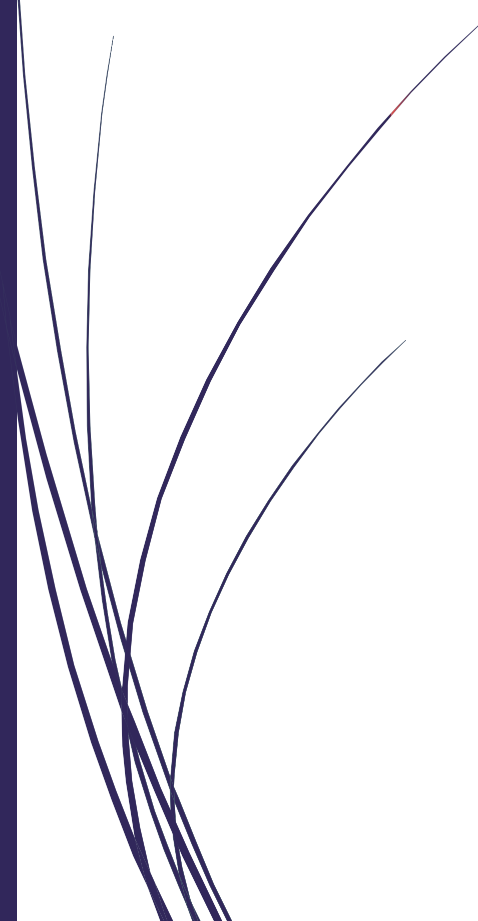
19/04/2019

Pricing American options on a NEC SX Aurora card

How fast is it ?



Wilfried KIRSCHENMAN
ALDWIN BY ANEO

Aldwin – Technical whitepaper

Authored by Wilfried Kirschenman, CTO adwin by ANEO, wkirschenmann@aneo.fr

© all right reserved

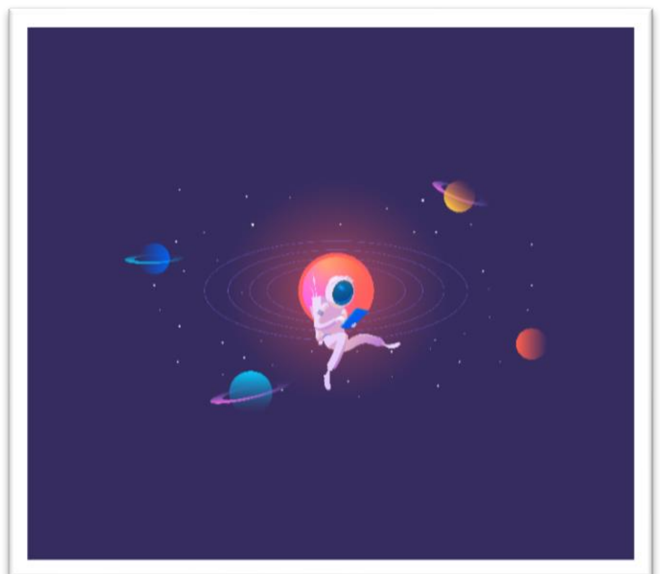


Table des matières

Disclaimer 2
SX-Aurora TSUBASA 3
  Not a sequel to the SX series : a reboot ! 3
  The Vector Engine 3
American option valuation 4
  Implementation and porting 6
Benchmarks 7
  Machine configuration 7
  Performances 7
Conclusion 8
Bibliography 9

Aldwin – Technical whitepaper

Authored by Wilfried Kirschenman, CTO adwin by ANEO, wkirschenmann@aneo.fr

© all right reserved

Introduction

NEC is famous for their vector supercomputers. The SX series was the first to exceed 1 gigaflop in the early 80s. It lead the famous Top500 of supercomputers in the early 90 and early 2000s.

Ever heard of the Earth-Simulator ? In late 2017, NEC announced the last of the series: the SX Aurora.

Last year, we had the opportunity to test the new architecture and here are the results: it works quite well!

First, let's present the machine in the next section. Then we will discuss its performances on a compute intensive benchmark. Finally will come the performances and the comparison to Intel's Skylake.

Disclaimer

The tests were done in xxxxx 2018.

Aldwin – Technical whitepaper
Authored by Wilfried Kirschenman, CTO adwin by ANEO, wkirschenmann@aneo.fr
© all right reserved

SX-Aurora TSUBASA

Not a sequel to the SX series: a reboot !

Yes. It's the same name. It's the same kind of processor architecture : a multicore vector processor designed for high performance applications. It provides a huge compute power and a huge memory bandwidth and comes with an efficient compiler with strong auto-vectorization capabilities.

But that's basically where the resemblance stops. The newborn is not a supercomputer, it's a PCIe-based accelerator card: you can have your own SX-Aurora in your workstation. The shift is aimed at satisfying the needs of different users and requirements. NEC provides a number of workstations models ranging from one to eight VE cards.

