Software development strategy in the TREX Center of Excellence

Anthony Scemama\textsuperscript{1}, Claudia Filippi\textsuperscript{2}
18/06/2020

\textsuperscript{1}Lab. Chimie et Physique Quantiques, IRSAMC, UPS/CNRS, Toulouse (France)
\textsuperscript{2}Faculty of Science and Technology, University of Twente (Netherlands)

Disclaimer: Currently in the process of signing the Grant Agreement with the EU
Exascale
Worldwide technological competition

- 1997: Teraflops/s\(^1\)
- 2008: Petaflops/s
- 2020?: Exaflops/s

\(^1\)flops/s: floating point operations per second
Worldwide technological competition

- 1997: Teraflops/s\(^1\)
- 2008: Petaflops/s
- 2020?: Exaflops/s

- Expected increase of computational power is exponential
- Moore’s Law is ending
- Technological breakthrough needed (quantum computing?)

\(^1\)flops/s: floating point operations per second
The supercomputing race

Worldwide technological competition

- 1997: Terascale: Distributed parallelism
- 2008: Petascale: Multi-core chips or accelerators
- 2020?: Exascale: Hybrid architectures are inevitable
Worldwide technological competition

- 1997: Terascale: Distributed parallelism
- 2008: Petascale: Multi-core chips or accelerators
- 2020?: Exascale: Hybrid architectures are inevitable

Peak flops/s improved by $1000\times$. What about

- Memory capacity per core?
- Memory bandwidth? latency?
- I/O bandwidth? latency?
- Network bandwidth? latency?
Simple Math

Example: \( z(i) = a(i) + b(i) \times c(i) \) on Intel Xeon 8168 (2017)
Example: $z(i) = a(i) + b(i) \times c(i)$ on Intel Xeon 8168 (2017)

Computational power

- 24 cores
- $2.5 \text{ GHz} = 2.5 \times 10^9 \text{ cycles/s}$
- Fused multiply-add (FMA) : 2 flops/FMA
- 512 bit vectors : $\times 8$ flops in double precision
- $24 \times 2.5 \times 10^9 \times 8 \times 2 = 960 \text{ Gflops/s}$
Simple Math

Example: \( z(i) = a(i) + b(i) \times c(i) \) on Intel Xeon 8168 (2017)

**Computational power**

- 24 cores
- \( 2.5 \text{ GHz} = 2.5 \times 10^9 \text{ cycles/s} \)
- Fused multiply-add (FMA) : 2 flops/FMA
- 512 bit vectors : \( \times 8 \) flops in double precision
- \( 24 \times 2.5 \times 10^9 \times 8 \times 2 = 960 \text{ Gflops/s} \)

**Memory bandwidth**

- \( 2666 \text{ MHz} = 2.666 \times 10^9 \text{ cycles/second} \)
- 8 bytes / cycle
- 6 memory channels
- \( 2.666 \times 10^9 \times 6 \times 8 = 128 \text{ GB/s} \)
Example: \( z(i) = a(i) + b(i) \times c(i) \) on Intel Xeon 8168 (2017)

Peak performance requires an arithmetic intensity of \( \frac{960 \text{ Gflops/s}}{128 \text{ GB/s}} = 7.5 \text{ flops/byte} \)

\[ \frac{2 \text{ flops}}{32 \text{ bytes}} = \frac{0.0625 \text{ flops}}{\text{byte}} \]

0.8% of the peak is expected for this loop, if streamed from memory.
Example: \( z(i) = a(i) + b(i) \times c(i) \) on Intel Xeon 8168 (2017)
Peak performance requires an \textbf{arithmetic intensity} of \( \frac{960 \text{ Gflops/s}}{128 \text{ GB/s}} = 7.5 \text{ flops/byte} \)

\[ z(i) = a(i) + b(i) \times c(i) \]

- 2 flops
- 3 loads + 1 store : \( 4 \times 8 = 32 \text{ bytes} \)
- Arithmetic intensity : \( \frac{2}{32} = 0.0625 \text{ flops/byte} \)
Example: \( z(i) = a(i) + b(i) \times c(i) \) on Intel Xeon 8168 (2017)

