Characterization of the performance bottlenecks, code profiling and performance analysis in different computing platforms

Luís Neto(a77763) and Gonçalo Raposo(a77211)

Abstract— **The present paper objective is to analyse how a given kernel can be tuned to explore and to take advantage of the features that a computing platform offers. The kernel used is matrix multiplication. Several optimizations and insights are presented about it. In the last sections some implementations and time measurements of the kernel for a many-core server, Intel Knights Landing, and an accelerator, NVIDIA Kepler are presented.**

I. INTRODUCTION

In this paper, various matrix multiplication implementations will be discussed and compared. First of all, a Roofline model was built for each machine and the respective ceilings for the team's laptop were added, based on the hardware characteristics.

The Roofline model is a widely accepted model that offers performance guidelines and insights about potential bottlenecks, by presenting information about the limit of computation and bandwidth.

Subsequently, the various implementations of the matrix multiplication algorithm in squared matrices will be analysed, always taking account the matrices size.

Moreover, the results presented in the next sections are justified by considering the computing platforms architecture, using the lower level Performance API, that provides us more specific insights about the algorithm and memory behaviour.

Finally, it will also be implemented and compared an optimised matrix multiplication algorithm for Intel Knights Landing and GPU Kepler.

II. HARDWARE CHARACTERISATION

The computing platforms used were the team laptop and the node 662 present at SeARCH Cluster. The former platform is a MacBook Pro, Early 2015, and the latter is Intel Xeon E5-2695V2. The following table will cover 3 major topics of hardware characteristics: the main memory, CPU device, and memory hierarchy.

The information in the table comes from different sources. From the Intel official website, a more general information such as the number of cores, architecture and processors model was retrieved.

The specifications that involve the memory hierarchy like n-way set associativity and size of each level of cache memory came from *cpuworld.com* website.

Manufacturer	Intel Corporation
Arquitecture	Broadwell
Model	Core i5-5257U
# Cores	2
# Threads	4
Processor Frequecy	2.7 GHz
Extensions	Intel SSE4.1, Intel SSE4.2, Intel AVX2
FP Performance Peak	86.4 GFlops
L1 Cache	64 KiB
L2 Cache	512 KiB
L3 Cache	3 MiB
Associativity	8/8/12 way set associative
RAM Memory	2 * 4GiB DDR3
RAM Latency	13.77 ns
Memory Bandwidth Peak	29.872 GiB/s
#Memory Channels	2

TABLE I
TEAM'S LAPTOP HARDWARE CHARACTERISATION

The hardware characterisation table for Intel Xeon E5-2695V2 is presented in the Appendix.

III. ROOFLINE MODEL

The Roofline model is an easy-to-understand 2-D graph that relates floating-point performance, memory performance, and arithmetic intensity. This relation offers insights about the system upper-bounds and how the kernel could reach the computing platform's peak of performance.

To build this model, the STREAM benchmark was used to estimate the maximum bandwidth.

The team's laptop is equipped with DDR3 RAM, two memory channels, 64-bit memory bus, a frequency of 933,5 MHz. The maximum bandwidth obtained with the benchmark was 18 GiB/s. The Node-662 has DDR3 RAM, four memory channels, 64-bit memory bus, a frequency of 800 MHz. The maximum bandwidth obtained with benchmark was 51,2 GiB/s. To calculate the attainable GFlops/sec and the Floating Point Peak Performance the following formulas were used:

$$\text{Attainable GFlops} = \text{Min(Peak FPP,Peak Memory Bandwidth x Operational Intensity)}$$

$$\text{Peak FPP} = \text{Nr. of CPU} * \text{Nr. of Cores} * \text{Clock Frequency} * \text{FMA} * \text{CPI} * \text{SIMD}$$

Since the team's laptop features an AVX2 implementation for SIMD with a width of 256 bits, FMA3 and 2 cores the Floating Point Peak Performance value is 86.4 GFlops.

Node 662 also features an AVX implementation for SIMD with a width of 256 bits, 2 CPU's each with 12 cores, however, it doesn't have FMA instructions, therefore, the corresponding peak is 460.8 GFlops.

Having all the values the corresponding Roofline model for each platform was built.

