Performance vs **P**ortability vs **P**roductivity vs **P**recision

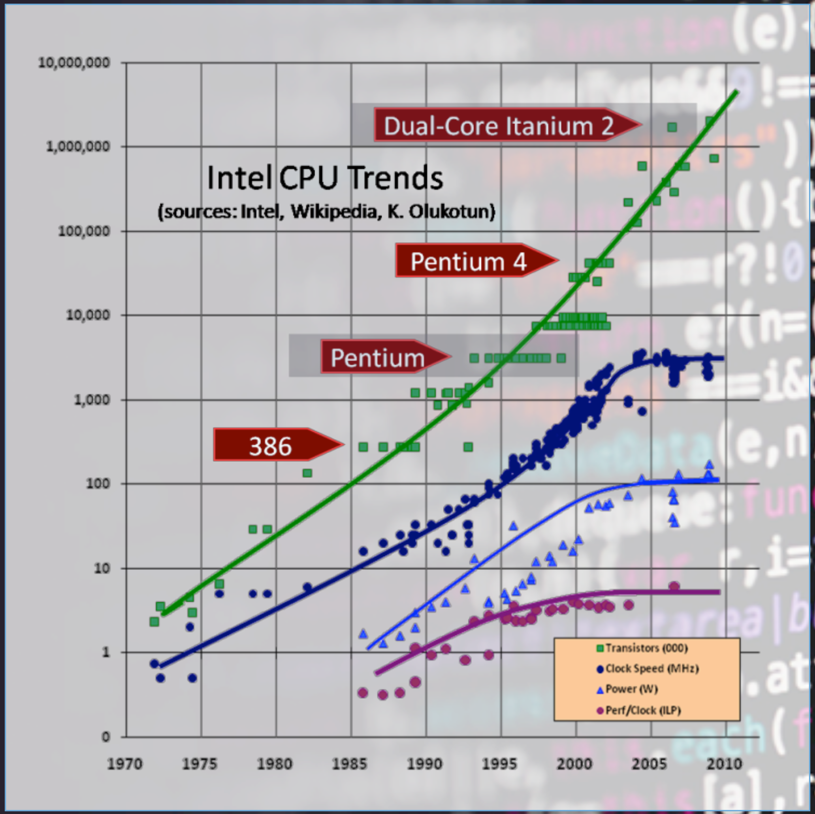
A trail in the computational hardware and software jungle

Learning to Discover : Advanced Pattern Recognition

David Chamont, Reprises, october 2019

The free lunch is over *Herb Sutter, 2005*

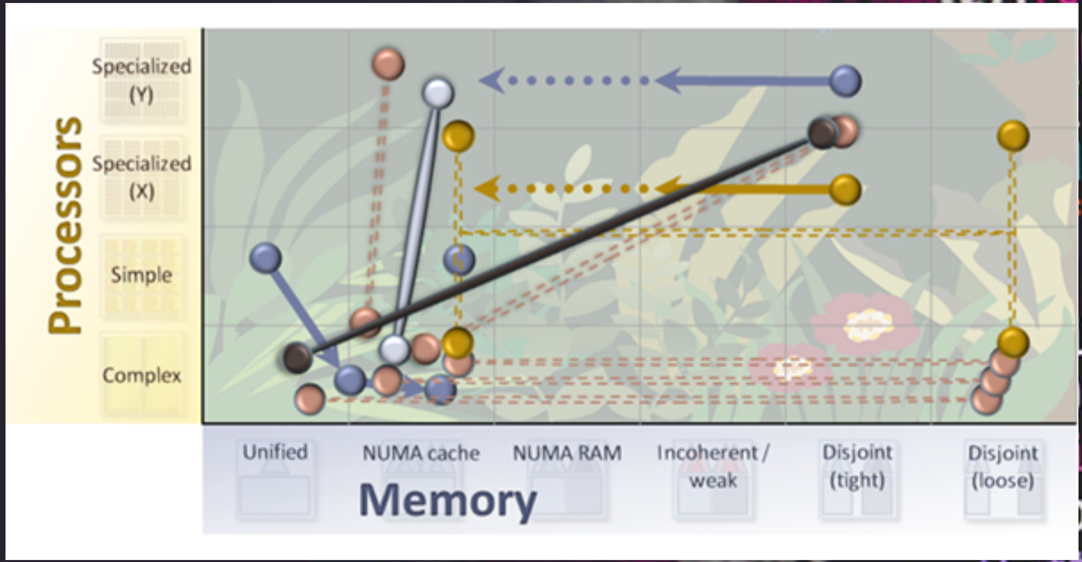
- Hardware : multi to many cores
  - The clock frequency stabilizes.
  - The number of transistors still increases, giving birth to many cores.
  - The memory does not grow so fast.
- Software: parallelism is not any more optional
  - The applications do not benefit directly from hardware upgrades.
  - Comeback of concurrent computation.