So it looks like a new kind of GPGPU competitor? No. When using GPGPU, your software runs on the CPU and some kernels are offloaded onto the accelerator. When using SX-Aurora, the full application is transferred and executed on the card, except for the system calls and some other I/O functionalities. Therefore, your data do not spend their time traveling on the PCIe bus.

To allow for users to run their applications easily on the card, NEC provides it with a large memory capacity (up to 48 GiB) and standard FORTRAN, C and C++ compilers. No need of a CUDA training! And it's easy to use whenever you need to do something that the card cannot do (using your favorite Infiniband network or access a hard drive), the corresponding system calls are transparently offloaded to the host CPU.

Running a typical program on the SX-Aurora is relatively simple: you simply run it as any classic x86 program.

The Vector Engine

The heart of SX-Aurora is the Vector Engine (VE) processor: a 8-cores chip. To allow for a good bandwidth, NEC couldn't use the historical strategy based on a large number of memory channel (SX-Ace had 16 channels of DDR3) as it would requires too much board space. Instead, NEC choose to use 6 HBM2 modules. You can move your data as fast as 1200 GB/s. As a comparison, the last NVIDIA GPGPU, the V100 "only" provides a 900 GB/s bandwidth.

Authored by Wilfried Kirschenman, CTO adwin by ANEO, wkirschenmann@aneo.fr
© all right reserved

There are eight vector core in each processor. This might seem few to those unfamiliar with vectors processors. Let's just say that each core is able to perform up to 192 double-precision floating-point operations per cycle. With a 1.6 GHz clock frequency, this lead to a 307.2 GFLOPS per core for a total of up to 2.45 TFLOPS per chip.

These performances are achieved thanks to the Vector Processing Unit (VPU). This VPU provides three independent vectorized FMAs, each of them able to process 256 double-precision FMAs (One FMA is one multiplication and one addition fuzzed in a single instruction. Thus, ones FMA instruction stands for two operations) each 8 cycles. The SX-Aurora indeed has a very wide vector length of 2048 bits (compared to the 512 bits of Intel's AVX512).

The choice of having such wide vectors that require eight cycle per instruction is probably to hide latency of memory transfers. With the choice of the memory architecture, this make the SX-Aurora a very well-balanced processor able to process each byte transferred from the memory.

Of course, as the whole program runs on the SX-Aurora, each vector core also has a Scalar Processing Unit (SPU) fully capable of running the card OS. One of the main task of the SPU is to keep the VPU as busy as possible.

We won't describe more thoroughly the Vector Engine architecture here. Interested readers can have a look on wikichip.org (https://en.wikichip.org/wiki/nec/microarchitectures/sx-aurora.

American option valuation

Authored by Wilfried Kirschenman, CTO adwin by ANEO, wkirschenmann@aneo.fr
© all right reserved

We used an in-house code pricing American options. All mathematical details can be found in [1,2]. Here, we will only describe it from a computing point of view. The algorithm can be described by the following steps.

1. **Generation of random sequences**: using the philox4_32_10 random number generator [3], 200 000 sequences of 100 elements are computed. These 100 elements correspond to the time steps of our valuation. The implementation has been inspired by the one provided with CUDA. Our implementation is vectorized so that each element of a vector belong~~ing~~ in a different sub-sequence.

2. **Diffusion of the underlying prices**: for each of the previous sequences, we compute the price trajectory of the underlying. Basically, some floating-point arithmetic and two exponentials per point of the grid. In the implementation, this step is performed on the fly with the generation of the sequences.

3. **Backward propagation**: this step is the most complex one. For each trajectory, the value of the option at a given timestep depends on the value of the underlying at the current timestep for all the trajectories and the value of the option for all trajectories at the next timestep. Hence, going backward, we compute for each timestep the value of the option. To do so, we have to:

   a. Compute using a Least-Mean Square (LMS) the average polynomial linking the value of the underlying at the current timestep and the value of the option at the next timestep. A small part of this computation is sequential (computing the LMS with a 4x4 matrix).

   b. Compute the value of the option for each trajectory using the previous polynomial.

4. **Aggregation of the values of all trajectories**: different averages and standard deviation are computed from all the results.

This code is very interesting as a benchmark because its different phases stress the architecture very differently.

1. The generation of random sequences is rather compute intensive as it requires almost no memory read at all, only writes.

2. The diffusion is still computationally intensive thanks to the exponentials, but more memory accesses are required. On a modern multicore CPU with AVX512 vectorization, the memory wall is hit.

Authored by Wilfried Kirschenman, CTO adwin by ANEO, wkirschenmann@aneo.fr
© all right reserved

3. The backward propagation is memory-bound (less than 1 instruction per memory access) and the cache memory doesn't really help.
4. Finally, the behavior of the aggregation depends on the cache memory performances.

Note that when the computation is performed in single precision, it incurs mixed precision computation in the backward propagation to ensure the quality of the results. Benefits are still there in terms of memory consumption but not in terms of execution time.

