Universidade do Minho

Masters in Computer Engineering

Advanced Architectures

Matrix multiplication: Performance Analysis

José Carlos Lima Martins A78821

Miguel Miranda Quaresma A77049

February 3, 2019

Abstract

Performance engineering is an area of increasing importance as applications such as scientific simulations increase in complexity and require developers to be aware of the underlying hardware and optimization techniques available to achieve respectable performance levels. The current paper presents a walk-through of the optimization of a dot product implementation through the application of different techniques. The impact of this techniques is measured by recording the execution time of each kernel as well as using hardware counters to gain further insights in how each technique affects the underlying system. To aid in the optimization techniques a Roofline model is elaborated for the hardware platform used for testing. Several techniques are then used to improve performance, ranging from block optimization to the use of SIMD architectures such as compiler guided vectorization or GPUs (NVIDIA K20M) and many cores (KNL) to achieve peak floating point performance.

1 Introduction

Parallel computing is the new paradigm for developers who wish to extract the maximum performance possible out of their applications. However, this paradigm doesn't dispense other optimization techniques such as vectorization or memory access locality and, in some cases, requires their use to obtain implementations that scale in modern hardware. The present work aims to implement, in an incremental way, several optimization techniques culminating in the use of two distinct hardware approaches to high performance computing: Many Cores vs. GPUs.

2 Characterization of hardware platforms

2.1 Main laptop - hardware specifications

The hardware specifications for the main laptop (see A.2) were obtained by running `lscpu` and `cat /proc/meminfo` in bash and by checking the manufacturer data sheets for both the CPU(Intel 8250U) and main memory.

2.1.1 Peak Memory Access Bandwidth - STREAM benchmark

Off-chip (peak) memory bandwidth is one of the most determining factors in computer performance and can be measured in a number of ways, for this report the STREAM benchmark was used [5]. To measure off-chip memory bandwidth we needed to ensure that the data used in the benchmark didn't fit in the last level of cache of each platform. In the case of the Intel 8285U processor, the L3 cache is non inclusive, meaning that the sum of the L2 and L3 cache($1 + 6 = 7$MiB) is used to determine the minimum data set size that doesn't fit in cache. As such, the following formula was used:

$$\frac{7 * 1024^2}{8} = 917504$$

Thus the number of elements per array was set to 1048576, which yielded a total of 8MB per array. The STREAM

benchmark was run 10 times and the median of the values was used as the peak memory bandwidth for this configuration(see A.3), which was equal to: 8.899 GiB/s.

As for the Cluster node 662, the last level cache has 30MiB, thus each array was set to 4063232 elements:

$$\frac{30 * 1024^2}{8} = 3932160$$

As with the main laptop, the benchmark was run 10 times and the median of the values taken as the off-chip peak memory bandwidth: 23.946 GiB/s(see A.3).

One thing to note is the fact that each STREAM benchmark presents four different results, each based on the operations performed to measure memory bandwidth. Given the nature of the present work (matrix multiplication), the fourth value (`TRIAD`) was chosen, since it is the result of the following operations: `a(i) = b(i) + q*c(i)` , which provides a more accurate representation of the operations involved in the dot product of two matrices.

2.2 Roofline Model's

The Roofline Model is extremely useful in identifying if and what optimizations can be applied to a given kernel [9]. It does so by relating arithmetic intensity (flops/ byte) with floating point performance (GFlops/s) and memory performance (GiB/s) and setting a roof for the attainable performance in a given machine/configuration. Additionally, by defining ceilings that further limit performance according to each optimization technique (vectorization, memory locality, etc), it enables easy identification of the easiest optimization techniques that result in the biggest performance gains. For each of the hardware platforms described earlier(A.1 and A.2) we elaborated a Roofline Model using the data already presented.

Figure 1: Merged Roofline Model w/o ceilings

One of the key takeaways from this kind of model is the maximum performance for CPU vs. Memory bound kernels. The point where both lines (peak memory bandwith and peak floating point performance) intersect, called the ridge point, determines how hard it is to achieve maximum performance on the given configuration. With this in mind it's possible to conclude that, as expected, the main laptop makes reaching maximum performance much easier than the cluster node. As a consequence, there are fewer kernels that are memory bound when running in the laptop than there are when running in the cluster node, which requires an arithmetic intensity of at least 19,24318 GFlops/byte to become CPU bound.

2.2.1 Ceilings

As mentioned earlier, the Roofline Model predicts the use of ceilings to limit performance according to different optimizations. This ability to predict performance gains based on each technique is useful when trying to decide which optimization to apply based on the possible gain it provides. Since different kernels have different characteristics such as the type and size of the data structures used as well as the operations performed, the following model includes

the ceilings for the main laptop configuration for the dot product kernel and can be used as an accurate predictor of which optimizations will result in the biggest increases in performance.

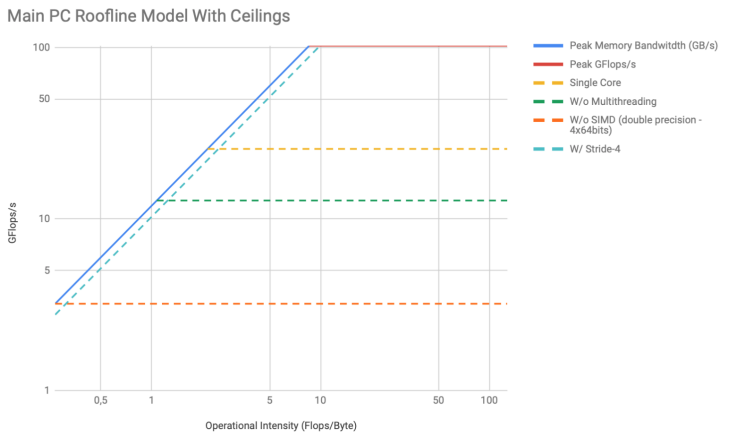

Figure 2: Main laptop roofline Model w/ ceilings

To determine the impact of each optimization on the performance ceiling it is often useful to either consult the manufacturer manual for optimization techniques or perform benchmarks that allow a comparison between optimized and non-optimized versions of a kernel. In this case, the strided access impact on memory bandwidth was obtained by modifying the STREAM benchmark to support strided access, in this case stride-4. The multi-core and multithreaded performance impacts were determined by taking the number of cores (and threads) and dividing the achievable performance by these, assuming a linear improvement in function of the number of cores(threads) used, since the Intel i5-8250U has 4 cores, this sets the ceiling at:

$$\frac{102.4}{4} = 25.6GFLOP/s$$

Using multi-threading allows for two threads to run concurrently in each core, meaning the absence of this technique further limits attainable performance to half the possible value:

$$\frac{25.6}{2} = 12.8GFLOP/s$$

As for the impact of vectorization, the width of the vector units was taken into account, in this case it supports 4 doubles, meaning the expected impact is, at most, 4 times the achievable performance, setting the ceiling at:

$$\frac{12.8}{4} = 3.2GFLOP/s$$

3 Analysis of different matrix dot product implementations

3.1 PAPI - Performance API

The Performance API(PAPI) is a useful tool for identifying bottlenecks and analyzing program execution. Due to the considerable amount of (hardware) counters made available by the API, it's paramount to identify the relevant ones on a use-case basis.

