GridRipper >> Lattice Template Library

Máté Ferenc Nagy-Egri

HAS Wigner RCP – GPU-Lab administrator

HAS Wigner RCP – Young reseatcher, VIRGO group

Eötvös Loránd University of Sciences – PhD student

The Future of Many-Core Computation in Science 2014

Budapest, Hungary

TABLE OF CONTENTS

MOTIVATION

All scientific calculations are motivated by a very concrete goal
Home-brew simulation software is highly specialized

The development time of GPU toolchains often outlast that of the HW involved

There are only a handful of libraries that aim at general scientific calculations (FFT, BLAS)

Incorporating these into toolchains more often than not involves data reformatting

The entry cost to GPGPU is still too high

Not many people get their hands dirty by handwriting kernels

If they do, they are content by writing highly specialized code

Generic scientific code is rare

GRIDRIPPER

C++ template library aiming at solving PDE on lattices
Generic in the sense of the equation to solve

Main motivation is general relativity /cosmology calculations
Sofware design is partly subordinated to the specific use-case

Makes efficient use of C++
Cluster support implemented

Features online data analysis

Precise calculations are computationally intensive

GPGPU acceleration became a neccessity

How to GPU accelerate a template library?

Well... good luck with that.

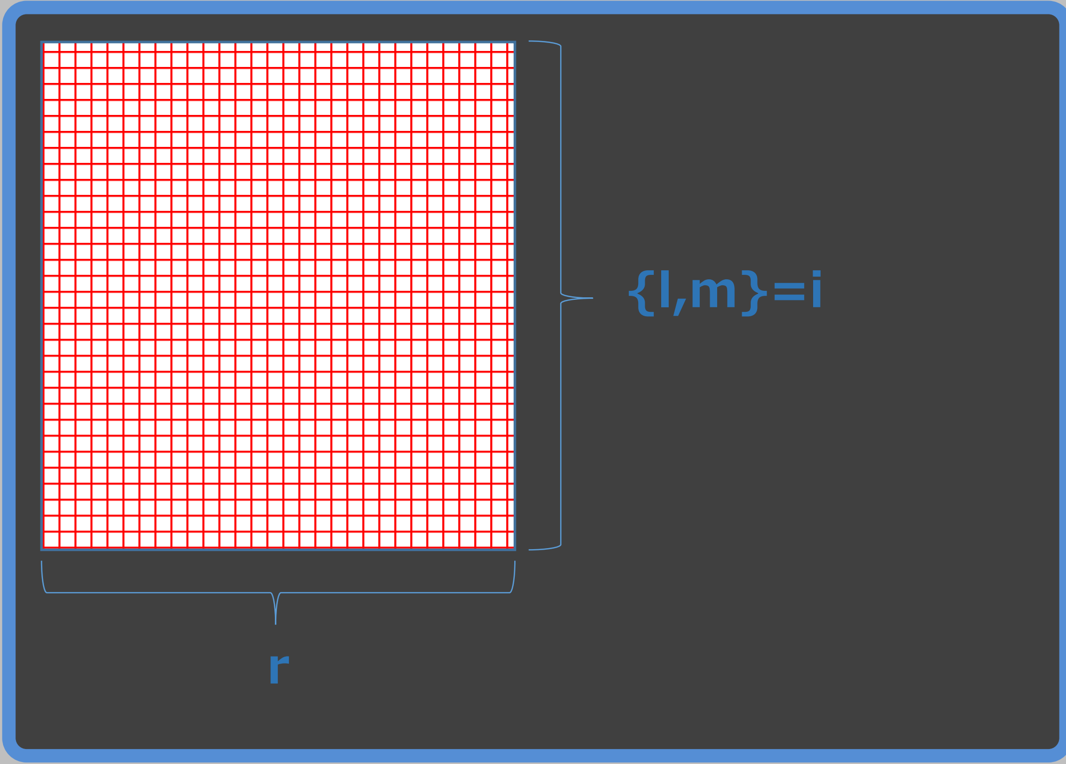
GRIDRIPPER

What causes the computation intensiveness in our use-case of GridRipper?