Welcome to the jungle

Herb Sutter, 2013

- Hardware : increasingly complex
  - Mix of big and small cores, specialized coprocessors, with dedicated communication channels.
  - The memory cannot be seen any more as an homogeneous black box.
  - The precision cannot any more be "double" everywhere and all the time.



- Software : necessarily hybrid
  - Graphic accelerators are not the philosophers' stone. To be combined with multi-threading, vectorisation, message-passing, distributed map-reduce, ...

Hardware jungle

- CPU vector registers
- GPGPU, ~~Xeon Phi~~
- FPGA Xilinx & Intel (Altera)
- Google Cloud TPU
- Brain Floating Point (bfloat16) ALUs
- *Qualcomm Cloud AI 100 (2020 ?)*
- *Quantum computers (??)*

Software jungle

- Code generation & DSL : TensorFlow, Loopy, xtensor.
- Vectorisation libraries : Vc, UME, bSIMD, xsimd,...
- Accelerators : OpenCL for GPU & FPGA, Vulkan.
- Directives : OpenACC/OpenMP.
- C/C++ libraries : Kokkos, Raja, Intel OneAPI...
- Runtimes & distribution : StarPU, HPX, Spark...
- Langages & extensions : SyCL, Pythran, Rust, Julia...

Reprises

An IN2P3 task force which tries to make its way in this jungle, and find the right balance between Performance, Portability, Productivity and Precision

- since 2 years
- a dozen engineers, from several labs of the institute.

Outline

- Computational hardware and software jungle
- Vectorization example : bSIMD
- GPU examples : OpenACC & Thrust
- Vectorization+GPU examples : OpenCL & SyCL
- Lessons learnt
- *Digression about functional programming*

SIMD Vectorization

some sleeping performance

Learning to Discover : Advanced Pattern Recognition

David Chamont, Reprises, october 2019

Why vectorize ?

- Processors have vector registers since a long time, and they gets longer and longer for each new generation.
- For the cost of one floating point operation, you can do 4, 8, 16... depending on your hardware and precision.

➢ What a waste not to use it !

How to vectorize ?

- Ask the compiler to auto-vectorize.
- Decorate your code with OpenMP directives.
- Use a library (VC, Xsimd, ...).
- At the lower level, explicitly call x86 intrinsic instructions.

bSIMD / headers

```
#include <boost/simd/include/functions/load.hpp>
#include <boost/simd/include/functions/store.hpp>
#include <boost/simd/include/functions/plus.hpp>
#include <boost/simd/include/functions/multiplies.hpp>

typedef simd::pack<double> pdouble ;
static const int psize = pdouble::static_size ;

.....
```

bSIMD / main

```
.....

// programme principal
int main( int argc, char * argv[] )
 {
  .....

  // prepare arrays
  double * const inputr = new double [dim] ;
  double * const inputi = new double [dim] ;
  double * const outputr = new double [dim] ;
  double * const outputi = new double [dim] ;

  // generate input
  .....

  .....
```

bSIMD / main

```
.....

 // compute
 compute_powers(dim,inputr,inputi,outputr,outputi,degree) ;

 // post-process output
 .....

 // cleaning
 delete [] inputr ;
 delete [] inputi ;
 delete [] outputr ;
 delete [] outputi ;
 return 0 ;
}
```