The available counters are dependent on the (hardware) platform, therefore we must first describe the hardware used for this study. This description is available in appendix in the section A.1.

To list the available counters we must first load the PAPI module(version 5.5.0) by issuing the command `module load papi/5.5.0` and, by running `papi_avail`, we obtain a comprehensive list of the commands provided by PAPI and an indicator on whether the platform supports them or not.

From this list we selected the following counters:

```
PAPI_L1_DCM - Level 1 data cache misses
PAPI_L2_DCM - Level 2 data cache misses
PAPI_L2_TCM - Level 2 cache misses
PAPI_L3_TCM - Level 3 cache misses
PAPI_L1_LDM - Level 1 load misses
PAPI_L1_STM - Level 1 store misses
PAPI_L2_STM - Level 2 store misses
PAPI_TOT_INS - Instructions completed
PAPI_FP_INS - Floating point instructions
PAPI_LD_INS - Load instructions
PAPI_SR_INS - Store instructions
PAPI_L2_DCH - Level 2 data cache hits
PAPI_L2_DCA - Level 2 data cache accesses
PAPI_L3_DCA - Level 3 data cache accesses
PAPI_L3_TCA - Level 3 total cache accesses
PAPI_L3_DCW - Level 3 data cache writes
```

The counters referring to cache data, such as misses and accesses, loads and stores, allow us to determine how much the application is taking advantage of the memory hierarchy and whether it's possible to increase memory locality to reduce the amount of misses or, in case it isn't, whether it's advantageous to use multiple threads to hide memory latency.

The counter for the number of instructions completed is necessary to calculate the execution time of an application. The floating point instruction counter will be useful to calculate floating point performance and, when used in conjunction with memory access counters, can be used to determine the arithmetic intensity(Flops/byte) of the dot product kernel.

3.2 Data Structures for a r662 node

To implement the dot product algorithm we need three NxN matrices of floats which sets the required size to $3 * N * N * 4$ *bytes*, (where 4 is the size of a float, in bytes). Solving this equation for each cache level size gives us the amount of elements that completely fill each one. The cache sizes for the r662 node can be found in the section A.1:

- Fits in L1 cache: $N = 52$
- Fits in L2 cache: $N = 128$
- Fits in L3 cache: $N = 1024$
- Only fits in main memory: (at least 1619) $N = 2048$

The equations for each cache level can be checked in the Appendix D.1. One thing to note is the choice of powers of 2 for the number of elements per row/column which makes it easier to implement certain optimization techniques such as vectorization by allowing aligned accesses.

3.3 Memory metrics

Memory bandwidth is one of the biggest bottlenecks in modern computing with CPU speeds greatly surpassing those of main memory as such, without the use of memory latency hiding techniques, the CPU would spend most of the time waiting for data to be fetched from memory. Consequently it's of paramount importance to find ways to limit the amount of times main memory is accessed favoring the smaller but faster memories that are closer to the CPU and thus present reduced latency in comparison to bigger memories. This can be achieved through the use of several techniques that affect different metrics which we'll identify in the current section.

3.3.1 Theoretical methods

The best execution time for each dot product corresponds to the smallest matrix size **i.e.** N=52, when the matrix fits completely in L1 cache. As such the number of RAM accesses per instruction as well as the amount of bytes transferred to/from RAM can be estimated based on this assumptions, as a function of the matrix size:

- Number of RAM accesses per instruction $\approx 1/32$[Eq: 10]
- Number of bytes transferred to RAM $\approx N * N * N * 4 \Leftrightarrow 562432(N = 52)$ [Eq: 8]
- Number of bytes transferred from RAM $\approx N * N * N * 4 * 3 \Leftrightarrow 1687296(N = 52)$ [Eq: 9]

3.3.2 With PAPI counters

Being approximations it's useful to confirm the values with real data which can be obtained via PAPI. Using that data we can develop, for each counter, an expression which calculates the value:

- Number of RAM accesses per instruction: `PAPI_TOT_INS, PAPI_L3_TCM` [Eq: 13]
- Number of bytes transferred to RAM: `PAPI_SR_INS`[Eq: 11]
- Number of bytes transferred from RAM: `PAPI_LD_INS` [Eq: 12]

3.3.3 Miss rate on reads - Kernel j-k-i

As previously said, one of the major bottlenecks in computer performance is main memory. When a `load` instruction is executed, neglecting operand pre-fetch techniques, the address being loaded is searched for in the higher levels of memory hierarchy and, when it isn't found it occurs what is known as a miss, a read miss. This miss causes the CPU to search in the next level of the memory hierarchy which is slower and incurs the CPU in wasted memory cycles waiting for the operand to be loaded. As such read miss rate is an extremely important metric in examining the behaviour of an application and can be calculated by resorting to PAPI counters for each cache level(see Appendix I).

3.4 Kernel Plotting in Roofline model

The number of floating point operations performed in any given kernel is given by the following expression: $(1+1)*N^3$ where $(1+1)$ corresponds to the multiplication and addition operations performed in each inner loop iteration and N^3 to the total number of iterations of that loop. Thus, for each data set size, the number of floating point operations performed is:

- **N=52** : 281216 Flops
- **N=128** : 4194304 Flops
- **N=1024** : 2147483648 Flops
- **N=2048** : 17179869184 Flops

Using the Roofline models previously elaborated we're now able to plot the achieved performance (see H) which, as expected, is well below the theoretical limit for the cluster node 662. This is caused by the lack of any optimizations such as improved memory locality, vectorization or multiprocessing.

3.5 Performance limiting factors - Memory vs CPU

Analyzing the results of the achieved performance (see Appendix H) it's obvious that one of the limiting factors to performance is memory bandwidth, which can be improved by taking advantage of the caches closer to the CPU.

Another key takeaway from the values obtained (see Appendix E) is that memory access patterns are of paramount importance and ought to be considered for applications concerned with performance. In this case, interchanging the loop order (**i-j-k**, **i-k-j**, **j-k-i**) greatly impacts performance based on whether matrices are accessed in a row-major or column-major mode. The results show that the **i-k-j** obtained the best results by exhibiting better spacial locality than **j-k-i** and **i-j-k** and better temporal locality, therefore taking advantage of cache and memory hierarchy. Furthermore, the **j-k-i** version of the dot product clearly benefits from matrix transposition which, by rearranging the way each matrix is stored in memory, allows for better spacial and temporal locality.

