The Future of Accelerator Programming in C++

Sebastian Schaetz

Biomedizinische NMR Forschungs GmbH
at the Max Planck Institute for Biophysical Chemistry, Goettingen

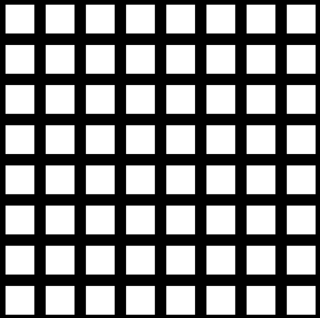
C++Now 2014

May 17th, 2014

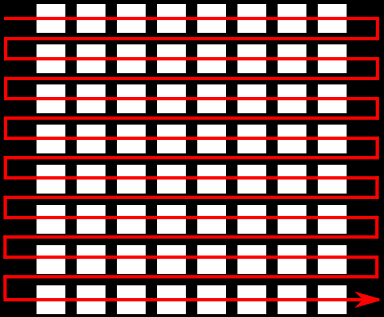
About Me

- developing C++ libraries for strange architectures
- working at medical imaging research institute
- developing and maintaining a multi-GPU signal processing program
- supporting scientists that prototype signal-processing algorithms
- supporting scientists that develop large simulations

Accelerator Programming



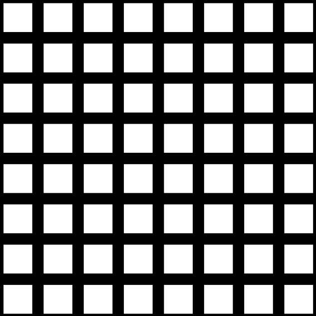
Accelerator Programming



CPU

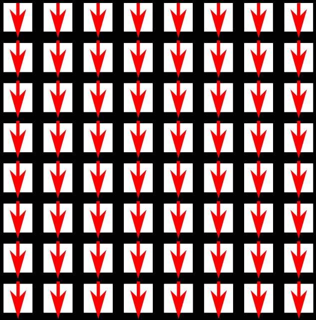
```
void scale(float* A,
    const int X, const
    int Y)
{
   int i=0;
   while (i<X*Y) {
        A[i] *= 42.;
        i++;
   }
}

scale(A, 8, 8);
```

Accelerator Programming



Accelerator Programming

```
__global__ void
    scale(float* A)
{
   int x = threadIdx.x;
   int y = blockIdx.x;
   int X = blockDim.x
   A[y*X+x] *= 42.0;
}

scale<<<8, 8>>>(A);
```



Accelerator

Accelerator Hardware Model

- co-processor
- hierarchical configuration of trimmed-down "cores"
  - thread: ALU
  - warp: shared instruction (SIMD)
  - block: local synchronization
  - grid/kernel: problem domain, global synchronization
- large number of registers (many concurrent contexts)
- dedicated memory with high memory bandwidth
- programmable DMA engine
- concurrent command dispatch

Fatahalian, K. (2010). From Shader Code to Teraflop: How Shader Cores Work. Beyond Programmable Shading Course. ACM SIGGRAPH, New York, NY, USA.

Accelerators and Vendors

- GeForce, Quadro, Tesla, Tegra[2] (Nvidia)
- Radeon, FirePro, APU[2], R-Series[2] (AMD)
- Xeon Phi (Intel)
- Mali[2] (ARM)
- Adreno[2] (Qualcomm)
- PowerVR[2] (Imagination Technologies)

- and FPGAs from Altera and Xilinx

[2] shared memory

Tools

- automated-tools: OpenMP, OpenACC
- (active) libraries
- do-it-yourself: OpenCL, CUDA

Rocki, K., Burtscher, M., & Suda, R. (2014). The Future of Accelerator Programming: Abstraction, Performance or Can We Have Both?

Veldhuizen, T., & Gannon, E. (1998). Active libraries: Rethinking the roles of compilers and libraries.