bSIMD / kernel

```
.....

void compute_powers ( int n, double * xreal, double * ximag, double * yreal, double * yimag, int d )
 {
  int i = 0 ;
  while ( i < n )
   {
    // load an simd set of values
    pdouble pxreal = simd::load<pdouble>(xreal) ;
    pdouble pximag = simd::load<pdouble>(ximag) ;

    // Computation
    pdouble prreal(1.0), primag(0.0), ptmp(0.0) ;
    for ( int j=0 ; j < d ; j++ )
     {
      ptmp    = prreal*pxreal - primag*pximag ;
      primag  = prreal*pximag + primag*pxreal ;
      prreal  = ptmp ;
     }

    .....
```

bSIMD / kernel

```
.....

   // store the result
   simd::store<pdouble>(prreal,yreal) ;
   simd::store<pdouble>(primag,yimag) ;

   // advance to the next simd vector
   i += psize ;
   xreal += psize ; ximag += psize ;
   yreal += psize ; yimag += psize ;
  }
}

.....
```

Feelings about SIMD

- Must be done !
- No library seems to get the upper hand today.
- Auto-vectorization is obviously the less invasive, but it tricky to know how and when it is really applied, and one must help the compiler, submitting relevant data structures ad algorithms.
- First step is always to AoS (arrays of objects) with SoA (one collection object which contains arrays of attributes).

Offloading computation to the GPU

yes but

Learning to Discover : Advanced Pattern Recognition
David Chamont, Reprises, october 2019

Why GPUs ?

- Best ratio Flops/Watt
- Together with data flood which feeds machine learning

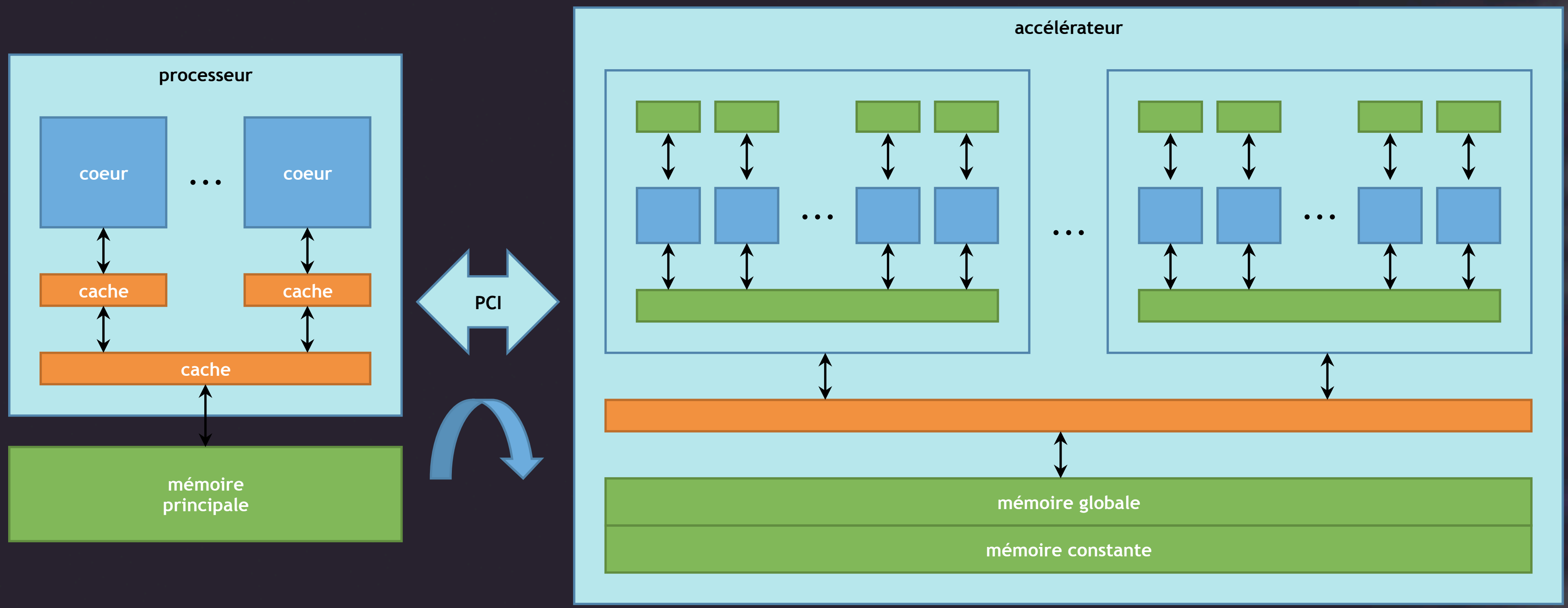
➢ winning spiral GPU / AA / BigData

- GPUs invades the supercomputers which fight to climb up TOP500 and GREEN500.
- The non-machine-learning must cope with it...

