Software development strategy in the TREX Center of Excellence

Anthony Scemama[1], Claudia Filippi[2]

18/06/2020

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

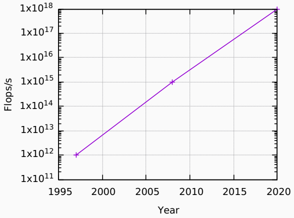
Disclaimer: Currently in the process of signing the Grant Agreement with the EU

Exascale

The supercomputing race

Worldwide technological competition

- 1997 : Teraflops/s[1]
- 2008 : Petaflops/s
- 2020? : Exaflops/s

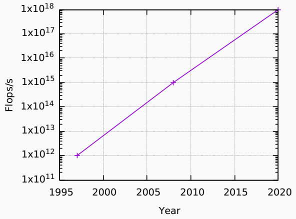


[1]flops/s: floating point operations per second

The supercomputing race

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

The supercomputing race

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)

Simple Math

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

Computational power

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

Memory bandwidth

- 2666 MHz $= 2.666\,10^9$ cycles/second
- 8 bytes / cycle
- 6 memory channels
- $2.666\,10^9 \times 6 \times 8 =$ 128 GB/s

Simple Math

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

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

Simple Math

Example: $z(i) = a(i) + b(i) \times c(i)$ on Intel Xeon 8168 (2017)
Peak performance requires an arithmetic intensity of (960 Gflops/s) / (128 GB/s) = 7.5 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 flops/byte

Simple Math

Example: $z(i) = a(i) + b(i) \times c(i)$ on Intel Xeon 8168 (2017)
Peak performance requires an arithmetic intensity of (960 Gflops/s) / (128 GB/s) = 7.5 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 flops/byte

0.8% of the peak is expected for this loop, if streamed from memory.

Simple Math

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 Tflops/s) / (1.555 TB/s) = 6.2 flops/byte

Simple Math

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 Tflops/s) / (1.555 TB/s) = 6.2 flops/byte

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

Simple Math

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 Tflops/s) / (1.555 TB/s) = 6.2 flops/byte
- PCIe : 64GB/s
- Required arithmetic intensity from RAM: (97 Gflops/s) / (64 GB/s) = 151.5 flops/byte (!)

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

Simple Math

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 Tflops/s) / (1.555 TB/s) = 6.2 flops/byte
- PCIe : 64GB/s
- Required arithmetic intensity from RAM: (97 Gflops/s) / (64 GB/s) = 151.5 flops/byte (!)

1.0% of the peak is expected for this loop, but $1555/128 \sim 12\times$ faster than a single CPU.
Streaming from memory is slower than on CPU

A few words on Quantum Monte Carlo

QMC is known to be “easily scalable”.

A few words on Quantum Monte Carlo

QMC is known to be “easily scalable”.

- Scalable : YES

  We can trivially run many trajetories in parallel
- Easily : NO!

A few words on Quantum Monte Carlo

QMC is known to be "easily scalable".

- Scalable : YES
  We can trivially run many trajetories 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 $\Longrightarrow$ *low arithmetic intensity*
- Very large systems $\Longrightarrow$ linear-scaling algorithms $\Longrightarrow$ 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 $\sim$10M–100M steps $\Longrightarrow$ one step $\sim$ 1 millisecond

Exploiting parallelism improve the efficiency is difficult

Software development

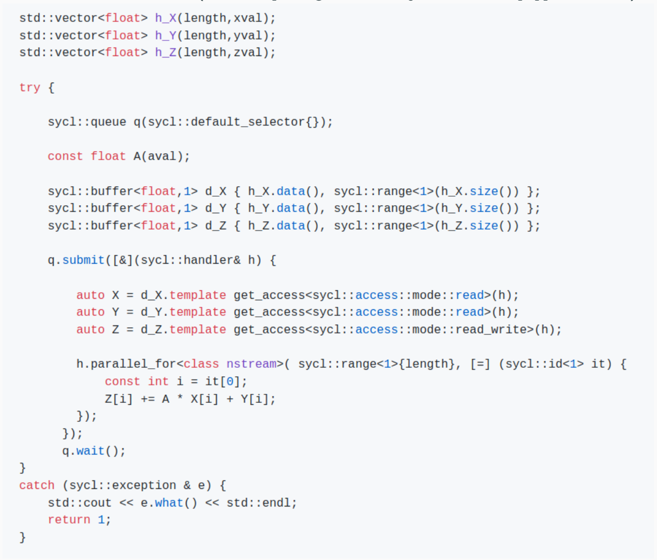
Programming for the exascale

- 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?