Library	CUDA	OpenCL	Other	Type
Thrust	X		OMP, TBB	header
Bolt		X	TBB, DX11	link
VexCL	X	X		header
Boost.Compute		X		header
C++ AMP		X	DX11	compiler
SyCL		X		compiler
ViennaCL	X	X	OMP	header
SkePU	X	X	OMP, seq	header
SkelCL		X		link
HPL		X		link
CLOGS		X		link
ArrayFire	X	X		link
CLOGS		X		link
hemi	X			header
MTL4	X			header
Kokkos	X		OMP, PTH	link
Aura	X	X		header

Programming Accelerators

- Coordination
- Computation

Programming Accelerators

- Coordination
  - concurrency
  - memory management
- Computation

Programming Accelerators

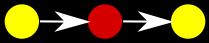
- Coordination
  - concurrency
  - memory management
- Computation
  - parallel primitives
  - custom accelerator functions
  - numerical analysis
  - performance portability
  - kernel-space exploration

Concurrency

overlap accelerator functions and transfer or multiple accelerator functions:

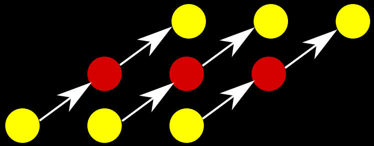
- asynchronous memory transfer
- asynchronous accelerator function invocation
- synchronization of memory transfer and accelerator functions

Concurrency



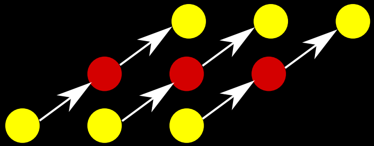
- dependency graph

Concurrency



- dependency graph
- object indicating independent operations

Concurrency



- dependency graph
- object indicating independent operations
- future, promise

Memory Management

- Implicit Memory Management
- Explicit Memory Management
  - containers that represent memory
  - functions to transfer memory

Memory Management: Explicit

```
device d(0);
device_array<float> A(size, d);
std::vector<float> B(size, 42.);
copy(B.begin(), B.end(), A.begin());
```

Memory Management: Explicit

```
device d(0);
device_array<float> A(size, d);
std::vector<float> B(size, 42.);
copy(B.begin(), B.end(), A.begin());

feed f(d);
copy_async(B.begin(), B.end(), A.begin(), f);
```

- synchronous and asynchronous
- difficult zero-copy

Memory Management: Explicit (cont.)

- single- or multi-dimensional through index and extent/bounds (N3851)
- single-accelerator or multi-accelerator container (VexCL)
- copy interface: assignment operator, iterator based, range based, special
- support copy from non-contiguous memory
- functions to fill containers (constant, identity)

Memory Management: Convenient

```
device d(0);
device_array<float> A(size, d);
A[0] = 42.;
cout << A[0];
```

Memory Management: Convenient

```
device d(0);
device_array<float> A(size, d);
A[0] = 42.;
cout << A[0];
```

- write and read every time
- track modifications and do bulk reads and writes
- replace array subscript with function call operator
- lazy copy
- accessor:
  - map memory to host (possible zero-copy)
  - scoped
  - state intentions (read, write, modify)

Memory Management: Reversed

```
std::vector<float> B(size, 42.);
array_view<float, 1> A(size, B.begin());
B[15] = 43.0;
A.refresh();
```

Memory Management: Reversed

```
std::vector<float> B(size, 42.);
array_view<float, 1> A(size, B.begin());
B[15] = 43.0;
A.refresh();
```

- memory copy on access (lazy)
- caching possible
- zero-copy possible

Computation

Computation: Limitations

- OpenCL
  - accelerator function string compiled at runtime
  - no C++ support (no templates)
- CUDA
  - accelerator function decorator __device__
  - only Nvidia hardware
- C++
  - ??? (get body of function as a string)

Parallel Primitives

- skeletons or higher-order functions
- Technical Specification for C++ Extensions for Parallelism (N3960)

```
std::vector<float> A(size);
using namespace std::experimental::parallel;
sort(par, v.begin(), v.end());
```