How to use them ?

- With an application already instrumented
  - Matlab, Mathematica, ...
- With a "high-level" library
  - CuBLAS, ArrayFire, TensorFlow, Kokkos, Eigen ...
- With directives or langage extension
  - OpenACC, OpenMP 4, SyCL
- Explicit low-level programming
  - CUDA
  - **OpenCL**

GPU structure



OpenACC / kernel

```
.....

void compute_powers
 ( int n, double * xreal, double * ximag,
  double * restrict yreal, double * restrict yimag, int d
 )
 {
# pragma acc kernels loop copyin(xreal[0:n],ximag[0:n]) copyout(yreal[0:n],yimag[0:n])
  for ( int i=0 ; i<n ; ++i )
   {
    double rreal_tmp, rreal = 1.0, rimag = 0.0 ;
    for (int j=0; j < d; j++)
     {
      rreal_tmp = rreal*xreal[i] - rimag*ximag[i] ;
      rimag     = rreal*ximag[i] + rimag*xreal[i] ;
      rreal     = rreal_tmp ;
     }
    yreal[i] = rreal ; yimag[i] = rimag ;
   }
 }

.....
```

OpenACC / main

```
.....

int main ( int argc, char * argv[] )
 {
  .....

  // prepare arrays
  double * inputr = (double *)malloc(dim*sizeof(double)) ;
  double * inputi = (double *)malloc(dim*sizeof(double)) ;
  double * outputr = (double *)malloc(dim*sizeof(double)) ;
  double * outputi = (double *)malloc(dim*sizeof(double)) ;

  // generate input
  .....
```

OpenACC / main

```
.....

 // compute
 compute_powers(dim,inputr,inputi,outputr,outputi,degree) ;

 // process results
 .....

 // cleaning
 free(inputr) ;
 free(inputi) ;
 free(outputr) ;
 free(outputi) ;

 return 0 ;
}
```

Thrust / headers

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/transform.h>

.....

struct complex
 {
  double real ;
  double imag ;
 } ;

.....
```

Thrust / kernel

```
.....

struct compute_powers
 {
  public :
    compute_powers( int degree ) : degree_(degree) {}
    __host__ __device__
    complex operator()( const complex & c )
     {
      complex r ;
      r.real = 1.0 ; r.imag = 0.0 ;
      double real_tmp ;
      for( int  j=0 ; j < degree_ ; j++ ) {
        real_tmp = r.real * c.real - r.imag * c.imag ;
        r.imag = r.real * c.imag + r.imag * c.real ;
        r.real = real_tmp ;
      }
      return r ;
     }
  private :
    int degree_ ;
 } ;

.....
```

Thrust / main

```
.....

int main( int argc, char * argv[] )
 {
  .....

  // prepare arrays
  thrust::host_vector<complex> hinput(dim), houtput(dim) ;
  thrust::device_vector<complex> dinput(dim), doutput(dim) ;

  // prepare input
  .....

  // transfer and compute
  dinput = hinput ;
  thrust::transform(dinput.begin(),dinput.end(),doutput.begin(),compute_powers(degree)) ;
  houtput = doutput ;

   // process output
  .....