3.6 Optimization Techniques

3.6.1 Block dot product

The block optimization in dot product increases the spacial locality of columns and rows, decreasing the number of replacements of cache lines and improving the execution time of the dot product. Without blocks, if the rows are large enough to not fit in a single cache line, there will be a high number of collisions when loading values and, thus repeated loads and replacements of cache lines. The use of blocks prevents this by limiting the size of each block so that each row fits in a single cache line, reducing the number of collisions making the amount of misses come closer to that the amount of cold misses.

3.6.2 Vectorization

Vectorization is another technique whose main objective is to hide memory latency and improve parallelism at the data level. It works by loading a vector from memory and operating on the elements of that vector simultaneously. Modern compilers are already able to identify vectorizable portions of code which normally only involve loops that perform arithmetic calculations on vector elements with no dependencies across iterations. Thus, the current implementation of the dot product with block optimization, represents an almost perfect candidate for being vectorized by the compiler. One of the issues that needs to be solved prior is the alignment of vector memory addresses which should be 16 byte aligned. This can be achieved by declaring the pointers with the `aligned` modifier, as such:

```
float __attribute__((aligned(32))) **a = ...
float __attribute__((aligned(32))) **b = ...
float __attribute__((aligned(32))) **c = ...
```

Another obstacle to auto-vectorization by the compiler is pointer aliasing, in this case:

```
void dotProductBlockOptimized(float **c,
                              float **a,
                              float **b,
                              int n){
    ...
    c[i][j] += a[i][k] * b[k][j];
```

the compiler won't vectorize the code because pointers `a,b` and `c` may point to the same memory location. To prevent this from happening the `__restrict__` keyword indicates that the pointer to which it is applied is the only alias to that memory location:

```
void dotProductBlockOptimized(float ** __restrict__ c,
                              float ** __restrict__ a,
                              float ** __restrict__ b,
                              int n)
```

Besides increasing data level parallelism and hiding memory latency, vectorization also reduces the amount instructions needed to execute a given kernel since each vector instruction performs the equivalent of 8 scalar instructions. Observing the results of applying this technique it's visible that the obtained speed-up ($\approx$ 5.66x) (see 10) is lower than the expected speed-up ($\approx$ 8x). This is because loading a vector from memory demands more bandwidth than loading a single value, putting more load on memory (bandwidth) and lowering achieved performance.

3.6.3 Multicore processing

Multicore processing is another form of parallelism that allows for the work load to be distributed among cores, reducing the time it takes to executed a parallel region of code significantly. The use of APIs such as OpenMP make adapting codes with high parallel potential trivial, though care has to be taken for shared variables and memory dependencies. Applying this technique to the dot product kernel involved adding a `pragma` directive before the inner-most loop to parallelize that region. For the tests 24 threads were used which would result, theoretically, in a 24x speed-up however the results show a larger speed-up due to the fact that this optimization technique was combined with vectorization, yielding a 37.58x speed-up (see 11). Yet again, this is far from the expected speed-up when we take into account vectorization, which would be $8 * 24 = 192$x which can be justified by the fact that we now have 24 threads sharing the last level cache as well as the memory bandwidth, hence why the speed-up is lower than expected.

3.6.4 Kepler GPU

The amount of floating-point units in GPUs and the easy implementation made possible by platforms like CUDA have made this devices one of the tools for high performance computing. The use of high amounts of floating point units allows memory latency to be hidden by switching between thread groups, known as warps, when data isn't yet available. GPUs follow "Same Instruction Multiple Threads" model which means that the threads in a warp will execute the same instruction on different operands. On the other hand, many of the control mechanisms involved in this process are implemented in hardware making it transparent to the developer. The implementation of the dot product algorithm on a GPU using CUDA consisted on adapting the current implementation to the CUDA model where each (CUDA) thread processes only one element of the data set:

```
*(dev_result+blockIdx.x*N+threadIdx.x)=0;

for(unsigned i=0; i < N; i++)
        *(dev_result+blockIdx.x*N+threadIdx.x) += *(dev_m1+blockIdx.x*N+i) * *(dev_m2+i*N+threadIdx.x);
```

As expected there was a visible increase in performance(see L and E) with most of execution time of the GPU implementation being attributed to the communication between the host(CPU) and the device(GPU) which accounted for almost 99% of the total execution time.

3.6.5 KNL - Coprocessor

Coprocessors such as the KNL architecture are an alternative to GPUs when it comes to executing highly parallel kernels. Known as manycore processors this devices are similar to GPUs in that they're focused on being efficient for executing highly parallel code, with manycores requiring that this parallelism be explicit. A distinct advantage of the KNL coprocessor over a CUDA GPU is the fact that applications developed for running on multicore processors are meant to be portable/compatible with manycores which isn't the case for GPUs. Another advantage of manycores over GPUs is the ability to run sequential code in a more efficient manner than the latter, making them more suited for irregular workloads [1]. The results obtained with the KNL Coprocessor present a considerable speed-up in comparison to the base version due to the exploitation of the high parallelism available in the dot product but fall short when compared to the Kepler GPU (see 13).

4 Conclusion

The use of matrices in scientific applications is frequent, as such the results obtained not only offer valuable insights on how to develop efficient code while also remaining useful for other application types that use similar data structures. Furthermore, the results show that the use of kernels that access memory in an optimal way, such as the i-k-j kernel, present the best performance. Additionally, although the communication between the GPU and the CPU is a major bottleneck for performance, the use of a GPU still presented better results than the use of a manycore when executing the dot product kernel (see 14).

References

[1] George C Caragea et al. "General-purpose vs. gpu: Comparison of many-cores on irregular workloads". In: (June 2010).

[2] *FLOPS*. `https://en.wikipedia.org/wiki/FLOPS`. Access in 20/12/2018.

[3] *How to calculate peak theoretical performance of a CPU-based HPC system*. `http://www.novatte.com/our-blog/197-how-to-calculate-peak-theoretical-performance-of-a-cpu-based-hpc-system`. Access in 20/12/2018.

[4] *Intel Xeon Phi 7210 specifications*. `http://www.cpu-world.com/CPUs/Xeon_Phi/Intel-Xeon%20Phi%207210.html`. Access in 02/01/2019.

[5] John McCalpin. "Memory bandwidth and machine balance in high performance computers". In: *IEEE Technical Committee on Computer Architecture Newsletter* (Dec. 1995), pp. 19–25.

[6] NVIDIA. *Kepler GK110/210 v1.0*. URL: https://computing.llnl.gov/tutorials/linux_clusters/gpu/Kepler-GK110-GK210-ArchitectureWhitepaper.pdf.

[7] NVIDIA. *TESLA K20 GPU ACCELERATOR*. URL: https://computing.llnl.gov/tutorials/linux_clusters/gpu/Tesla-K20-Board-Specs.pdf. 2013.