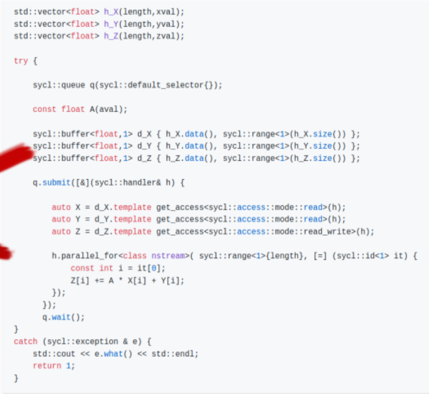
No: Proof

Vector addition (from `https://github.com/jeffhammond/dpcpp-tutorial`)



```
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;
}
```

No

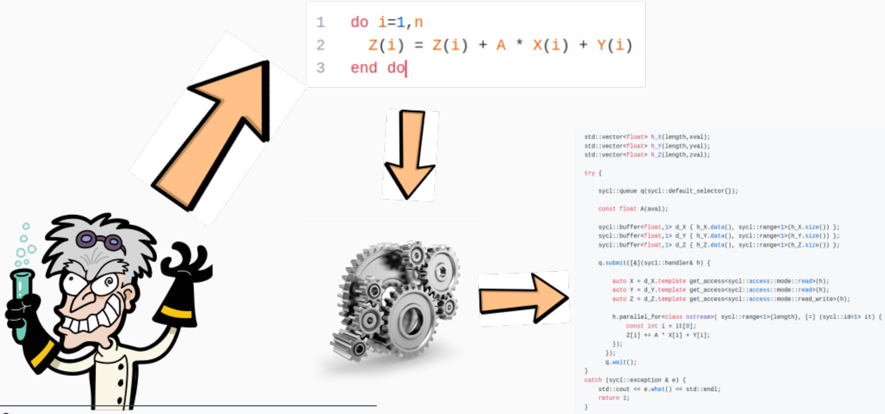


```
std::vector<float> h_X(length,xval);
std::vector<float> h_Y(length,yval);
std::vector<float> h_Z(length,zval);

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;
}
```

https://commons.wikimedia.org/wiki/File:Mad_scientist_transparent_background.svg

The dream

A compiler[2] that can read an average researcher's code and transform it into highly efficient code on an exascale machine.



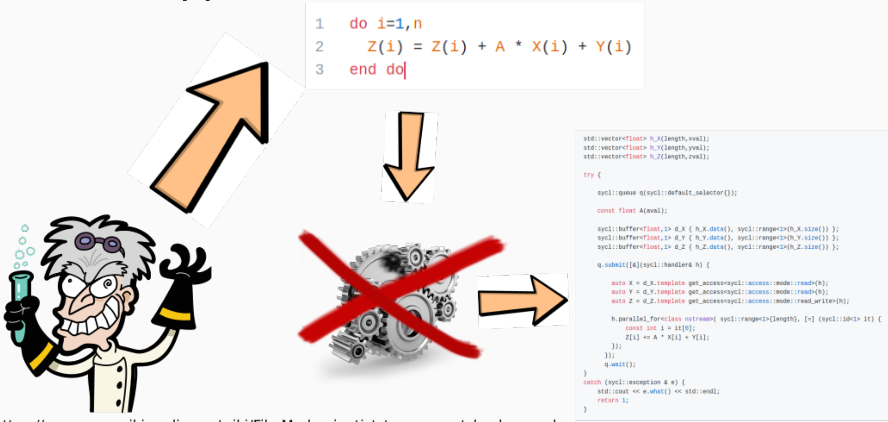
```fortran
do i=1,n
  Z(i) = Z(i) + A * X(i) + Y(i)
end do
```

```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;
}
```

[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)

Reality

“AI” is not ready yet . . .



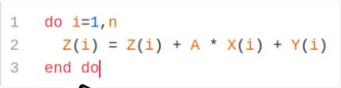
```fortran
do i=1,n
  Z(i) = Z(i) + A * X(i) + Y(i)
end do
```

```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;
}
```

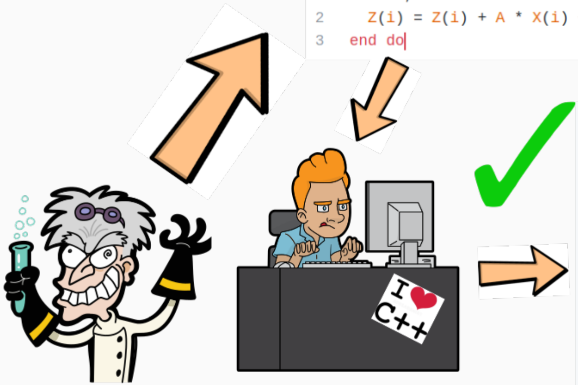
https://commons.wikimedia.org/wiki/File:Mad_scientist_transparent_background.svg

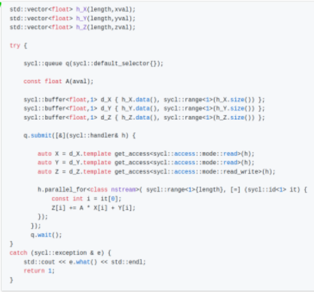
Reality

. . . so let's use "NI" and add a human layer between the machine and the researchers : a *bio-compiler*



```
do i=1,n
  Z(i) = Z(i) + A * X(i) + Y(i)
end do
```





```
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;
}
```

https://commons.wikimedia.org/wiki/File:Mad_scientist_transparent_background.svg

QMCkl

Quantum Monte Carlo Kernel Library

1. Identify the common kernels in the QMC codes of TREX

QMCkl

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

QMCkl

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

QMCkl

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

QMCkl

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

QMCkl

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
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

$\vdots$

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

Conclusion

"*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.