Implementation and porting

Our implementation is designed to exploit the full capabilities of both scalar and vector architectures. The code is implemented in C++ 17. We use OpenMP [4] to multithread the application and implemented the vectorization in two way: with OpenMP or with the Eigen library [5]. For the Generation, the vectorization requires some specific allocation. Hence, for this part, the vectorization is explicitly done. The width of this vectorization is a code variable. The value must divide the number of trajectories (200 000 in our case).

As NEC provides a C++ compiler. However, it doesn't support C++17 yet. So we had to backport the code the C++14 (i.e.: just removing some `if constexpr`). However, we had to face another issue: the NEC compiler doesn't optimize the code well when it uses `static const` variables. As Eigen heavily relies on C++ metaprogramming, such variables are found everywhere in it and we had to remove all reference to the library. Except for this, porting the code to the SX-Aurora was straightforward: the only modification made was the width vectorization. We set it to 10 000 to enforce the instruction pipelining.

Aldwin – Technical whitepaper
Authored by Wilfried Kirschenman, CTO adwin by ANEO, wkirschenmann@aneo.fr
© all right reserved

Benchmarks

Machine configuration

We compared the performances of this code on the SX-Aurora and on an Intel server with 2 processors Intel Skylake 6148. The SX-Aurora card used in the benchmark is a SKU B. The following table give the mains characteristics of these two architectures:

	Intel Skylake 6148 (2 sockets)	SX-Aurora TSUBASA SKU B
Memory (GiB)	192	48
Bandwidth (GB/s)	238.42	1200
Number of cores	2x20	8
Frequency (GHz)	2.4	1.6
Vector width (bits)	512	2048
Computational Power (GFLOPs)	2200	2450
Cache size (last level)	L1: 1.25 MB L2: 20 MB L3: 27.5MB	16 MB

Performances

We ran the benchmark in both single and double precision and got the following results:

	Intel Skylake 6148 (1 core)	Intel Skylake 6148 (2x20 cores)	SX-Aurora TSUBASA SKU B (1 core)	SX-Aurora TSUBASA SKU B (8 core)

Authored by Wilfried Kirschenman, CTO adwin by ANEO, wkirschenmann@aneo.fr
© all right reserved

Single precision	27.9s	5.05s	1.28s	1.04s
Double precision	48.1s	5.97s	1.57s	1.28s

The first thing that we can observe is the fact that on both architectures, the performances does not scale up to with the core usage. This is expected due to the fact that the application is mostly memory bound. As a consequence, we observe that the execution time on the 8 cores of the SX-Aurora is 5x better than the execution time on the 2x20 Skylake cores. This acceleration corresponds to the ratio of the memory bandwidths (238 GB/s vs 1200 GB/s).

	SX-AURORA		Intel Skylake 6148	
	Double precision	Single precision	Double precision	Single precision
Random generation and diffusion	0.619s	0.391s	33.4s	11.1s
backPropagation	0.893s	0.868s	14.1s	16.2s
aggrégation	0.0184s	0.0176s	0.451s	0.562s

	SX-AURORA (8 threads)		Intel Skylake 6148 40 threads	
	Double precision	Single precision	Double precision	Single precision
Random generation and diffusion	0.289s	0.0984s	1.38s	0.637s
backPropagation	0.969s	0.936s	4.39s	4.39s
aggrégation	0.00509s	0.00468s	0.0122s	0.0154s

Conclusion

Aldwin – Technical whitepaper
Authored by Wilfried Kirschenman, CTO adwin by ANEO, wkirschenmann@aneo.fr
© all right reserved

The SX-Aurora TSUBASA is a vector supercomputer embedded on an PCIe card and the performances show it. We benchmarked the platform with an application tailored to take advantage of vectorization on CPU. Even though we faced some issues while porting the code due to the fact that it uses advanced C++ features, environment is still less constrained that for other accelerators. By far! If you stick to C++14 and avoid `static const` variables, it should be ok. At the end, if your code can be vectorized and you will probably end up with a speedup of x5 compared to a dual socket Skylake machine.

Bibliography

1. LONGSTAFF, Francis A. et SCHWARTZ, Eduardo S. Valuing American options by simulation: a simple least-squares approach. The review of financial studies, 2001, vol. 14, no 1, p. 113-147.

2. DESVILLES, Gilles B. The Cost of Accuracy in the Least Squares Monte Carlo Approach. 2010.

3. SALMON, John K., MORAES, Mark A., DROR, Ron O., et al. Parallel random numbers: as easy as 1, 2, 3. In : Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2011. p. 16.

4. BRADLEY, Con et GASTER, Benedict R. Exploiting loop-level parallelism for SIMD arrays using OpenMP. In : International Workshop on OpenMP. Springer, Berlin, Heidelberg, 2007. p. 89-100.

5. GUENNEBAUD, Gaël, JACOB, Benoit, et al. Eigen. URl: http://eigen. tuxfamily. org, 2010.