[8] *Using LC's GPU Clusters - Quick Guide*. https://computing.llnl.gov/tutorials/linux_clusters/gpu/. Access in 02/01/2019.

[9] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: http://doi.acm.org/10.1145/1498765.1498785.

Appendix A Full characterization of the hardware platform

A.1 SeARCH Cluster node 662

- (Dual) CPU:
  - Manufacturer: Intel
  - Model: Xeon E5-2695 v2
  - Number of cores: 12
  - Peak FP performance: 460.8 GFlops/s [2, 3]
- Main Memory:
  - Size: 64 GB
- Memory Hierarchy:
  - Cache sizes and associativity:
    - L1d (per core): 32 KiB, 8-way
    - L1i (per core): 32 KiB, 8-way
    - L2 (per core): 256 KiB, 8-way
    - L3 (per package): 30 MiB, 20-way
  - Memory access bandwidth: 23.94614 GB/s
- Extras:
  - GPU: 2x Nvidia K20m [8, 7, 6]
    - Cuda Cores: 2496 (each one with FP and Integer unit)
    - SMX: 13
      - Threads / Warp: 32
      - Max Threads / Thread Block: 1024
      - Max Warps / Multiprocessor: 64
      - Max Threads / Multiprocessor: 2048
      - Max Thread Blocks / Multiprocessor: 16
    - Cuda Cores per SMX: 192
    - Double Precision Units: 64 (divided in 4 "lanes" of 16 units)
    - Local Memory: 65536 x 32-bit registers
    - Shared Memory: 64 KiB divided between shared memory and L1 cache, configurable; 48 KiB Read-Only Data Cache
    - Global Memory: 5GB (GDDR5)
    - GPU Clock: 706 MHz
    - Memory Clock: 2.6 GHz
  - Coprocessor: Intel(R) Xeon Phi(TM) CPU 7210 [4]
    - Frequency: 1.3 GHz
    - Cores: 64
    - Threads: $64 * 4 = 256$
    - Cache:
      - L1 Data (per core): 32 KiB
      - L1 Instructions (per core): 32 KiB
      - L2: $32 * 1024KB = 32MB$
    - On Package Memory: 16 GB with 8 channels

A.2 Main laptop - hardware specifications

- Manufacturer: Asus
- Model: VivoBook S14 S410UN
- CPU:
  - Manufacturer: Intel
  - Model: i5-8250U 1.60GHz
  - Reference: 124967
  - Number of cores: 4 cores, 8 threads (Hyper-Threading)
  - Peak FP performance: 102.4 GFlops/s [2, 3]
- Main Memory:
  - Memory Frequency: 2667 Mhz (Tcc = 0.4 ns)
  - CL (Column access strobe Latency): 15
  - Latency: 6 ns (= 15 * 0.4 ns)
  - Size: 8 GB
- Memory Hierarchy:
  - Cache sizes and associativity:
    - L1d (per core): 32 KiB, 8-way
    - L1i (per core): 32 KiB, 8-way
    - L2 (per core): 256 KiB, 4-way
    - L3 (per package): 6144 KiB, 12-way
  - Memory access bandwidth: 9112.2 MB/s

A.3 STREAM benchmark

```
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 1048576 (elements), Offset = 0 (elements)
Memory per array = 8.0 MiB (= 0.0 GiB).
Total memory required = 24.0 MiB (= 0.0 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Number of Threads requested = 8
Number of Threads counted = 8
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 1463 microseconds.
   (= 1463 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           9089.2     0.002594     0.001846     0.007676
Scale:          8666.1     0.002213     0.001936     0.002747
Add:            9269.6     0.002824     0.002715     0.002953
Triad:          9208.2     0.002806     0.002733     0.002899
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
```

Figure 3: STREAM benchmark output for the main laptop

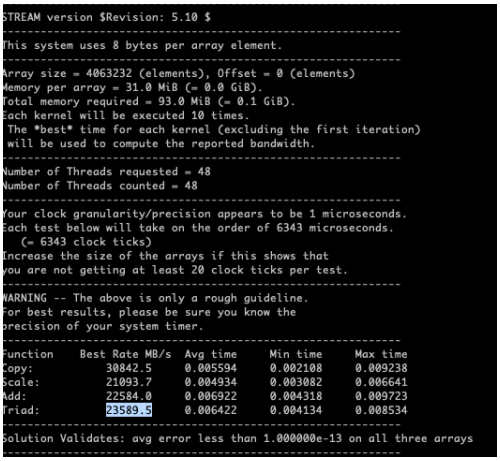

Figure 4: STREAM benchmark output for 662 Cluster node

Appendix B Available PAPI Counters - Node 662

Name	Deriv	Description (Note)
PAPI_L1_DCM	No	Level 1 data cache misses
PAPI_L1_ICM	No	Level 1 instruction cache misses
PAPI_L2_DCM	Yes	Level 2 data cache misses
PAPI_L2_ICM	No	Level 2 instruction cache misses
PAPI_L1_TCM	Yes	Level 1 cache misses
PAPI_L2_TCM	No	Level 2 cache misses
PAPI_L3_TCM	No	Level 3 cache misses
PAPI_TLB_DM	Yes	Data translation lookaside buffer misses
PAPI_TLB_IM	No	Instruction translation lookaside buffer misses
PAPI_L1_LDM	No	Level 1 load misses
PAPI_L1_STM	No	Level 1 store misses
PAPI_L2_STM	No	Level 2 store misses
PAPI_STL_ICY	No	Cycles with no instruction issue
PAPI_BR_UCN	Yes	Unconditional branch instructions
PAPI_BR_CN	No	Conditional branch instructions
PAPI_BR_TKN	Yes	Conditional branch instructions taken
PAPI_BR_NTK	No	Conditional branch instructions not taken
PAPI_BR_MSP	No	Conditional branch instructions mispredicted
PAPI_BR_PRC	Yes	Conditional branch instructions correctly predicted
PAPI_TOT_INS	No	Instructions completed
PAPI_FP_INS	Yes	Floating point instructions
PAPI_LD_INS	No	Load instructions
PAPI_SR_INS	No	Store instructions
PAPI_BR_INS	No	Branch instructions
PAPI_TOT_CYC	No	Total cycles
PAPI_L2_DCH	Yes	Level 2 data cache hits
PAPI_L2_DCA	No	Level 2 data cache accesses
PAPI_L3_DCA	Yes	Level 3 data cache accesses
PAPI_L2_DCR	No	Level 2 data cache reads
PAPI_L3_DCR	No	Level 3 data cache reads
PAPI_L2_DCW	No	Level 2 data cache writes
PAPI_L3_DCW	No	Level 3 data cache writes
PAPI_L2_ICH	No	Level 2 instruction cache hits
PAPI_L2_ICA	No	Level 2 instruction cache accesses
PAPI_L3_ICA	No	Level 3 instruction cache accesses
PAPI_L2_ICR	No	Level 2 instruction cache reads
PAPI_L3_ICR	No	Level 3 instruction cache reads
PAPI_L2_TCA	Yes	Level 2 total cache accesses
PAPI_L3_TCA	No	Level 3 total cache accesses
PAPI_L2_TCR	Yes	Level 2 total cache reads
PAPI_L3_TCR	Yes	Level 3 total cache reads
PAPI_L2_TCW	No	Level 2 total cache writes
PAPI_L3_TCW	No	Level 3 total cache writes

```
PAPI_FDV_INS  No   Floating point divide instructions
PAPI_FP_OPS   Yes  Floating point operations
PAPI_SP_OPS   Yes  Floating point operations; optimized to count scaled single precision vector operations
PAPI_DP_OPS   Yes  Floating point operations; optimized to count scaled double precision vector operations
PAPI_VEC_SP   Yes  Single precision vector/SIMD instructions
PAPI_VEC_DP   Yes  Double precision vector/SIMD instructions
PAPI_REF_CYC  No   Reference clock cycles
```

Appendix C Peak floating point performance formulas

C.1 Cluster 662 Node

$$2.4(frequency) * 12(cores) * 8(DFlops\_per\_cycle) * 2(cpu's\_per\_node)) = 460.8GFlops/s \quad (1)$$

C.2 Main laptop

$$1.6(frequency) * 4(cores) * 16(DFlops\_per\_cycle) * 1(cpu's\_per\_node)) = 102.4GFlops/s \quad (2)$$

Appendix D Matrix size formulas

D.1 General formula

$$no_{matrices} * matrix\_order * float\_size = memory\_size \quad (3)$$

D.2 Matrix fitting in L1 cache

$$3 * N * N * 4 = 32 * 1024 \Leftrightarrow N \approx 52 \quad (4)$$

D.3 Matrix fitting in L2 cache

$$3 * N * N * 4 = 256 * 1024 \Leftrightarrow N \approx 147 \quad (5)$$

D.4 Matrix fitting in L3 cache

$$3 * N * N * 4 = 30 * 1024^2 \Leftrightarrow N \approx 1619 \quad (6)$$

D.5 Matrix fitting in L3 cache for Multicore

$$3 * N * N * 4 = \frac{2 * 30 * 1024^2}{24} \Leftrightarrow N \approx 467 \quad (7)$$

Appendix E Non optimized dot product measurements

	i-j-k	i-k-j	j-k-i	j-k-i (transposed)
Matrix Order	K-Best(K=3)(us)	K-Best(K=3)(us)	K-Best(K=3)(us)	K-Best(K=3)(us)
52	199,00±9,95	174.00±8.70	239.67±11.98	207.67±10.38
128	2823,33±141,17	2616±130.8	3296.33±164.82	2956.33±147.82
1024	8297943,33±414897,17	1346980.333±67349.02	10388926±519446.30	10686387.67±534319.38
2048	49543536,67±2477176,83	10789776.33±539488.82	72615491,67±3630774,58	34268445,67±1713422,28

