a collaboration workflow around a shared repository which allows different contributors to collaborate on the benchmark code, results and analysis. The source code, results, profiling data and all other reports are checked-in at repository located at https://github.com/uwsbel/benchmarks. The top level organization is as per the problems we work on. So the root has directories - Reduce, Scan, Sort, SAXPY and SpMV. Inside each, there is a separate directory for each code base. Each has its own source files, Makefiles, raw data collected etc. We use a shared repository model to maintain the code. There is a "master" branch for tried and tested code. Apart from that, each collaborator works on a development branch while prototyping. For example, a given user can work on branch "devel_user" and when it has been established that code is functional and the right results have been collected, it can be merged with "master" branch.

7.2 Performance metrics Database

Given a multitude of parameters with which each benchmark can be configured with, and also the number of platforms at our disposal, a single run can generate a lot of data. The amount of data or results further multiplies when all the combinations are run exhaustively. At this scale, analyzing the outputs and interpreting the outcomes to gain meaningful insights becomes overwhelming. To help with this process and to provide additional flexibility, we maintain all the results generated in a SQL-based database. This database is used to capture all the relevant information for a given run of the benchmark. Given below is the schema for the database and description of tables.

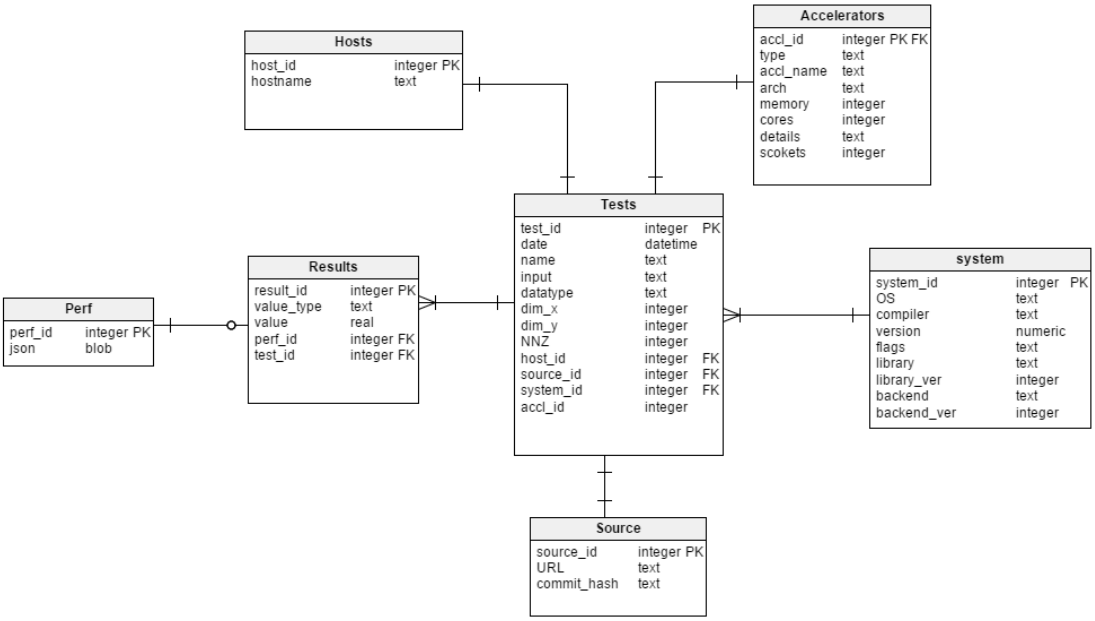


Figure 34. Database Schema

Description of tables:

- Tests

Column Name	**Data Type**	**Comments**
test_id	Integer	Primary Key, identifies unique row
date	Datetime	Date of the test run
name	Text	Name of the benchmark
input	Text	Input type – Vector or Matrix name
datatype	Text	Datatype of the elements

dim_x	Integer	X Dimension
dim_y	Integer	Y Dimension
NNZ	Integer	Number of non-zero entries
host_id	Integer	Foreign Key, identifies host
accel_id	Integer	Foreign Key, identifies accelerator used
system_id	Integer	Foreign Key, identifies compilers, libraries etc.
source_id	Integer	Foreign Key, identifies version of source code