  return 0 ;
 }
```

Feelings about GPU

- The best ratio Flops/Watt... if the GPU is operating at full capacity. which requires :
  - copying input data (and this has a cost)
  - enough computation (high arithmetic intensity)
  - deal with the many cores competing for the GPU resource
  - ensure those cores have things to do while waiting GPU results
- No library seems to get the upper hand today ? Eigen ?
- Programming them may seems tedious, but the effort will pay off, even if you do not finally run on GPU.

OpenCL

Vectorization + acceleration portable
across CPU, GPU, FPGAs... damned !?!

Learning to Discover : Advanced Pattern Recognition
David Chamont, Reprises, october 2019

OpenCL / kernel

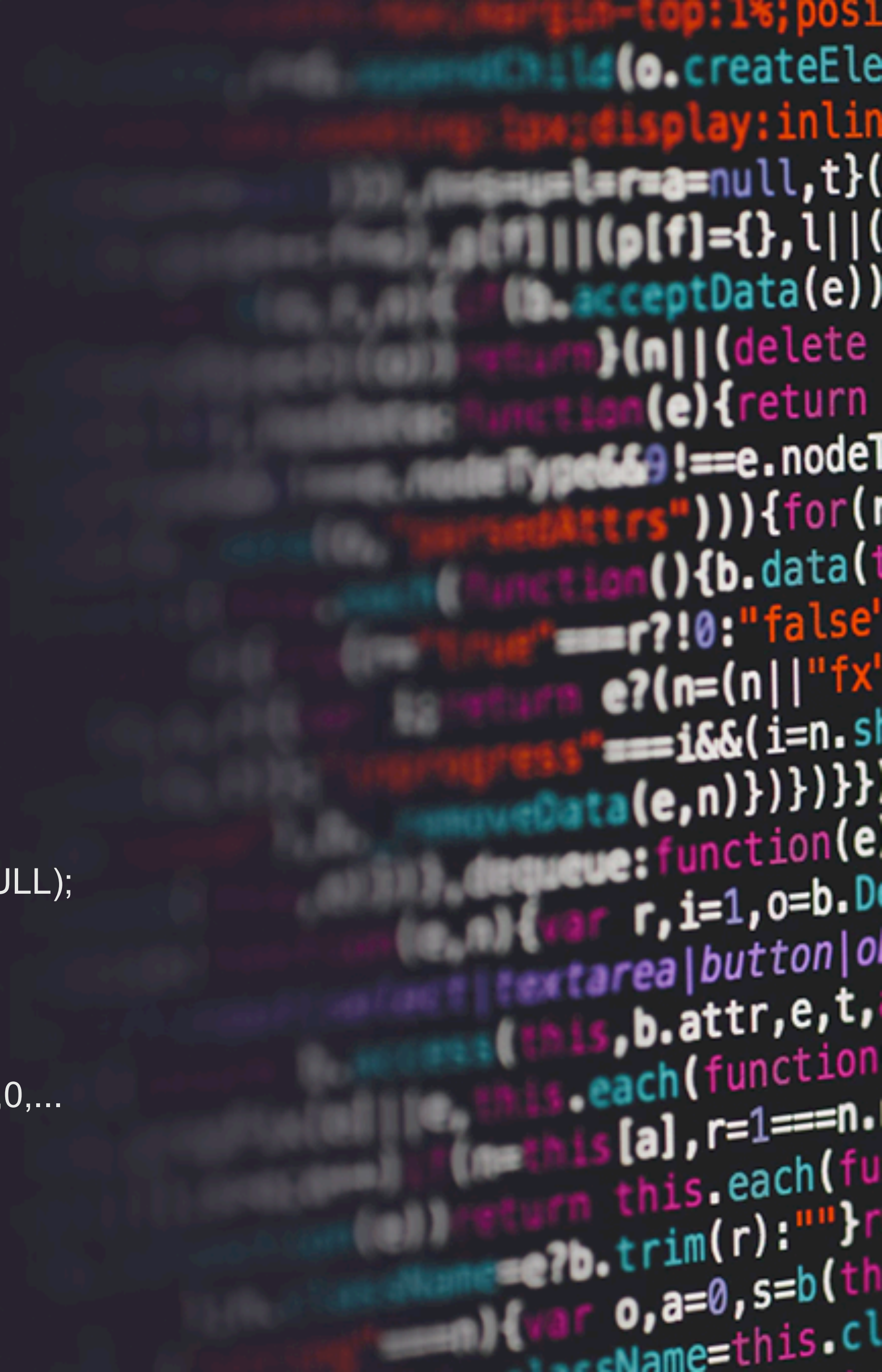
```
#include <CL/opencl.h>

...