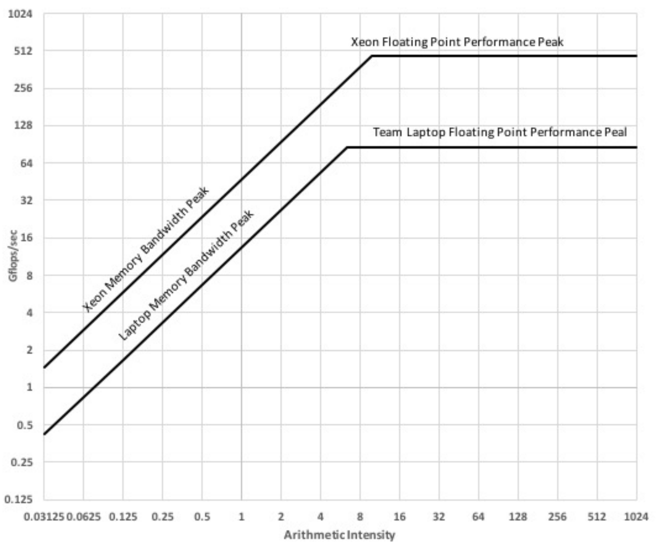

Fig. 1. Laptop and XEON Roofline model

With the analysis of the graph above, is possible to conclude that Node 662 reaches a peak of performance 5.3 times higher than the team's laptop. This result is expected since the computing platform has 24 cores.

A. *Roofline ceilings*

To have a Roofline that provides useful information about the platform's bottlenecks, the ceilings must be added the model. Notice that the lower ceilings optimizations are most likely to be achieved by a compiler or programmer than higher ceilings.[1].

1) Computational Ceilings: The first ceiling represents the usage of instruction level parallelism, this means that in each clock cycle one instruction is complete, thus, the value of this ceiling is equal to the clock frequency.

The next ceiling represents the usage of FMA instructions. Each FMA instruction can perform one multiplication and one addition in each clock cycle, thus, this ceiling is placed 2 times higher than the last one.

The next ceiling represents the usage of AVX2 implementation in the team's laptop. Since the implementation offer instructions with a width of 256 bits it can execute 8 floating point operations each cycle, therefore, this ceiling is placed 8 times higher than the last one.

Finally, the last computational ceiling, represents the usage of all the cores available in the machine, thus, this ceiling is placed 2 times higher than the last one.

2) Memory Ceilings: One of the major optimizations inside a DRAM chip is to transfer data on both rising and falling edges of the clock signal. When a DRAM chip has this optimisation it is called a double data rate DRAM chip. The team's laptop posses this kind of technology, so it is excepted a 2 times increase of maximum bandwidth.

By adding memory interfaces the data transfer between the main memory and the memory controller can be increased. The team's laptop posses a Dual-channel memory architecture, therefore,it is excepted a 2 times increase of maximum bandwidth.

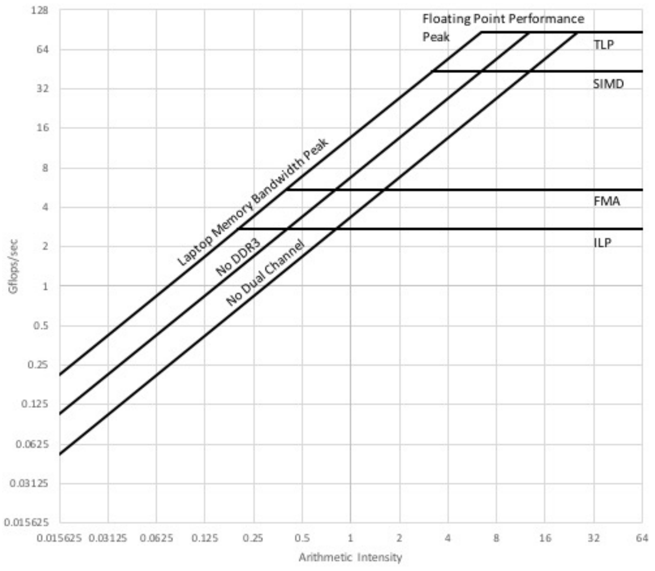

Fig. 2. Roofline model with Computational and Memory Ceilings for the team's laptop

IV. Performance Analysis Metrics

To have more information about the memory and algorithm behaviour the PAPI (v5.5.0) was used. This API permits the monitorization of a wide variety of hardware counters. To get relevant and useful information, a filter was applied to the counters available in node 662.