- more sensible:

```
std::vector<float> A(size);
device d(0);
feed f(d);
accelerator_policy ap(f);
sort(ap, v.begin(), v.end());
```

Parallel Primitives

- algorithms for both host and accelerator
- lambda or function objects to specify custom operator

```
BOOST_COMPUTE_FUNCTION(int, add_four, (int x),
{
  return x + 4;
});

boost::compute::transform(vector.begin(),
   vector.end(), vector.begin(), add_four);
```

Parallel Primitives: Fancy Iterators

```
device_array<int> A(3);
device_array<char> B(3);

auto first =
    make_zip_iterator(make_tuple(A.begin(),
    B.begin()));
auto last =
    make_zip_iterator(make_tuple(A.end(),
    B.end()));

maximum< tuple<int,char> > binary_op;
tuple<int,char> init = first[0];
reduce(first, last, init, binary_op);
```

Writing Accelerator Functions

- backend DIY-style (CUDA, OpenCL)
- lambda expression as argument to parallel primitives (Boost.Compute, Thrust, Bolt)

Writing Accelerator Functions (C++AMP)

- as lambda passed to `parallel_for_each`

```
int aCPP[] = {1, 2, 3, 4, 5};
int resCPP[size];
array_view<const int, 1> a(size, aCPP);
array_view<int, 1> res(size, resCPP);
res.discard_data();

parallel_for_each(
  // compute domain (number of threads)
  res.extent,
  [=](index<1> idx) restrict(amp)
  {
    res[idx] = a[idx] * 42.;
  }
);
```

Writing Accelerator Functions (HSL)

- DSL through macro-based instructions

```
void dp(Array<float> v1, Array<float> v2,
   Array<float> ps)
{
  Int i;
  Array<float, 1, Local> sharedM(128);
  sharedM[lidx] =  v1[idx] * v2[idx];
  barrier(LOCAL);
  if_(lidx == 0 ) {
    ps[gidx] = sharedM[0];
    for_( i = 1, i < Witems, i++ ) {
      ps[gidx] += sharedM[i];
    }
  }
}
eval(dp).global(N).local(Witems)(v1, v2, ps);
```

Writing Accelerator Functions (VexCL)

- Expression Templates

```
vex::FFT<double, cl_double2> fft(ctx, n);
vex::FFT<cl_double2, double> ifft(ctx, n,
    vex::fft::inverse);

vex::vector<double> rhs(ctx, n), u(ctx, n),
    K(ctx, n);

u = ifft( K * fft(rhs) );
```

- Kernel generation with Boost.Proto from existing code (limited but useful)

Numerical Analysis (ViennaCL)

- LU, QR, Cholesky factorization, singular values, Hessenberg
- inverse, matpow, rank, det
- image convolution
- iterative solvers:
  - conjugate gradient
  - stabilized CG
  - generalized minimum residual
- preconditioner
- eigenvalue computation
- QR factorization
- mixed-precision conjugate gradient

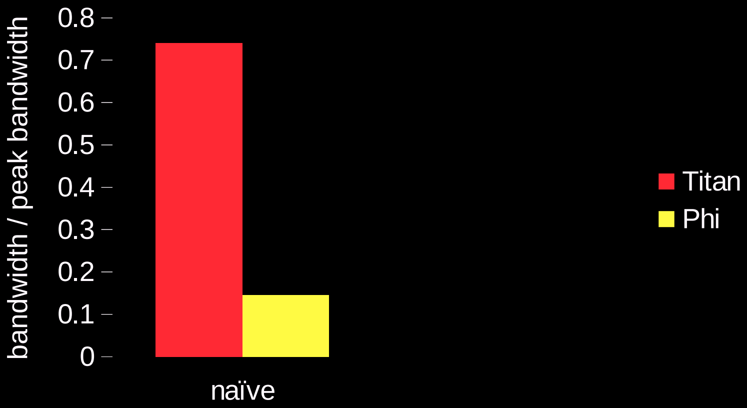
Aura

```
aura::initialize();
aura::device d(0);
aura::feed f(d);

aura::module m = aura::create_module_from_file(
    kernel_file, d, AURA_BACKEND_COMPILE_FLAGS);
aura::kernel k = aura::create_kernel(m, "scale");

int x = 128; int y = 128; int z = 64;
std::vector<float> hv(product(bounds(x, y, z)), 42.);
aura::device_array<float> dv(bounds(x, y, z), d);

aura::copy(dv, hv, f);
aura::invoke(k, mesh(y, x), bundle(x),
    args(dv.begin_ptr(), .1), f);
aura::copy(hv, dv, f);
aura::wait_for(f);
```