Field variables are are expanded on the basis of spherical harmonics

The 3D real space is reduced to 1D where every coordinate holds a vector of spherical coefficients

In the spherical expansion, multiplication and division by field variables become costly operations



DEVELOPMENT EXPERIENCE

As proof of concept: solve PDE with monolithic shader/kernel
Joint RK4, RHS evaluation, numerical tricks/corrections

The code became virtually debug immune
Numerical instability was introduced at a hidden point

Months of development time was wasted on trying to find the source of the problem

Ultimately, the code was trashed and only the experience and a lesson learned remained

Code must be broken down to smaller modules that are easier to debug

Altering parts of the code must not require GPGPU knowledge

The field equations themselves have to be generated and not handwritten

MUCH of the burden has to be placed on the library developer. (That is me)

DEVELOPMENT EXPERIENCE

EXPRESSION TEMPLATES

DEVELOPMENT EXPERIENCE

There is an excellent tool at hand that is capable of generating code:
GCC / MSVC / Clang

Expression templating a C++ metaprogramming technique that is capable of concatenating operations

Operations and concatenation itself can be tested seperately

Code generation happens statically, at compile time
No runtime penalty, infact...

Implementing Expression Templates is tricky

It is a powerful tool yet it is not almighty

Expression templating deep recursions can bring a compiler to it's knees

Let's see how it actually looks like

```
template <typename T>
class Expression
{
public:

   typedef std::valarray<double> container_type;
   typedef std::size_t           size_type;
   typedef double                value_type;

   size_type  size() const {
return static_cast<T const&>(*this).size(); }
   value_type operator[](size_type i) const {
return static_cast<T const&>(*this)[i];     }

   operator T&() {
return static_cast<T&>(*this); }
   operator T const&() const {
return static_cast<const T&>(*this); }
};
```

```
class Vector : public Expression<Vector>
{
public:

   value_type& operator[](size_type i)       {
return _data[i]; }
   value_type  operator[](size_type i) const {
return _data[i]; }
   size_type   size() const { return _data.size(); }

   Vector(size_type n) : _data(n) {}

   template <typename T>
   Vector(VecExpression<T> const& vec) {
      T const& v = vec;
      _data.resize(v.size());

      for (size_type i = 0; i != v.size(); ++i)
         _data[i] = v[i];
   }

private:

    container_type _data;
};
```

```
template <typename E1, typename E2>
class Sum : public Expression<Sum<E1, E2>>
{
public:
    Sum(Expression<E1> const& u, Expression<E2> const& v) : _u(u), _v(v) { assert(u.size() == v.size()); }

   size_type   size()                       const { return _v.size(); }
   value_type  operator[](size_type i) const { return _u[i] + _v[i]; }

private:
    E1 const& _u;
    E2 const& _v;
};

template <typename E1, typename E2>
Sum<E1, E2> const operator+(Expression<E1> const& u, Expression<E2> const& v) { return Sum<E1, E2>(u, v); }
```

```
int main()
{
   Vector a(LENGTH);
   Vector b(LENGTH);
   Vector r(LENGTH);

   r = 2.0 * (a + b);

   return 0;
}
```

```
Vector(Expression<Scale<Sum<Vector,Vector>>> const& v)
{
      Scale<Sum<Vector,Vector>> const& vec = v;
      _data.resize(vec.size());

      for (size_type i = 0; i != vec.size(); ++i)
         _data[i] = 2.0 * (a[i] + b[i]);
}
```

LATTICE TEMPLATE LIBRARY

The Standard Template Library holds great building blocks for higher level algorithms

The iterator scheme became a major success
Let’s try to adopt it to a wider range of applications

Iterators inherently cannot abstract N-dimensional data access
Some alternative is required

STD containers have template parameters that specialize their behavior
Lattices could also make use of this approach

All lattices occupy N-dimensions

All lattices have a set of boundary conditions

All lattices ultimately will be transformed by a series of operations

Let’s see how it could look like

LATTICE TEMPLATE LIBRARY

```
int main()
{
   ltl::lattice< 1, ltl::periodic_boundary> my_latt;

   for(auto elem : my_latt)
      *elem = elem.index();

   ltl::transform(my_latt.extent(),
                  [&](ltl::lattice< 1, ltl::periodic_boundary>::index idx)
   {
      my_latt[idx] = (*(idx+1) - *idx) / idx.index();
   });

   return 0;
}
```

Rapid prototyping of lattice calculations

Implementation details are implicit to the user

Cluster and GPGPU capable

How can such a class generate kernel code?

TECHNOLOGIES

We need compilers that understand host-side AND device-side code

C++AMP is a Microsoft invented extension to C++, that does just this

SYCL is a Khronos extension to OpenCL which is also capable of this

CUDA is an NVIDIA invention that also can do Expression Templates, is well established, but is a closed platform

Actively collaborating with Microsoft in getting the most efficient proof of concept running

Arranged access to SYCL compiler providing competition to C++AMP

Near future plans involve implementing the GridRipper motivated calculation

Farther plans include creating a simple Lattice Template Library

Thank you for you attention!