To list all the counters papi_avail was used and the following table describes which counters were used:

Counter	Description
PAPI_L1_DCM	Level 1 data cache misses
PAPI_L2_TCM	Level 2 cache misses
PAPI_L3_TCM	Level 3 cache misses
PAPI_LD_INS	Load instructions
PAPI_TOT_INS	Instructions completed
PAPI_FP_OPS	Floating point instructions

Using these native and derived counters useful metrics can be obtained, such as, cache levels miss rates, RAM accesses per instruction, the number of bytes transferred to/from the RAM.

V. Matrix Multiplication Algorithm Implementation Analysis

This algorithm consists in computing the product between two squared matrices, A and B, and store the result in a matrix C, where each line and column have N elements. The next sections will present 3 different implementations.

A. IJK Order Implementation

In this implementation,A and C are accessed row-wise taking advantage of spatial locality, on the other hand, matrix B is accessed column-wise, with a stride of N, which will increase significantly the cache misses.

To reduce the amount of cache misses, matrix B is preprocessed using the transpose function. By doing so, all matrices can be accessed row-wise, with a unit stride.

B. IKJ Order Implementation

In this implementation, all matrices are accessed row-wise, with a unit stride exploiting spatial locality, therefore, a low number of cache misses are expected

C. JKI Order Implementation

For this index order, all matrices are accessed column-wise and with a stride of N, a very high negative impact in performance is expected.

To optimise this implementation, initially A and B are preprocessed, the multiplication is computed and store in C, finally, C is transposed to obtain the final result.

D. Experimental Setup

In this paper, Intel C++ Compiler 2019 was used to compile all the implementation. In the measurements of the implementations the K-best scheme was applied, with K=3 with 5% tolerance and at most 8 execution times, also between each measurement the cache is cleared.

To conclude, all implementations were tested with 4 different data structures sizes. Each size was chosen based on the following constraint:

3 * Sizeof Float * Number of elements $<$ Cache Level Size

With the application of the above constrain, the following table was obtained:

Size	Description
30x30	All data structures fit in Cache level 1
120x120	All data structures fit in Cache level 2
600x600	All data structures fit in Cache level 3
2400x2400	All data structures fit in RAM

VI. Comparing different implementations

A. Main Memory Behaviour

Since different implementations were tested, knowing RAM accesses per instruction and the number of bytes transferred to/from RAM is useful to know which implementation maybe spending to much time in memory accesses and possibly fix it.

After the analysis of the algorithm, it is possible to identify that all implementation access the elements of A, B and C in row or column wise.

If the matrix elements are accessed column-wise, it is possible that every iteration results in a memory access, in contrast, if they are accessed in row-wise, ideally, we will only make a memory access every 16 iterations, since a cache line can hold up to 16 float elements.

Having this in mind the following formulas estimate the number of accesses:

1) **Colunm-wise**: N
2) **row-wise**: $N * \frac{sizeoffloat}{cachelinesize}$

These formulas represent the number of total RAM access in one loop.

To calculate the total number of RAM accesses in the whole algorithm, the number of accesses must be multiplied by the number of iterations. The transpose function used in this paper has $\frac{N(N+1)}{2}$ iterations and in each iteration two elements are accessed from the RAM. Finally, to obtain the number of bytes loaded or stored in RAM and the number of RAM accesses per instruction, the number of accesses must be multiplied by the cache line size, 64 bytes, and divided by the number of instructions, respectively.

Implementation	30x30	120x120	600x600	2400x2400
IJK	861.308E-6	68.967E-6	10.159E-6	10.795E-6
JKI	869.605E-6	80.699E-6	44.742E-6	29.765E-6
IKJ	843.871E-6	67.508E-6	8.932E-6	2.315E-6
IJK + transpose	773.425E-6	85.304E-6	19.49E-6	7.486E-6
JKI + transpose	922.371E-6	114.313E-06	30.047E-6	12.918E-6

TABLE II
RAM ACCESSES PER INSTRUCTION

Implementation	30x30	120x120	600x600	2400x2400
IJK	10.06	59.48	867.77	59,370.85
JKI	10.83	61.35	4,231.29	168,996.94
IKJ	8.38	38.81	635.96	10,406.60
IJK + transpose	8.67	48.98	1,361.85	33,245.33
JKI + transpose	9.79	66.67	2,145.38	58,714.23