Peak performance requires an arithmetic intensity of \((960 \text{ Gflops/s}) / (128 \text{ GB/s}) = 7.5 \text{ flops/byte}\)

\[
z(i) = a(i) + b(i) \times c(i)
\]

- 2 flops
- 3 loads + 1 store: \(4 \times 8 = 32\) bytes
- Arithmetic intensity: \(2/32 = 0.0625 \text{ flops/byte}\)

0.8% of the peak is expected for this loop, if streamed from memory.
Example: $z(i) = a(i) + b(i) \times c(i)$ on an Nvidia A100 GPU (2020)

**A100 GPU**

- 9.7 Tflops/s
- 1.555 TB/s
- Required arithmetic intensity on GPU: $(9.7 \text{ Tflops/s}) / (1.555 \text{ TB/s}) = 6.2 \text{ flops/byte}$
Example: \( z(i) = a(i) + b(i) \times c(i) \) on an Nvidia A100 GPU (2020)

**A100 GPU**

- 9.7 Tflops/s
- 1.555 TB/s
- Required arithmetic intensity on GPU: \( \frac{(9.7 \text{ Tflops/s})}{(1.555 \text{ TB/s})} = 6.2 \text{ flops/byte} \)

1.0% of the peak is expected for this loop, but \( \frac{1555}{128} \approx 12 \times \) faster than a single CPU.
Example: $z(i) = a(i) + b(i) \times c(i)$ on an Nvidia A100 GPU (2020)

A100 GPU

- 9.7 Tflops/s
- 1.555 TB/s
- Required arithmetic intensity on GPU: $(9.7 \text{ Tflops/s}) / (1.555 \text{ TB/s}) = 6.2 \text{ flops/byte}$
- PCIe : 64GB/s
- Required arithmetic intensity from RAM: $(97 \text{ Gflops/s}) / (64 \text{ GB/s}) = 151.5 \text{ flops/byte}$ (!)

1.0% of the peak is expected for this loop, but $1555/128 \sim 12 \times$ faster than a single CPU.
Example: $z(i) = a(i) + b(i) \times c(i)$ on an Nvidia A100 GPU (2020)

**A100 GPU**

- 9.7 Tflops/s
- 1.555 TB/s
- Required arithmetic intensity on GPU: $(9.7 \text{ Tflops/s}) / (1.555 \text{ TB/s}) = 6.2 \text{ flops/byte}
- PCIe: 64GB/s
- Required arithmetic intensity from RAM: $(97 \text{ Gflops/s}) / (64 \text{ GB/s}) = 151.5 \text{ flops/byte}$ (!)

1.0% of the peak is expected for this loop, but $\frac{1555}{128} \sim 12 \times$ faster than a single CPU.

Streaming from memory is slower than on CPU
QMC is known to be “easily scalable”.
QMC is known to be “easily scalable”.

- Scalable : YES
  - We can trivially run many trajectories in parallel
- Easily : NO!
QMC is known to be “easily scalable”.

- Scalable: YES
  We can trivially run many trajectories in parallel
- Easily: NO!

For *exascale simulations*, we need to be *massively parallel and efficient*.
A few words on Quantum Monte Carlo

Efficiency

- Small systems $\equiv$ small matrices $\implies$ low arithmetic intensity
- Very large systems $\implies$ linear-scaling algorithms $\implies$ very low arithmetic intensity by nature

Parallelism within a trajectory

- DMC trajectories need to be ergodic, and one trajectory is sequential by nature
- One trajectory performs $\sim10M$–$100M$ steps $\implies$ one step $\sim$ 1 millisecond

Exploiting parallelism improve the efficiency is difficult
Software development
• Progress in quantum chemistry may require codes with new ideas/algorithms
• New ideas/algorithms are implemented by physicists/chemists
• Exascale machines will be horribly complex