Table 5. Database Table – Tests

- Results

Column Name	Data Type	Comments
result_id	Integer	Primary Key, identifies unique row
time	Real	Time of the run measured
time_type	Integer	Measured time type – Inclusive / Exclusive
value	Text	Metric measured
value_type	Real	Measured metric type
perf_id	Integer	Foreign Key, identifies performance profiling record
test_id	Integer	Foreign Key, identifies the test run

Table 6. Database Table – Results

- Perf

Column Name	Data Type	Comments
perf_id	Integer	Primary Key, identifies unique row
json	Blob	Performance profiling data

Table 7. Database Table – Perf

- Hosts

Column Name	Data Type	Comments
host_id	Integer	Primary Key, identifies unique row
hostname	Text	Host name of the system

Table 8. . Database Table – Hosts

- Accelerators

Column Name	Data Type	Comments
accl_id	Integer	Primary Key, identifies unique row
type	Text	Type of the accelerator – CPU, GPU, MIC
accl_name	Text	Name of the accelerator
arch	Text	Architecture name
memory	Integer	Amount of memory in GBs
cores	Integer	Number of cores
details	Text	Link to specifications from manufacturer

Table 9. Database Table – Accelerators

- Systems

Column Name	Data Type	Comments
system_id	Integer	Primary Key, identifies unique row
OS	Text	Operating system
compiler	Text	Compiler used
version	Numeric	Compiler version
flags	Text	Compiler optimization flags
library	Text	Math library used
library_ver	Numeric	Library version
backend	Text	Parallelization backend
backend_ver	Numeric	Backend version

Table 10. Database Table – Systems

- Sources

Column Name	Data Type	Comments
source_id	Integer	Primary Key, identifies unique row
URL	text	Link to repository hosting code
commit_hash	text	Hash for version of the code from git

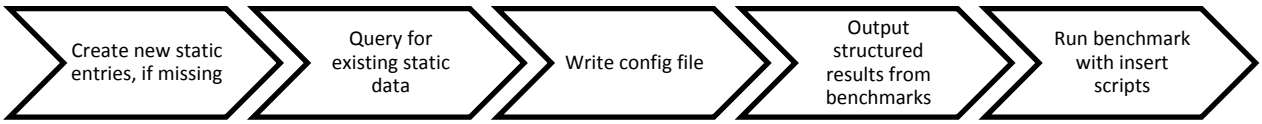
Table 11. Database Table – Sources

7.3 Populating the database

The process of creating entries in the tables of the database is automated with the help of Python scripts. This is achieved with the help of SQLAlchemy, which is an open source SQL toolkit and object-relational mapper (ORM) for the Python programming language. The SQLAlchemy Expression Language presents a system for representing relational database structures and expressions using Python constructs. The Expression Language presents a method of writing backend-neutral scripts that can execute SQL expressions as instructed.

There are two types of tables in above schema – Tests and Results, for which new entries are inserted every time a benchmark is run. This are classified as dynamic tables. Other tables, which include tables for Hosts, Accelerators, Systems, Sources and Perf, are

classified as static. The static data is entered before any benchmark is run and the rows in Tests and Results link against these rows at runtime. A JSON configuration file called config.json is used for this purpose. A typical workflow then looks as following:



The command line interface offers following utilities:

- put.py: Create static entries for Hosts, Accelerators, Systems
- get.py: Given a table name, get all rows to identify the primary keys. Use grep in conjunction with get.py to filter the results.
- insert.py: Run benchmarks with insert.py to populate Tests and Results tables with dynamic data. The benchmarks need to output the results in structured format for this to work. The output is essentially a list of key-value pairs where keys directly map to columns in Tests and Results tables.
- config.json: This configuration file is used by insert.py script to correctly setup the environment. It provides the URL for the server where the database is hosted. It also stored foreign keys for static rows, against which dynamic rows link. A sample configuration file looks as follows:

```
{
  "db_url": "sqlite:///perfdb",
  "host_id": "3",
  "accl_id": "6",
  "system_id": "30",
  "source_id": "1",
  "perf_id": "1"
}
```