Aura - Kernel

```
AURA_KERNEL void scale(AURA_GLOBAL float*
    data, float scalar)
{
  int id = get_mesh_id();
  int s = get_mesh_size();
  for (int i=0; i<64; i++) {
    data[id] = scalar * data[id];
    id += s;
  }
}
```

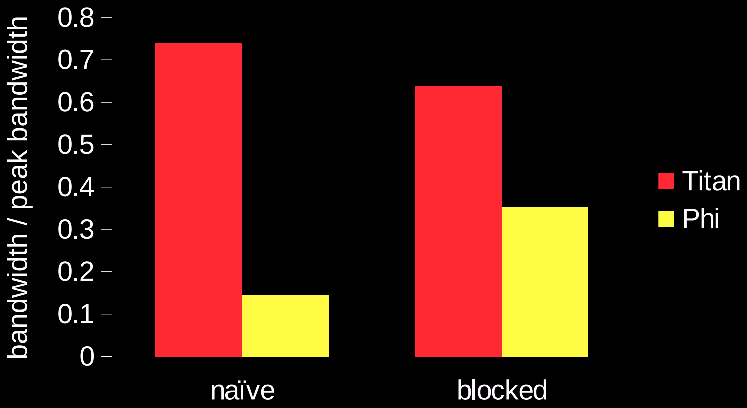
Yay Benchmarks!



Aura - Kernel

```
AURA_KERNEL void peak_copy(
    AURA_GLOBAL float* data, float scalar)
{
  const int bsize = 16;
  const int mult = 64;
  int id = (get_mesh_id() / bsize)*bsize*mult +
      get_mesh_id() % bsize;
  int s = get_mesh_size();
  for (int i=0; i<64; i++) {
      data[id + i * bsize] =
          scalar * data[id + i * bsize];
  }
}
```

Again: Yay Benchmarks!



Kernel Space Exploration (Obsidian)

- Haskell library for accelerators
- "raise the level of abstraction [...] and still give the programmer control over the details relevant to kernel performance"
- making available hardware hierarchy in an abstraction:
  - warp (divergence)
  - block (synchronization, communication, cache memory)
  - kernel (loop-nests, global synchronization)

FSvensson, B. J., Sheeran, M., & Newton, R. R. (2014). A Language for Nested Data Parallel Design-space Exploration on GPUs.

Kernel Space Exploration (Halide)

- DSL for image processing
- splitting of what to compute and how to compute

```
Var x, y;
Func gradient("gradient_tiled");
gradient(x, y) = x + y;

Var x_out, x_in, y_out, y_in;
gradient.split(x, x_out, x_in, 2);
gradient.split(y, y_out, y_in, 2);
gradient.reorder(x_in, y_in, x_out, y_out);

Image<int> output = gradient.realize(4, 4);
```

Ragan-Kelley, J. et al. (2013, June). Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In PLDI 2013 (pp. 519-530). ACM.

Kernel Space Exploration

- copious-parallelism
- auto-tuning

Conclusion

- abstractions that allow kernel-space exploration
- elegant way to write kernel code (C++ AMP like lambdas for all platforms)
- monolithic set of libraries that build upon each other and help DSP programmers, applications programmers and scientific programmers to solve their particular problems
- consider future hardware

Contact

Code: https://www.github.com/sschaetz/aura

Blog: http://www.soa-world.de/echelon

Twitter: @sebschaetz

E-Mail: seb.schaetz@gmail.com