Is it reasonable to ask physicists/chemists to write codes for exascale machines?
Vector addition (from https://github.com/jeffhammond/dpcpp-tutorial)

```cpp
std::vector<float> h_X(length, xval);
std::vector<float> h_Y(length, yval);
std::vector<float> h_Z(length, zval);

try {
    sycl::queue q(sycl::default_selector{});

    const float A(aval);

    sycl::buffer<float, 1> d_X { h_X.data(), sycl::range<1>(h_X.size()) }
    sycl::buffer<float, 1> d_Y { h_Y.data(), sycl::range<1>(h_Y.size()) }
    sycl::buffer<float, 1> d_Z { h_Z.data(), sycl::range<1>(h_Z.size()) }

    q.submit([&](sycl::handler &h) {
        auto X = d_X.template get_access<sycl::access::mode::read>(h);
        auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
        auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);

        h.parallel_for<class nstream>(sycl::range<1>(length), [=] (sycl::id<1> it) {
            const int i = it[0];
            Z[i] += A * X[i] + Y[i];
        });
        q.wait();
    })
    catch (sycl::exception & e) {
        std::cout << e.what() << std::endl;
        return 1;
    }
```
std::vector<float> h_X[length, xval];
std::vector<float> h_Y[length, yval];
std::vector<float> h_Z[length, zval];

try {
    sycl::queue q(sycl::default_selector());
    const float A[aval];

    sycl::buffer<float, 1> d_X [h_X.deta()], sycl::range<1>({h_X.size()});
    sycl::buffer<float, 1> d_Y [h_Y.deta()], sycl::range<1>({h_Y.size()});
    sycl::buffer<float, 1> d_Z [h_Z.deta()], sycl::range<1>({h_Z.size()});
    q.submit([&](sycl::handler& h) {
        auto X = d_X.template get_access<sycl::access::mode::read>(h);
        auto Y = d_Y.template get_access<sycl::access::mode::read>(h);
        auto Z = d_Z.template get_access<sycl::access::mode::read_write>(h);
        h.parallel_for<class myread>(sycl::range<1>({length}), [=] (sycl::sid<1> it) {
            const int i = it[0];
            Z[i] += A[i] * X[i] + Y[i];
        });
        q.wait();
    })
    catch (sycl::exception & e) {
        std::cout << e.what() << std::endl;
        return 1;
    }
}
The dream

A compiler\(^2\) that can read an average researcher’s code and transform it into highly efficient code on an exascale machine.

\(^2\)Wikipedia: A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language)
“AI” is not ready yet ...
...so let’s use “NI” and add a human layer between the machine and the researchers: a *bio-compiler*
Quantum Monte Carlo Kernel Library

1. Identify the common kernels in the QMC codes of TREX
Quantum Monte Carlo Kernel Library

1. Identify the common kernels in the QMC codes of TREX
2. Agree on a standardized API with scientists and HPC experts
Quantum Monte Carlo Kernel Library

1. Identify the common kernels in the QMC codes of TREX
2. Agree on a standardized API with scientists and HPC experts
3. Let the library handle data movement between different kernels
Quantum Monte Carlo Kernel Library

1. Identify the common kernels in the QMC codes of TREX
2. Agree on a standardized API with scientists and HPC experts
3. Let the library handle data movement between different kernels
4. Scientists propose a reference simple implementation
Quantum Monte Carlo Kernel Library

1. Identify the common kernels in the QMC codes of TREX
2. Agree on a standardized API with scientists and HPC experts
3. Let the library handle data movement between different kernels
4. Scientists propose a reference simple implementation
5. HPC experts *bio-compile* it in high-performance implementations
Quantum Monte Carlo Kernel Library

1. Identify the common kernels in the QMC codes of TREX
2. Agree on a standardized API with scientists and HPC experts
3. Let the library handle data movement between different kernels
4. Scientists propose a reference simple implementation
5. HPC experts \textit{bio-compile} it in high-performance implementations
6. Integrate the library in TREX codes
A very old and successful practice

- BLAS/Lapack (Linear Algebra)
- MPI (Communication)
- FFTW (Fourier transforms)
- OpenMP (shared-memory parallelism)
- OpenGL (3D graphics)
- MPEG (Audio/Video encoding/decoding)
- Video game rendering engines
Benefits of this model

For scientists

- We don’t impose a programming language
- Codes will not die with a change of architecture

Separation of concerns

- Scientists will never have to manipulate low-level HPC code
- HPC experts will not be required to be experts in theoretical physics
“QMC can scale easily” assumes that one can efficiently compute one trajectory. Using efficiently the hardware to accelerate the realization of one trajectory is the challenge that will be taken by QMCkl to enable easy scalability.