const char * KernelSource = "          \\n" \\
"  __kernel void square(               \\n" \\
"    __global float* input,            \\n" \\
"    __global float* output,           \\n" \\
"    const unsigned int count)         \\n" \\
"  {                                   \\n" \\
"    int i = get_global_id(0);         \\n" \\
"    if(i < count)                     \\n" \\
"      output[i] = input[i]*input[i]; \\n" \\
"  }                                   \\n" \\
"                                      \\n";

...
```

OpenCL / main

```
.....

int main(int argc, char** argv)
{
  // Host input data
  float data[DATA_SIZE];

  // Prepare kernel
  int err;
  cl_device_id device_id;
  err = clGetDeviceIDs(NULL,CL_DEVICE_TYPE_GPU,1,&device_id,NULL);
  cl_context context = clCreateContext(0,1,&device_id,NULL,NULL,&err);
  cl_command_queue queue = clCreateCommandQueue(context,device_id,0,&err);
  cl_program prog = clCreateProgramWithSource(context,1,(const char **)&KernelSource,...
  err = clBuildProgram(prog,0,NULL,NULL,NULL,NULL);
  cl_kernel kernel = clCreateKernel(prog,"square",&err);

  // Device io arrays
  cl_mem input = clCreateBuffer(context,CL_MEM_READ_ONLY,sizeof(float)*count,...
  cl_mem output = clCreateBuffer(context,CL_MEM_WRITE_ONLY,sizeof(float)*count,...

  .....
```

OpenCL

```
.....

// Write our data set into the input array in device memory
err = clEnqueueWriteBuffer(queue,input,CL_TRUE,0,sizeof(float)*count,data,0,...

// Set arguments to kernel
err  = clSetKernelArg(kernel,0,sizeof(cl_mem),&input);
err |= clSetKernelArg(kernel,1,sizeof(cl_mem),&output);
err |= clSetKernelArg(kernel,2,sizeof(unsigned int),&count);

// Queue kernel and wait for its execution end
size_t global = count;
err = clEnqueueNDRangeKernel(queue,kernel,1,NULL,&global,NULL,0,NULL,NULL);
clFinish(commands);

// Read back the results from the device to verify the output
float results[DATA_SIZE];
err = clEnqueueReadBuffer(queue,output,CL_TRUE,0,sizeof(float)*count,results,0,...

.....
```



OpenCL

```
.....

 // Process results
 .....

 // Shutdown and cleanup
 clReleaseMemObject(input);
 clReleaseMemObject(output);
 clReleaseProgram(prog);
 clReleaseKernel(kernel);
 clReleaseCommandQueue(commands);
 clReleaseContext(context);
 return 0 ;
}
```

SyCL / headers

```
#include <sycl.hpp>

using namespace cl::sycl;

#define LENGTH (1024)


int main() {

  // prepare input data
  std::vector h_a(LENGTH) ; // a vector
  std::vector h_b(LENGTH) ; // b vector
  std::vector h_c(LENGTH) ; // c vector
  std::vector h_r(LENGTH, 0xdeadbeef) ; // d vector (result)

  ...
```

SyCL / kernel

```
...
{
  // Device buffers
  buffer d_a(h_a) ; buffer d_b(h_b) ; buffer d_c(h_c) ;  buffer d_r(h_d) ;
  queue myQueue ;

  command_group(myQueue, [&]() {

    // data accessors
    auto a = d_a.get_access<access::read>();
    auto b = d_b.get_access<access::read>();
    auto c = d_c.get_access<access::read>();
    auto r = d_r.get_access<access::write>();

    // kernel
    parallel_for( count, kernel_functor( [=](id<> item) {

      int i = item.get_global(0);
      r[i] = a[i] + b[i] + c[i];

    }));

  });

}
```

Feelings about OpenCL

- No physicists will ever write OpenCL code !
- SyCL is the hot topic, since Intel has joined the game, but it requires C++17 and functional programming skills.
- Portability is not Performance Portability...

Global lessons & Recommendations

From IN2P3 group Reprises

Bad News

- A global profiling is mandatory: **CPU times**, **I/O moves**, **Floating Point Errors** (see the Verrou tool)
- GPU et FPGA are not efficient with every and any problem.
- Auto-vectorization does not pollute the code...
  but requires an implicitly adapted code.
- Directives are easy... and limited.
- Code generation and DSLs... complicates the build.
- There is no ideal universal langage.
- Physicists must renounce the confort of double précision and illusion of bit reproductibility.

Good News

- Some transformations will pay off,
  whatever the technology finally chosen :
  - organize your data as structures of arrays (SoA),
  - prefer algorithms embarrassingly parallel,
  - adopt a functional programming style.
- Even in case of deceiving final performance gain,
  trying a new technology will :
  - improves quality and sequential performance,
  - ease the transition towards any other technology
    (because the costly underlying transformations are done)
  - ➢ don't be afraid to choose the wrong technology, go ahead !

Recommendations

What can be the way forward in using accelerators (GPU, FPGA) and HPC in our field ?

1. Apply a global profiling (if the application pre-exists),
2. Reduce the precision as much as it is allowed,
3. Favor algorithms which exposes most parallelism,
4. Review your data structures,
5. Start with non-invasive technologies such as directives.

Is dedicated re-coding necessary, or are abstraction layer libraries like alpaka, kokkos, SYCL the way forward ?

- It depends on the application context and the wished balance between portability, performance, productivty, precision, durability...

A new community

International Workshop on Performance, Portability and Productivity in HPC (P3HPC)

- 1st session at Super Computing '18, Dallas.
- 2nd session at Super Computing '19, Denver.

DOE

* https://performanceportability.org/

Some new metrics

$$\Phi(a,p,H) = \frac{|H|}{\sum_{i \in H} \frac{\min(F_i, B_i \times I_i(a,p))}{P_i(a,p)}}$$

S.J.Pennycook, J.D.Sewall, and V.Lee,
"A metric for performance portability", 2016

Functional programming

not only with Haskell

Learning to Discover : Advanced Pattern Recognition

David Chamont, Reprises, october 2019

FP philosophy

- Tends towards mathematical logic :
  - a variable always refer to the same value
  - given the same arguments, a function always return the same result ;
  - some higher order functions can receive functions as arguments, or return a function as result ;
  - a type algebra eases the transformation of types.
- Some implementation tricks speed up the execution
  - lazy evaluation
  - smart immutable data structures

FP Benefits

- In theory
  - More provable code
  - More readable code
- Practically
  - Avoid any unintended state change
    Michael Feathers : OO makes code understandable by encapsulating moving parts. FP makes code understandable by minimizing moving parts.
  - Easy test
  - Ease parallelism
    no shared states, no problem.

Prehistoric C++

```
void count_lines_in_files( const vector<string> & files, vector<int> & nb_lines )  {
    vector<string>::iterator fileitr ;
    for ( fileitr = files.begin() ; fileitr != files.end() ; ++fileitr ) {
        int line_count = 0 ;
        char c = 0 ;
        ifstream in(*fileitr) ;
        while (in.get(c)) {
            if (c == '\n') { line_count++ ; }
        }
        nb_lines.push_back(line_count) ;
    }
}
```

Historic C++

```
int count_lines( const string & filename )  {
    ifstream in(filename) ;
    typedef istreambuf_iterator<char> ifiterator ;
    return std::count(ifiterator(in), ifiterator(), '\n') ;
}

vector<int> count_lines_in_files( const vector<string> & files )  {
    vector<int> results(files.size()) ;
    std::transform(files.cbegin(), files.cend(), results.begin(), count_lines) ;
    return results ;
}
```

Modern C++

```
auto count_lines( string const & filename ) -> int  {
    ifstream in(filename) ;
    using ifiterator = istreambuf_iterator<char>;
    return std::count(ifiterator(in), ifiterator(), '\n') ;
}

auto count_lines_in_files( vector<string> const & files )  -> vector<int> {
    vector<int> results(files.size()) ;
    std::transform(execution::par,files.cbegin(), files.cend(), results.begin(), count_lines) ;
    return results ;
}
```

Future C++

```
auto open_file( string const & filename )  -> ifstream {
    return ifstream(filename) ;
 }

auto count_lines( ifstream file )  -> int {
    using ifiterator = istreambuf_iterator<char> ;
    return count( ifiterator(in), ifiterator(), '\n') ;
}

auto count_lines_in_files( vector<string> const & files ) -> vector<int> {
    return files | transform(open_file) | transform(execution::par,count_lines) ;
}
```