Summary of Programming for Concurrency and Co-Processors

Jim Pivarski and Vincenzo Innocente

March 29, 2018
FPGAs as Compute Accelerators

- Microsoft Catapult and Bing
  - Improve performance, reduce power consumption
- Reduce the number of von Neumann abstraction layers
  - Bit level operations
- Power only logic cells and registers needed
- Current test devices in LHCb
  - Nallatech PCIe with OpenCL
  - Intel® Xeon®+FPGA
Test Case: RICH PID Algorithm

- Calculate Cherenkov angle $\Theta_c$ for each track $t$ and detection point $D$, not a typical FPGA algorithm
- RICH PID is not processed for every event, processing time is too long!

Calculations:
- solve quartic equation
- cube root
- complex square root
- rotation matrix
- scalar/cross products

Reference: LHCb Note LHCb-98-040
Compare Verilog - OpenCL

- Development time
  2.5 months – 2 weeks
  3400 lines Verilog – 250 lines C

- Performance
  Cube root: x35 – x30
  RICH: x35 – x26

- FPGA resource usage Stratix® V

<table>
<thead>
<tr>
<th>FPGA Resource Type</th>
<th>Verilog RTL</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALMs</td>
<td>88</td>
<td>63</td>
</tr>
<tr>
<td>DSPs</td>
<td>67</td>
<td>82</td>
</tr>
<tr>
<td>Registers</td>
<td>48</td>
<td>24</td>
</tr>
</tbody>
</table>
Andrei Gheata: VecCore: Expressing HEP algos in explicit SIMD types

How do we get it?

- **Auto-vectorization**
  - Compiler optimization converting repetitive scalar instructions (loops) to SIMD code
- **Compiler pragmas**
  - Code annotations persuading the compiler into vectorizing
  - OpenMP, CilkPlus
  - May not preserve exact scalar behavior
- **SIMD libraries**
  - VCL, Vc, UME::SIMD, VecCore
  - Explicit programming using specific vector types and operations
- **Compiler intrinsics**
  - Built-in inline compiler functions accessing architecture-specific vector instructions
- **Assembly**
  - The really low-level stuff on top of HW implementation

```c
float a[N], b[N], c[N];
for (int i = 0; i < N; i++)
a[i] = b[i] * c[i];

float a[N], b[N], c[N];
#pragma omp simd
#pragma ivdep
for (int i = 0; i < N; i++)
a[i] = b[i] * c[i];

#include <VecCore/VecCore>
using Float_v = 
  backend::VcVector::Float_v;
Float_v a = b * c;

#include <x86intrin.h>
__m256 a, b, c;
a = _mm256_mul_ps(b, c);
asm volatile("vmulps %ymm1, %ymm9");
```
namespace vecCore {

    template <typename T> struct TypeTraits;
    template <typename T> using Mask = typename TypeTraits<T>::MaskType;
    template <typename T> using Index = typename TypeTraits<T>::IndexType;
    template <typename T> using Scalar = typename TypeTraits<T>::ScalarType;

    // Vector Size
    template <typename T> constexpr size_t VectorSize();

    // Get/Set
    template <typename T> Scalar<T> Get(const T &v, size_t i);
    template <typename T> void Set(T &v, size_t i, Scalar<T> const val);

    // Load/Store
    template <typename T> void Load(T &v, Scalar<T> const *ptr);
    template <typename T> void Store(T const &v, Scalar<T> *ptr);

    // Gather/Scatter
    template <typename T, typename S = Scalar<T>>
    T Gather(S const *ptr, Index<T> const &idx);
    template <typename T, typename S = Scalar<T>>
    void Scatter(T const &v, S *ptr, Index<T> const &idx);

    // Masking/Blending
    template <typename H> bool MaskFull(H const &mask);
    template <typename H> bool MaskEmpty(H const &mask);

    template <typename T> void MaskedAssign(T &dst, const Mask<T> &mask, const T &src);
    template <typename T> T Blend(const Mask<T> &mask, const T &src1, const T &src2);
}

https://github.com/root-project/veccore/blob/master/doc/API.md
Andrei Gheata: VecCore: Expressing HEP algos in explicit SIMD types

Internal vectorization example: the Mandelbrot set

iterate \( f(z) = z^2 + c \) \( N \) times and check if \( z \) diverges

```cpp
template<typename T>
void mandelbrot(T xmin, T xmax, T ymin, T ymax, size_t max_iter,
    T dx = (xmax - xmin) / T(nx);
    T dy = (ymax - ymin) / T(ny);
    for (size_t i = 0; i < nx; ++i) {
        for (size_t j = 0; j < ny; ++j) {
            size_t k = 0;
            T x = xmin + T(i) * dx, cr = x;
            T y = ymin + T(j) * dy, ci = y;
            do {
                x = zr*zr - zi*zi + cr;
                y = 2.0 * zr*zi + ci;
                zr = x;
                zi = y;
            } while (++k < max_iter & !image[ny*j+i] = k;
        }
    }
```
**Martin Durant:** Dask: distributed computing for scientific Python

**DASK: HOW TO SCALE UP WITH A MINIMUM OF HASSLE**

Run dask on your laptop, or on a large cluster: just specify the scheduler address.

```
In [1]: import dask.distributed
    client = dask.distributed.Client('dask-scheduler:8786')
    client
```

```
Out[1]:

```

@dask.delayed
def f(x, y):
    do_thing_with_inputs
    return output
```
Martin Durant: Dask: distributed computing for scientific Python
Martin Durant: Dask: distributed computing for scientific Python
Matti Kortelainen: Parallelized tracking algorithms

Challenges for massive parallelism

- We have small matrices ($5 \times 5 / 6 \times 6$)
  - Typical approach for matrix operation vectorization does not scale for us
    - $*$ = vectorize the calculations of a single matrix operation
- Pattern recognition problem is combinatorial with dynamic output size
  - Need to create new track candidates on the fly
  - In general need dynamic memory allocations
- Traditional algorithms have lots of control logic
  - Harmful for SIMD
- Data transfers from CPU/host memory to accelerator(s)
  - For short computations data transfer time may dominate
- E.g. in Intel Skylake the use of AVX/AVX-512 instructions slows down the frequency of the core
  - So to benefit from AVX, significant portion of computational kernel has to be vectorized
Matti Kortelainen: Parallelized tracking algorithms

**Triplet Propagation Algorithm**

- First create doublets from hits of pairs
- Take a third layer and propagate only the generated doublets
- Consider a fourth layer and propagate triplets
- Store found quadruplets and start from another pair of layers
Matti Kortelainen: Parallelized tracking algorithms

Cellular Automaton (CA)

- The CA is a track seeding algorithm designed for parallel architectures
- It requires a list of layers and their pairings
  - A graph of all the possible connections between layers is created
  - Doublets aka Cells are created for each pair of layers (compatible with a region hypothesis)
  - Fast computation of the compatibility between two connected cells
  - No knowledge of the world outside adjacent neighboring cells required, making it easy to parallelize
- However this is not a static problem, not at all...

Now a library, v0.1
https://github.com/HEP-SF/TrickTrack