n.b.results are shown with 5% tolerance

Appendix F RAM Access - Theoretical formulas

F.1 Bytes transferred to RAM

$$N * N * N * 4 \tag{8}$$

with N=matrix order and 4=float size in bytes

F.2 Bytes transferred from RAM

$$N * N * N * 4 * 3 \tag{9}$$

with N=matrix order, 4=float size in bytes and 3=no. of matrices

F.3 RAM Accesses per Instruction

$$\frac{\frac{3*N^3}{16}}{6 * N^3} = \frac{1}{32} \tag{10}$$

Appendix G RAM Access - PAPI formulas

G.1 Bytes transferred to RAM

$$PAPI\_SR\_INS * 4 \tag{11}$$

with 4=float size in bytes

G.2 Bytes transferred from RAM

$$PAPI\_LD\_INS * 4 \tag{12}$$

with 4=float size in bytes

G.3 RAM Accesses per Instruction

$$\frac{PAPI\_L3\_TCM}{PAPI\_TOT\_INS} \tag{13}$$

Appendix H Achieved Performance Plot in Roofline Models

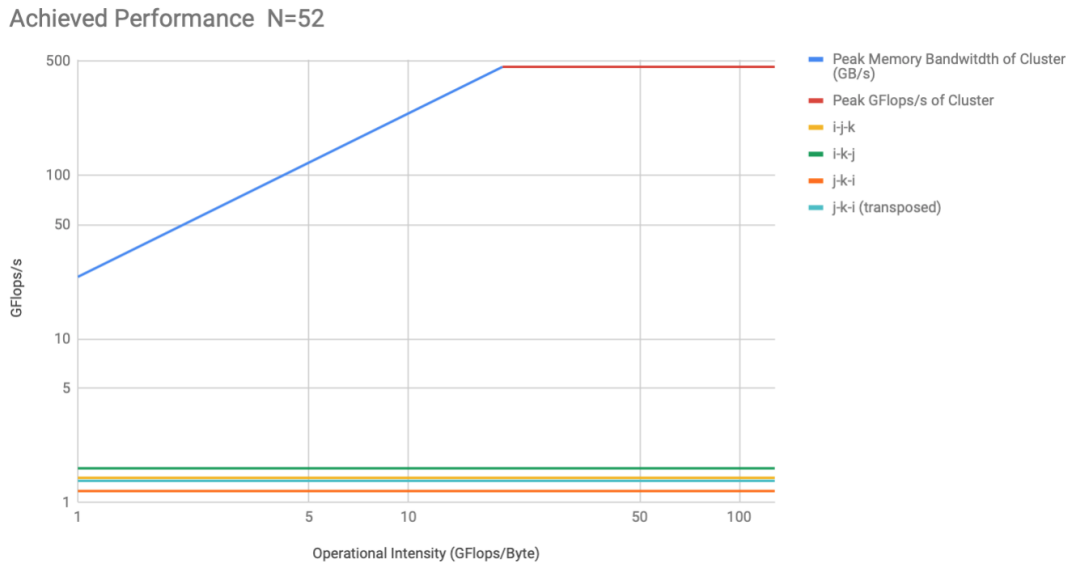

Figure 5: Achieved Performance for N=52

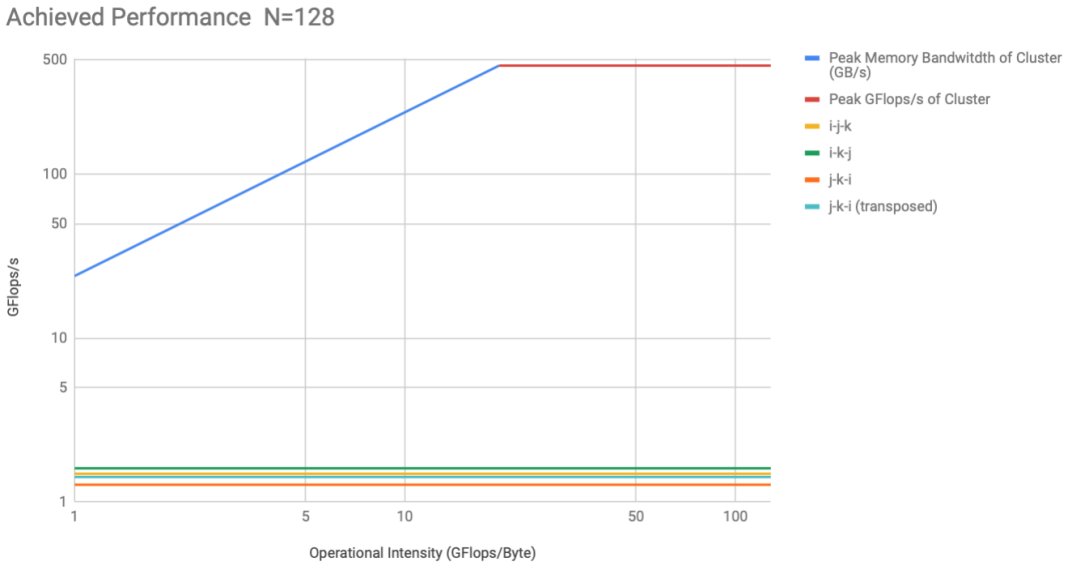

Figure 6: Achieved Performance for N=128

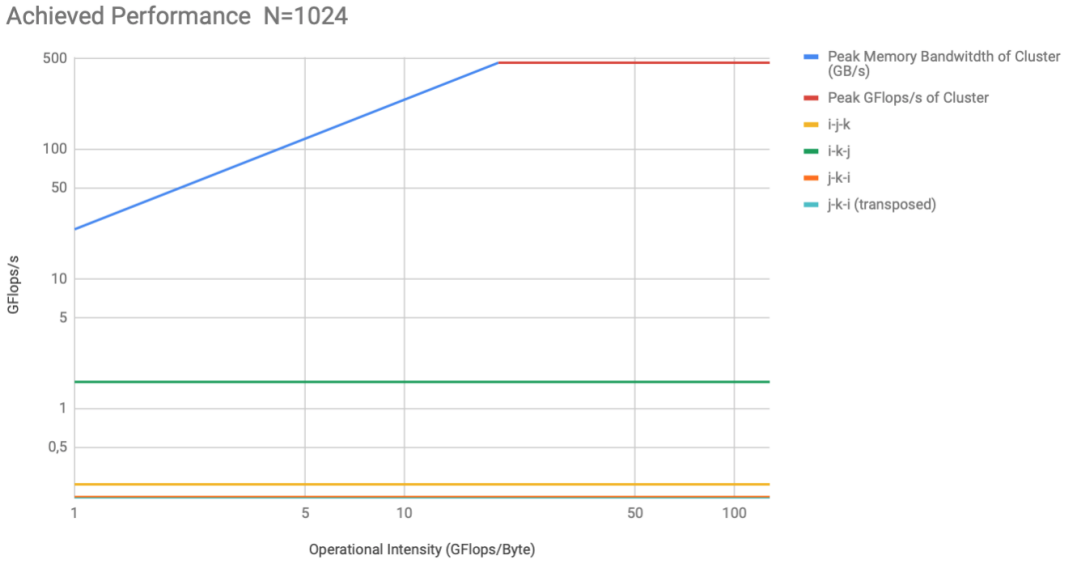

Figure 7: Achieved Performance for N=1024

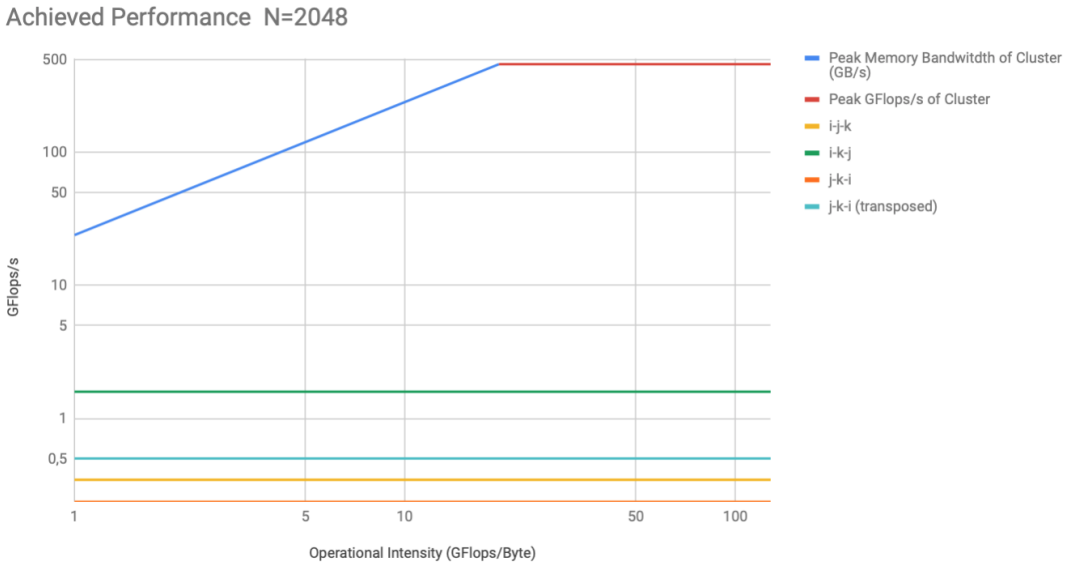

Figure 8: Achieved Performance for N=2048

Appendix I Miss rate on memory reads

I.1 L1

$$\frac{L1\_LDM}{LD\_INS} \tag{14}$$

I.2 L2

$$\frac{L2\_DCM - L2\_STM}{L1\_LDM} \tag{15}$$

I.3 L3

$$\frac{L3\_TCM}{L2\_DCM} \tag{16}$$

Appendix J Block Optimization + Vectorization

	i-k-j
Matrix Order	K-Best(K=3) and Tolerance (5%) (us)
52	31,00±1,55
128	458,33±22,92

Appendix K Block Optimization + Vectorization + Multi core

	i-k-j
Matrix Order	K-Best(K=3) and Tolerance (5%) (us)
2048	287100,33±14355,02

Appendix L K20m

	i-k-j	
Matrix Order	K-Best(K=3) and Tolerance (5%) (us)	data transfer times (us)
2048	30583,00±1529,15	30563,00

Appendix M KNL with Block Optimization + Vectorization + Multi core

	i-k-j
Matrix Order	K-Best(K=3) and Tolerance (5%) (us)
2048	239057,33±11952,87

Appendix N Baseline vs. Blocking+Vectorization

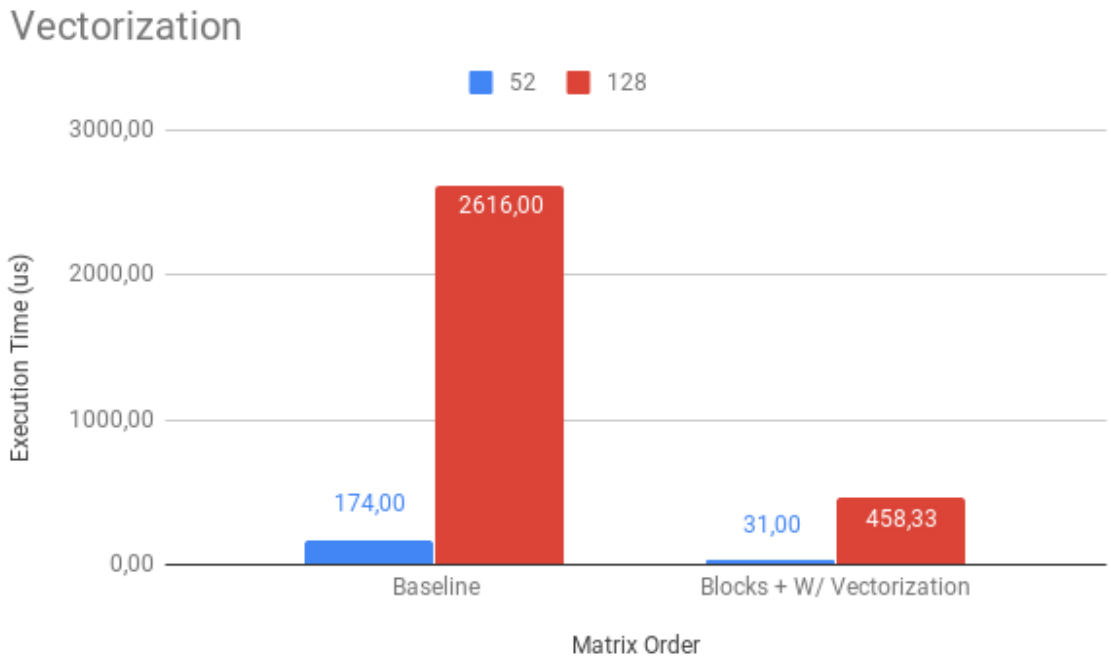

Figure 9: Execution Time with 4 blocks size

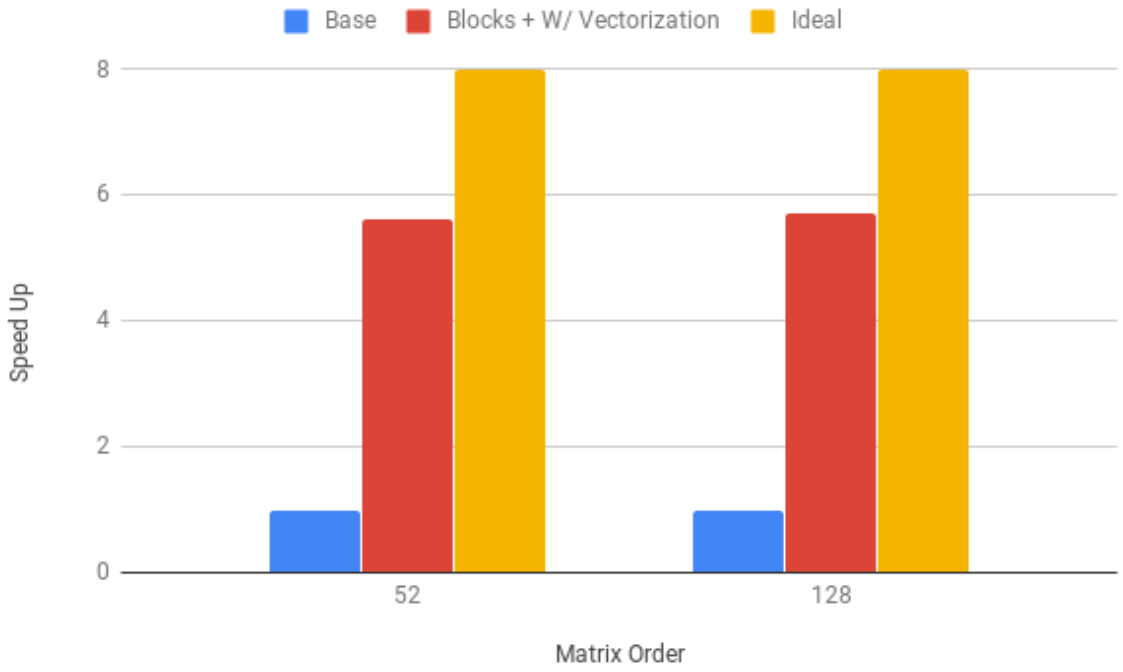

Figure 10: Speed Up

Appendix O Baseline vs. Blocking+Vectorization+Multicore