7.4 Web Based Interface for the Database

In addition to the command line interface, we also have designed a web-based interface written in HTML and PHP that can read and write from the database from an internet browser. The interface offers a functionality that is equivalent of what command line utilities have to offer. In that it consists of three parts:

- The dynamic data – data inserted into Tests and Results – still remains the responsibility of insert.py script described earlier. The website has option to query the static data from Hosts, Accelerators, Systems and Sources table. This is data with which the benchmarks link up with the help of config.json file. This is the first step that needs to be completed manually.
- The second step, which is also manual, is optional. If any of the static data required for the testing is not already available in the database, the website can be used to insert it into respective tables and later link with the same as described in step one.
- As with the command line interface, once the configuration file is set up, benchmarks need to be run using the insert.py script. This step is independent of web interface.
- Finally, once the required benchmarks are run and the data is inserted into the database, one can use the website to query the database. There are two types of queries supported – Q*uick queries* to see the results either by the benchmark or hardware use, and *detailed queries* to perform more flexible querying and analysis. Detailed queries allow selecting all the fields that are available from all the tables and also filtering the results using standard SQL syntax. There is also

an option to export the results shown as a Comma Separated Value (CSV) file, which can later be used in conjunction with programs such as Excel, Matlab, or LaTeX.

8. Lessons Learned

This study allowed us to get a realistic picture of the gap that is present between the performance as marketed by hardware and software vendors and the real-world performance as seen on systems deployed in a typical high performance computing server environment. For example, the theoretical peak rate that NVidia Tesla K40c can deliver is about 1.66 Teraflops in double precision mode. However, these numbers are very much dependent on problem under study. Sparse matrix vector multiplication, which is very demanding on the memory subsystem in terms of bandwidth, can only reach tens of GFLOPs rate, which is two to three orders of magnitude lower than the maximum possible, depending upon the input matrix. In nutshell, one should be very cautious when taking the advertised peak rates for a hardware asset as an immediate proxy for performance in scenarios of practical relevance.

On the other hand, it is not only the application requirements that dictate the performance profiles. The underlying hardware configuration and software stack on top which the applications run also play a crucial role. As seen in Fig.25, the commercial libraries such as Thrust and Intel MKL, which are heavily optimized, tend to outperform the community driven alternatives. In most cases, Thrust based benchmarks had higher performance than VexCL based OpenCL implementations where everything else – such

as backend, compilers etc., remained same. This tells us that there remains room for improvement for the numerical computing libraries such as Boost and VexCL.

Finally, we also learned the importance of having a clearly designed and well-thought workflow when it comes to something that is complex as benchmarking. It helps to have an understanding of what utilities the operating systems such as Linux have to offer – such as Perf, makefiles and scripting for automation. While setting up the software infrastructure, we came across many software tools and techniques that anyone who is working in computational engineering can benefit from. Github is very helpful in setting up a hybrid of centralized and distributed source control management to be used in collaborative environments. Tools such as SQL based databases also aid the process of data archiving and analysis. The decision to go with an embedded database such as SQLite was a result of tying the workflow with Github based collaboration, which allows easier access control. A more traditional approach of deploying the database on a dedicated server brings along additional logistical complexity in terms of security and more software dependencies when running across multiple hosts. In its current form, the entire setup is self-contained and independent of any external software dependencies. Therefore, a typical user only needs to clone or fork the Github repository to the local machine in order to use it in its entirety. This simplicity comes at the price of somewhat reduced functionality in terms of what services the database and the web-based interface has to offer.