TABLE III
KIBYTES TRANSFERRED FROM/TO RAM

B. Cache behaviour

To obtain the information on the cache behaviour the memory reads of each level were measured with PAPI counters. To obtain the miss rate the following formulas were used:

$$L1\, miss\, rate = \frac{PAPI\_L1\_DCM}{PAPI\_LD\_INS}$$

$$L2\, miss\, rate = \frac{PAPI\_L2\_TCM}{PAPI\_L1\_DCM}$$

$$L3\, miss\, rate = \frac{PAPI\_L3\_TCM}{PAPI\_L2\_TCM}$$

In IV, it is possible to observe that the JKI implementation without transpose has a higher L1 miss rate and a lower

L3 miss rate than the implementation with transpose. This happens because in the latter implementation all the matrices are accessed row-wise, therefore, the algorithm spatial and temporal locality is increased.

implementation	Miss Rate	30x30	120x120	600x600	2400x2400
JKI	MR L1	0.0315%	0.2908%	4.4803%	4.1779%
	MR L2	5.8308%	0.0729%	0.4618%	5.0457%
	MR L3	5.9214%	3.8412%	0.0243%	0.0025%
JKI + transpose	MR L1	0.0290%	0.2579%	0.2645%	0.2684%
	MR L2	5.0303%	0.0874%	0.8431%	0.2076%
	MR L3	5.9531%	4.3244%	0.1132%	0.2114%

TABLE IV
CACHE LEVELS MISS RATE OF JKI IMPLEMENTATION

C. Floating Point Operations

The estimate number of Floating Point Operations is always $N^3 * 2$, since each implementation has 3 loops where each one has N iterations and the innermost loop performs 2 Floating Point Operations.

In order to prove our estimated PAPI_FP_OPS were used to measure how many floating point operations are executed in the program.

Implementation	30x30	120x120	600x600	2400x2400
IJK	54,297	3,471,574	435,046,714	27,778,100,174
JKI	54,437	3,595,398	2,361,628,493	59,866,322,544
IKJ	54,147	3,490,584	436,946,504	27,758,723,470
IJK + transpose	60,537	3,630,892	448,362,677	28,411,719,312
JKI + transpose	54,116	3,486,976	435,296,997	27,785,107,383

TABLE V
FLOATING POINT OPERATIONS

D. Execution Time

The implementations referred before were executed multiples times with the various data sets. In VI it is possible to verify that all implementations times are similar for 30x30 data sets, due to the fact that Cache level 1 is big enough to contain all 3 matrices. With 120x120 data sets the results are also similar due to the small size of the data sets.

For the 600x600 data sets, the differences in execution times between implementations is clear. It is also clear that both implementations that access one or more matrices column-wise, benefits with the application of the transpose, this optimisation brings a performance increase of 1.2 and 8.3 for the IJK and JKI implementations, respectively, being the JKI implementation the worst and IKJ the best.

Finally, for the 2400x2400 data sets, is possible to observe that IKJ implementation is the best between all implementations and JKI implementation is also the worst. It is also possible to observe that all implementation time measurements, where the transpose function is applied, are very similar to measurements of IKJ implementation. This happens due to the fact that applying the transpose function in the matrices increases spatial locality and reduces the number of cache misses.

Implementation	30x30	120x120	600x600	2400x2400
IJK	0.0217	1.181	147.956	9,344.063
JKI	0.0200	1.173	1,016.427	28,010, .100
IKJ	0.0180	0.974	121.317	7,643.937
IJK + transpose	0.0207	1.021	124.732	7,662.187
JKI + transpose	0.0200	1.007	122.846	7,723.400

TABLE VI
TIME MEASUREMENTS(MS)

With the results from III and V the achieved performance was plotted of each implementation in the XEON roofline model.

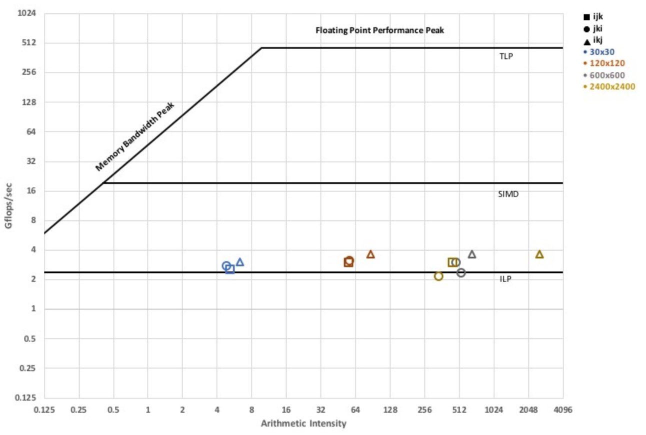

Fig. 3. Achieved performance

In this graphic it is possible to recognise, that all markers are in the right side of the ridge point, therefore, this kernel is compute bound.