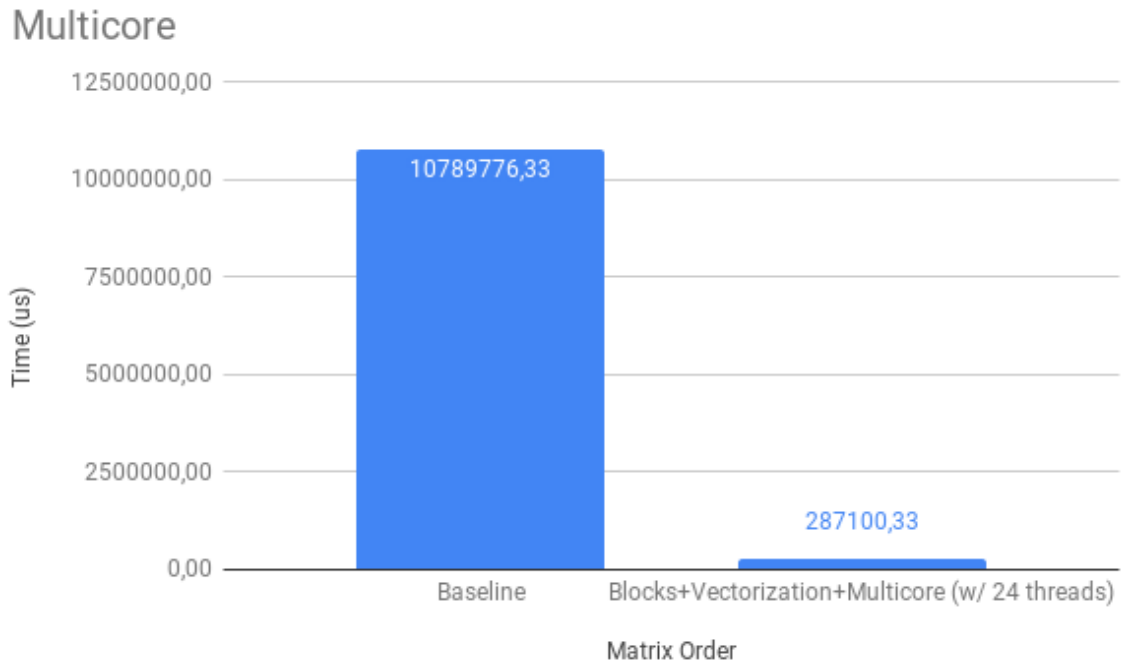

Figure 11: Execution Time with 4 blocks size

Appendix P Baseline vs. Accelerators

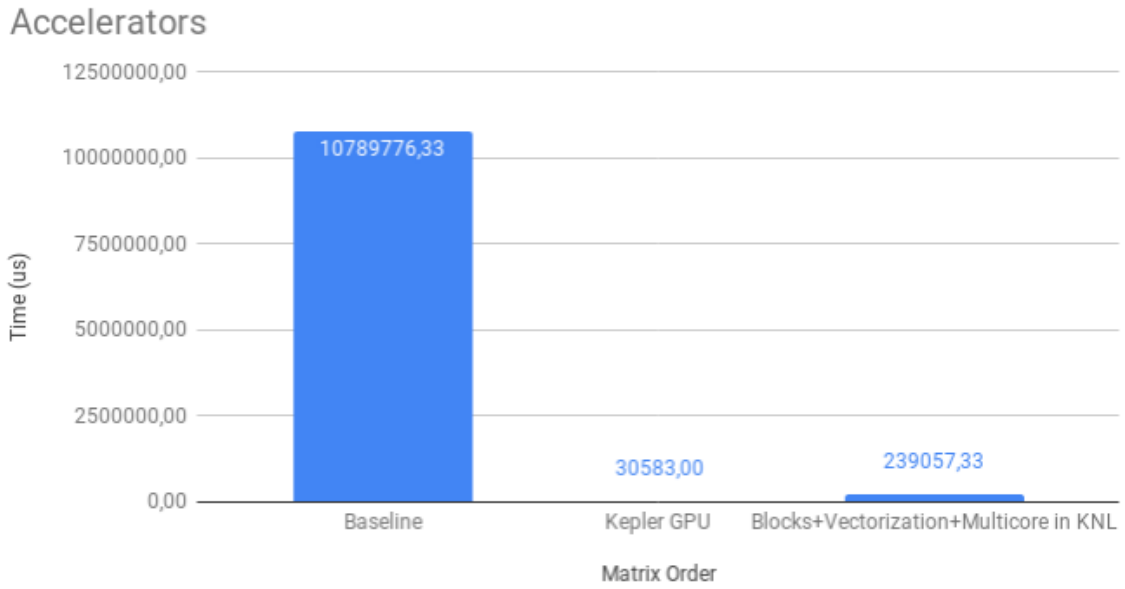

Figure 12: Execution Time with 4 blocks size (Only for KNL) for N=2048

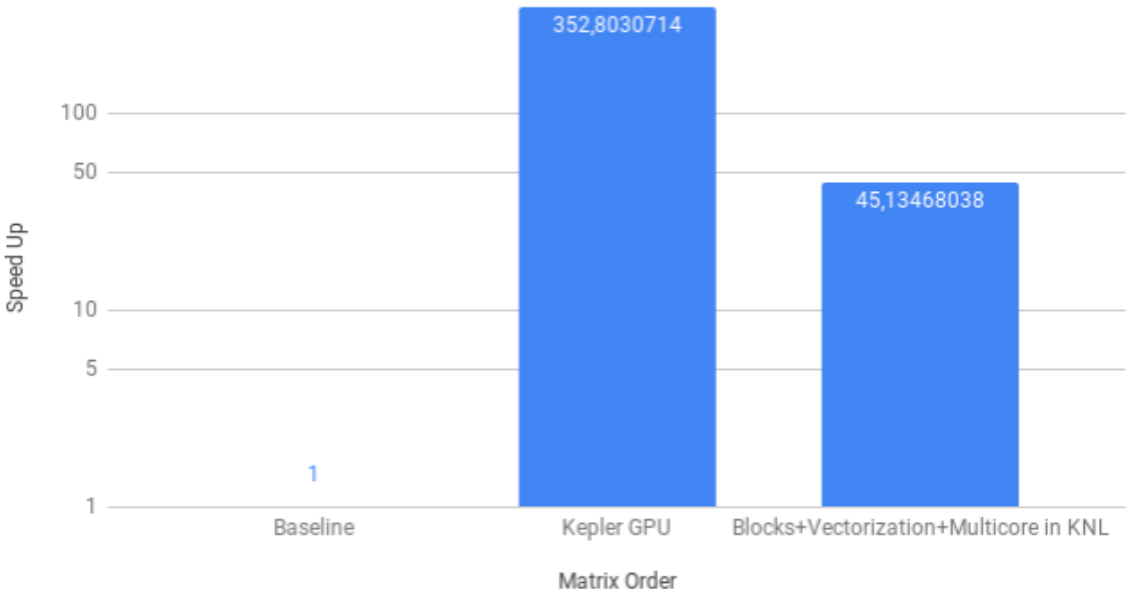

Figure 13: Speed Up

Appendix Q General Speed Up

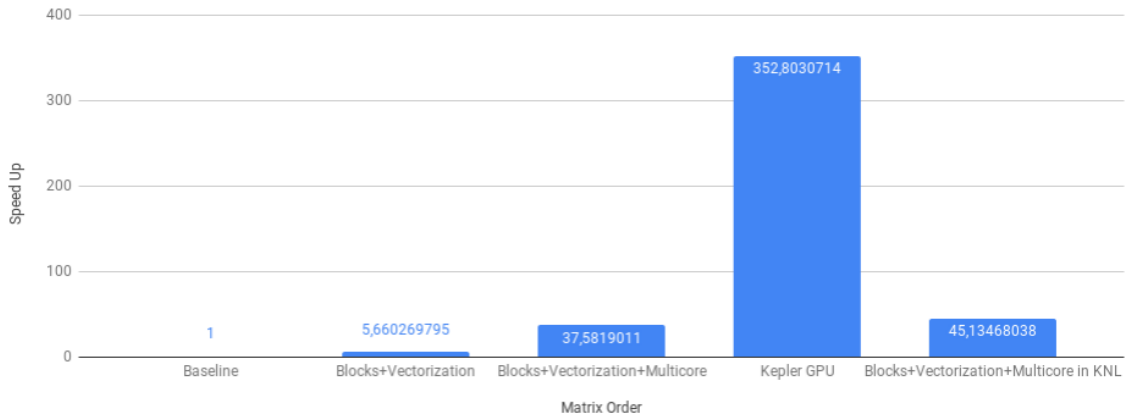

Figure 14: Speed Up