9. Future Work

The work done as a part of this study can act as a platform for future studies and projects carried in the Simulation Based Engineering Lab (SBEL). With constantly evolving hardware and software packages that update often, a standardized benchmarking system can play an important role in getting the realistic picture of the state-of-the-art. Some of the areas that can be worked on in near-future deal with the software tools being used: transition from makefiles to a more higher level abstraction of CMake for compiling the benchmarks, using the database as server instead of embedding it into the source repository, and building a web-app to plot the results from web-based queries. In terms of more long term objectives, the database can act as a central repository of wealth of data spanning across multitude of systems. It will only grow in value as more researchers use it and as more time passes it can be used to study how the performance of primitive/basic computational building blocks used in computation engineering codes has evolved.

10. References

1. Zhang, X. (2001). *Application-Specific Benchmarking.* Retrieved from http://www.eecs.harvard.edu/~syrah/application-spec-benchmarking/publications/thesis.pdf

2. Demidov, D. (n.d.). *VexCL Documentation*. Retrieved from https://ddemidov.github.io/vexcl/

3. *Thrust :: CUDA Toolkit Documentation*. (n.d.). Retrieved from http://docs.nvidia.com/cuda/thrust/

4. *Intel Math Kernel Library*. (n.d.). Retrieved from https://software.intel.com/en-us/intel-mkl

5. *blaze-lib - A high performance C++ math library*. (n.d.). Retrieved from https://code.google.com/p/blaze-lib/

6. *Prefix sum: Wikis*. (n.d.). Retrieved from http://www.thefullwiki.org/Prefix_sum

7. *IBM Knowledge Center*. (n.d.). Retrieved from http://www-01.ibm.com/support/knowledgecenter/SSFHY8_5.2.0/com.ibm.cluster.essl.v5r2.essl100.doc/am5gr_hsaxpy.htm

8. Catanzaro, B. (n.d.). *OpenCL™ Optimization Case Study*. Retrieved from Diagonal Sparse Matrix Vector Multiplication Test: http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-diagonal-sparse-matrix-vector-multiplication-test/

9. *AMD Opteron™ Processor Solutions*. (n.d.). Retrieved from http://products.amd.com/pages/OpteronCPUDetail.aspx?id=760

10. *Intel® Core™ i7-5960X Processor Extreme Edition*. (n.d.). Retrieved from http://ark.intel.com/products/82930/Intel-Core-i7-5960X-Processor-Extreme-Edition-20M-Cache-up-to-3_50-GHz

11. *Intel® Xeon® Processor E5-2690 v2*. (n.d.). Retrieved from http://ark.intel.com/products/75279/Intel-Xeon-Processor-E5-2690-v2-25M-Cache-3_00-GHz

12. *Intel® Xeon Phi™ Coprocessor 5110P*. (n.d.). Retrieved from http://ark.intel.com/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1_053-GHz-60-core

13. *Tesla GPU Accelerators for Servers*. (n.d.). Retrieved from http://www.nvidia.com/object/tesla-servers.html

14. *Tesla K20 GPU Active Accelerator*. (n.d.). Retrieved from http://www.nvidia.com/content/PDF/kepler/Tesla-K20-Active-BD-06499-001-v04.pdf

15. *GeForce GTX™ 700 Series*. (n.d.). Retrieved from http://www.nvidia.com/gtx-700-graphics-cards/gtx-770/

16. *AMD A-Series APU Processors*. (n.d.). Retrieved from http://www.amd.com/en-us/products/processors/desktop/a-series-apu

17. Banerjee, S. (n.d.). *Performance Tuning of Computer Systems.* Retrieved from http://www.site.uottawa.ca/~mbolic/elg6158/Subhasis_profiling.pdf

18. Tasora, A., & Negrut, D. (n.d.). *ProjectChrono.*Retrieved from http://projectchrono.org/chronoengine/index.html .

19. *OProfile - A System Profiler for Linux*. (n.d.). Retrieved from http://oprofile.sourceforge.net/news/