VII. Optimisations

A. Block Optimisation

This optimisation organises the data structures in chunks or blocks. This technique loads a block into the cache memory, does all the necessary operations on the chunk, discards the chunk and repeats the process. The key idea is to reuse the data as much as possible.

Implementation/Block Size	32	48	96
IJK	9,764.39	9,336.78	9,093.24
JKI	10,306.8	9,992.94	9,846.15
IKJ	10237.4	9928.47	9796.61

TABLE VII
TIME MEASUREMENTS(MS) WITH DIFFERENT BLOCK SIZES FOR THE LARGEST DATA SET

In VII is possible to verify that there are no improvements in execution time. This happens because the kernel already takes advantage of temporal and spatial locality, therefore, this optimisation only introduces overhead due to the extra cycles. It is also possible to verify that IJK has a better execution time than IKJ, this maybe happening because B is preprocessed before the multiplication.

B. Vectorization

The computing platform that was used, node 662, features AVX extension, therefore, it can apply one instruction over 8 elements. With these instructions the number of instructions executed can be reduced substantially, improving the performance of the execution.

Implementation	30x30	120x120
IJK + vectorization	0.041	2.554
JKI + vectorization	0.043	2.674
IKJ + vectorization	0.041	2.651

TABLE VIII
TIME MEASUREMENTS(MS) WITH VECTORIZATION AND AN BLOCK SIZE OF 10

C. Multicore

Until now, the kernel was only executed with a single core, however, the computing platform possesses 24 cores, therefore, the kernel isn't taking advantage from using all the above, so, an implementation with OpenMP was made to take full advantage of these cores. This implementation was tested for 3 different block sizes.

Implementation/Block Size	32	48	96
IJK + OpenMP	327.033	266.619	255.685
JKI + OpenMP	330.718	283.960	273.460
IKJ + OpenMP	294.396	247.392	234.465

TABLE IX
TIME MEASUREMENTS(MS) WITH OPENMP (24 THREADS), VECTORIZATION FOR THE LARGEST DATA SET

D. Intel Knights Landing many-core server

Intel Knights Landing many-core server has 64 and supports up to 4 threads per core, therefore, the kernel was tested with 64, 128 and 256 threads and for 3 different block sizes.

Implementation/Block Size	32	48	96
IKJ + KNL (64 threads)	198.60	130.24	89.43
IKJ + KNL (128 threads)	174.84	121.71	81.11
IKJ + KNL (256 threads)	210.12	136.16	87.11

TABLE X
TIME MEASUREMENTS(MS) WITH OPENMP, VECTORIZATON AND VARIOUS BLOCK SIZES IN KNL FOR THE LARGEST DATA SET

E. GPU Kepler

For this task a NVIDIA Kepler was used. With the developed kernel each thread will process one element of the matrix. To test the kernel a matrix size of 2400x2400 was used.

Implementation	Computation Time	Communication Time
IKJ + GPU (Kepler)	135.40	47.92

TABLE XI
TIME MEASUREMENTS(MS) IN GPU KEPLER

F. Results analysis

The results in VII show that the blocking optimisation doesn't bring any gain of performance. If this optimisation it is applied to a kernel that already takes advantage of temporal and spatial locality it will only add loop overhead, reducing the performance. For this optimisation a block size of 32, 48 and 96 were tested being the latter the size that obtains better results.

By observing the results in VIII, it is possible to conclude that the vectorization technique doesn't increase performance, this can occur because the load and store operations have a start-up time that has a negative impact in performance, if it is implemented with small data sets. Other possible explanation can be that, the thermal throttle in the chip is being activated reducing the clock frequency for safety purposes since, the vectorial units have a great impact in the processor temperature.

The multicore approach has obtained considerably high increase of performance. In order to compare the multicore approach with a the sequential approach a IKJ implementation using vectorization and a block size of 96 was tested. The sequential version has obtained an execution time of 3396 ms the OpenMP version with 24 cores, vectorization and a block size of 96 has obtained an execution time of 234.465 having a speedup of 14.5.

The GPU implementation was optimised with blocking, with a tile size of 16. Compared to the OpenMP implementation the GPU approach has obtained a performance improvement of 1.3.

The implementation for the many-core server also obtained performance gains, providing the best times in this practical project, having a speedup of 2.9 and 2.3 compared to OpenMP and GPU approach, respectively. While having a closer look at the results of the Knights Landing, it possible to observe a performance degradation with the usage of 128 and 256 threads. One of the most plausible reasons for the degradation of the performance and overall results when adding more then 64 threads, is the fact that there isn't enough bandwidth to provide data to all the vectorial units that are used (128 units of 512 bits in simultaneous) and, even if it could provide this amount of data, there would be memory contingency problems. The KNL implementation

VIII. CONCLUSIONS

When developing a kernel with the objective of obtaining maximum performance it is possible to encounter some adversities that, at first sight, don't have a reasonable explanation.

The hardware characterisation is important to know the computing platform features that can improve performance. Some examples are: SIMD implementations and Fused Multiply Addition instructions.

The compilation of the program is also a extremely important step if combined with hardware characterisation. By default the compiler assumes that your kernel will be used in any computing platform, in consequence, some features, like the full width of vectorial operations, won't be used.

The representative model and the profiling tool used in this paper allow any user to identify possible improvements and performance problems in the kernel.

The analysis of the kernel is necessary, since, it can offer insights about the algorithm performance. Having analysed the matrix multiplication kernel, it was possible to conclude that the matrices must be accessed row-wise. If, by any reason, the access can't be row-wise, the matrices should be transposed in order to take advantage of spatial locality, temporal locality and memory hierarchy.

Having a well-tuned sequential version of the kernel, there are various optimisations that one can do, some of them are Blocking, Vectorization and the usage of multicore.

Finally, the usage of a many-core server and a GPU are some options to achieve even higher performance, however, the kernel must be optimised in order to obtain significant gains.

APPENDIX

IX. Stream Benchmark

A. Node 662

- Compilation: -O3 -DSTREAM_TYPE=float -qopenmp -DSTREAM_ARRAY_SIZE=80000000 -DNTIMES=6
- Environment:OMP_NUM_THREADS=24

B. Team's Laptop

- Compilation: -O3 -DSTREAM_TYPE=float -qopenmp -DSTREAM_ARRAY_SIZE=80000000 -DNTIMES=6
- Environment:
  OMP_NUM_THREADS=2

X. Node 662 Hardware Characterisation

Manufacturer	Intel Corporation
Architecture	Ivy Bridge
Model	Intel Xeon E5-2695V2
# Cores	24
# Threads	48
Processor Frequency	2.4 GHz
Extensions	Intel SSE4.1, Intel SSE4.2, Intel AVX
FP Performance Peak	460.8 GFlops
L1 Cache	64KiB
L2 Cache	256KiB
L3 Cache	30MiB
Associativity	8/8/20 way set associative
RAM Memory	64 GiB
RAM Latency	14.16 ns
Memory Bandwidth Peak	47 GiB/s
#Memory Channels	4

TABLE XII
Cluster Node Hardware Characterisation

XI. Comparison between the various optimisations with IKJ order

	Time	Speed Up
IKJ	7,643.94	1
IKJ + Blocking	9796.61	0.78
IKJ + Previous + Vectorization	3396	2.88
IKJ + Previous + OpenMP	234.465	14.48
IKJ + Blocking in GPU	183.32	1.28
IKJ + Previous in KNL (64 threads)	89.426	2.05
IKJ + Previous in KNL (128 threads)	81.111	1.10
IKJ + Previous in KNL (256 threads)	87.11	0.93

TABLE XIII
IKJ implementation comparation with multiple operations, dataset with 2400*2400 elements and 96 as block size

XII. Compilation Flags

Default Compilation: -qopt-report-phase=vec -qopt-report=5 -static-libstdc++ -std=c++11 -Wno-unused-parameter -fno-alias -fargument-noalias -fstrict-aliasing
With Vectorization: -march=ivybridge -qopt-prefetch -unroll

With OpenMP: -march=ivybridge -qopt-prefetch

KNL: -march=KNL

References

[1] Samuel Webb Williams, Andrew Waterman and David A. Patterson. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures.