CUDA Made Simple

The Thrust Library

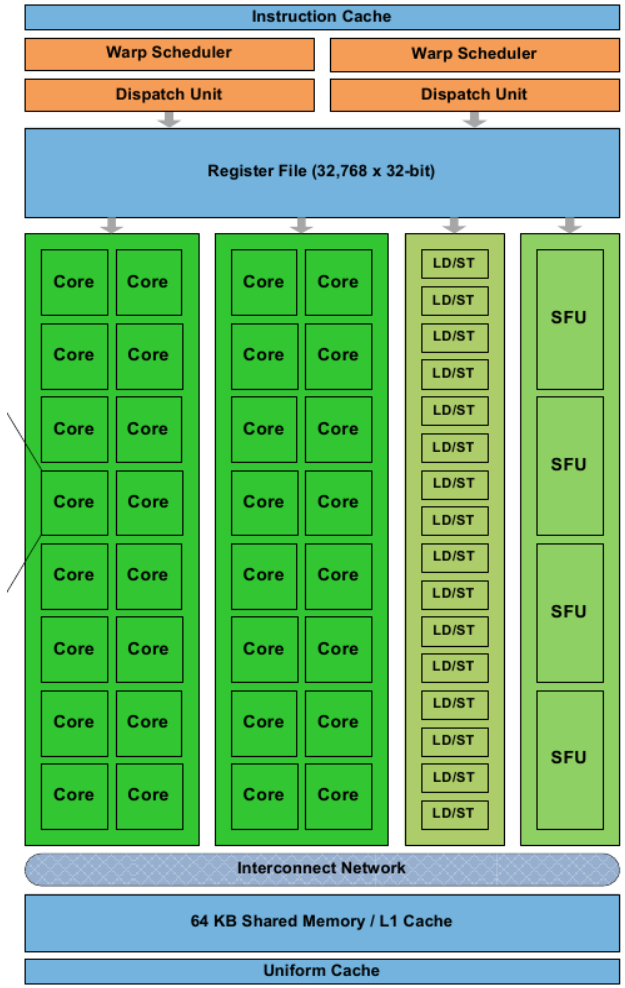
Mario Mulansky

University of Potsdam

March 15, 2011

About GPGPU

- **G**eneral **P**urpose computation on **G**raphics **P**rocessing **U**nits
- Employ the parallel power of graphic cards for numerical simulations
- Great speed ups (20x – 100x) possible
- Several technics:
  - ATI Stream
  - **Nvidia CUDA**
  - OpenCL (universal)
- With Thrust, we can use the power of CUDA in an elegant and easy way



Fermi Streaming Multiprocessor (SM)

A First Example

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

int main(void)
{ // generate 16M random numbers on the host
  thrust::host_vector<int> h_vec( 16*1024*1024 );
  thrust::generate(h_vec.begin(), h_vec.end(), rand);
  // transfer data to the device
  thrust::device_vector<int> d_vec = h_vec;
  thrust::sort(d_vec.begin(), d_vec.end()); // sort data on the device
  // transfer data back to host
  thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
}
```

A First Example

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

int main(void)
{ // generate 16M random numbers on the host
  thrust::host_vector<int> h_vec( 16*1024*1024 );
  thrust::generate(h_vec.begin(), h_vec.end(), rand);
  // transfer data to the device
  thrust::device_vector<int> d_vec = h_vec;
  thrust::sort(d_vec.begin(), d_vec.end()); // sort data on the device
  // transfer data back to host
  thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
}
```

C-version:

```
int* h_vec = new int[SIZE];
for( i=0 ; i<SIZE ; ++i ) h_vec[i] = rand();
int* d_vec; cudaMalloc( (void**) &d_vec , SIZE*size_of(int) );
cudaMemcpy( d_vec, h_vec, SIZE*size_of(int), cudaMemcpyHostToDevice);
// sorting has to be implemented, not shown here
cudaMemcpy(h_vec, d_vec, SIZE*size_of(int), cudaMemcpyDeviceToHost);
```

About Thrust

Objective

- Productivity
- Generic programming
- High performance with minimal effort
- Integration with CUDA C/C++ code

Ideas

- Implement STL-like algorithms for CUDA
- High level programming interface to GPU devices
- Hide parallelization details
- Easy switching between GPU, CPU, OpenMP

What is Thrust?

- C++ template library for CUDA
  - Mimics Standard Template Library (STL)
- Containers
  - `thrust::host_vector< T >`
  - `thrust::device_vector< T >`
- Algorithms
  - `thrust::sort()`
  - `thrust::transform()`
  - `thrust::reduce()`
  - `thrust::for_each()`
  - etc.

If you know about STL algorithms, using Thrust is straightforward.

Containers

- Replace C-arrays, compatible with STL containers
- Increase readability and re-usability of code
- Hide CUDA memory allocation methods (`cudaMalloc`, `cudaFree`)
- Basis for CUDA parallelization

```
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);
//C: int* h_vec = new int[2];

// copy host vector to device
thrust::device_vector<int> d_vec = h_vec;
//C: int* d_vec; cudaMalloc( (void**) &d_vec , 2*size_of(int) );
//   cudaMemcpy( d_vec, h_vec, 2*size_of(int),  cudaMemcpyHostToDevice);

// manipulate device values from the host
d_vec[0] = 13;
d_vec[1] = 27;

std::cout << "sum: " << d_vec[0] + d_vec[1] << std::endl;
// vector memory automatically released w/ free() or cudaFree()
```

Such element access is very very very slow.

Iterators

- Abstraction of pointers to array elements
- A pair of iterators defines a sequence

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

d_vec.begin(); // returns iterator at first element of d_vec
d_vec.end(); // returns iterator one past the last element of d_vec

// [begin, end) pair defines a sequence of 4 elements
```

Iterators

- Abstraction of pointers to array elements
- A pair of iterators defines a sequence

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

d_vec.begin(); // returns iterator at first element of d_vec
d_vec.end(); // returns iterator one past the last element of d_vec

// [begin, end) pair defines a sequence of 4 elements

thrust::device_vector<int>::iterator begin = d_vec.begin();
thrust::device_vector<int>::iterator end = d_vec.end();
//C: int* begin = &d_vec[0]; int* end = begin+4;

int length = end - begin; // compute size of sequence [begin, end)
end = d_vec.begin() + 3; // define a sequence of 3 elements

*begin = 13;          // same as d_vec[0] = 13;
int temp = *begin; // same as temp = d_vec[0];
begin++;              // advance iterator one position
*begin = 25;          // same as d_vec[1] = 25;
```

Algorithms

Sequences defined by iterators are supplied to algorithms.
For example:

```
// sort vector d_vec given by the sequence [ begin , end )
thrust::sort( d_vec.begin() , d_vec.end() );

// calculates the sum of the sequence [ begin , end )
int sum = thrust::reduce( d_vec.begin() , d_vec.end() );

thrust::host_vector<int>::iterator begin = h_vec.begin();
thrust::host_vector<int>::iterator end = h_vec.end();
thrust::generate( begin , end , rand );
//equivalent to: while( begin != end) *begin++ = rand();
```

Implementation of the algorithms is hidden. E.g. `sort` for device vectors is a rather complicated, parallelized CUDA-radix sort.
You don't have to deal with details of parallelization.

Host/Device Calculations

- Thrust tracks memory space and distributes algorithms accordingly onto CPU or GPU

```
// initialize random values on host
thrust::host_vector<int> h_vec(1000);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// copy values to device
thrust::device_vector<int> d_vec = h_vec;

// compute sum on host (CPU)
int h_sum = thrust::reduce(h_vec.begin(), h_vec.end());

// compute sum on device (GPU)
int d_sum = thrust::reduce(d_vec.begin(), d_vec.end());

cout << h_sum << " = " << d_sum << endl;
```

- Allows to easily switch between CPU/GPU calculations

C++ Template Programming

Templates are used to implement type-independent algorithms

```
// function template to add numbers (type T is variable)
template< typename T >
T add(T a, T b)
{
  return a + b;
}

// add integers
int x = 10; int y = 20; int z;
z = add<int>(x,y); // type of T explicitly specified
z = add(x,y); // type of T determined automatically

// add floats
float x = 10.0f; float y = 20.0f; float z;
z = add<float>(x,y); // type of T explicitly specified
z = add(x,y); // type of T determined automatically
```

Function Objects (Functors)

Functors are classes that can be called like functions.

```
// templated functor to add numbers
template< typename T >
struct add
{
  T operator()(T a, T b)
  {
    return a + b;
  }
};

int x = 10; int y = 20; int z;
add<int> add_functor; // create an add functor for T=int
z = add_functor(x,y); // invoke functor on x and y

float x = 10; float y = 20; float z;
add<float> add_functor; // create an add functor for T=float
z = add_functor(x,y);// invoke functor on x and y
```

Generic Algorithm

A generic algorithm applies an operation to a sequence of values.
Using C-style arrays an implementation could look as follows:

```
// apply function f to sequences x, y and store result in z
template <typename T, typename Function>
void transform(int N, T * x, T * y, T * z, Function f)
{
  for (int i = 0; i < N; i++)
    z[i] = f(x[i], y[i]);
}

int N = 100;
int x[N]; int y[N]; int z[N];

add<int> func;    // add functor for T=int

transform(N, x, y, z, func); // compute z[i] = x[i] + y[i]

transform(N, x, y, z, add<int>()); // equivalent
```

Implementation of loop independent of container and operation.

Thrust Algorithms

Thrust provides many standard algorithms and a number of general operations (functors)

- Transformations
- Reductions
- Sorting

- plus, minus, multiplies, divides
- minimum, maximum
- negate, modulus, absolute_value
- equal_to, greater, less

Example: negation

Compute $\vec{x} = -\vec{x}$ (inplace)

```
template< typename T >
struct negative {

  void operator()( T &x ) const {
    x = -x;
  }

};

thrust::for_each( x.begin() , x.end() , negative<double>() );
```

The `for_each`-call is equivalent to:

```
while( x_begin != x_end )
  f( x );
```

where `f(x)` is the ()-Operator of `negative` which does `x = -x`

Example: saxpy

Compute $\vec{z} = a\vec{x} + \vec{y}$.

```
template <typename T>
struct saxpy_functor
{
  const T a;

  saxpy_functor(T _a) : a(_a) {}

  // tell CUDA that the following code can be executed on the CPU and the GPU
  __host__ __device__
  T operator()(const T& x, const T& y) const {
    return a * x + y;
  }
};

thrust::transform( x.begin() , x.end() , y.begin() , z.begin() ,
                   saxpy_functor< double >( a ) );

// while( x_begin != x_end ) *z_begin++ = f( *x_begin++ , *y_begin++ );
```

Example: Reduction

Calculate the product of a sequence $p = \prod x_i$

```
p = thrust::reduce( x.begin() , x.end() , 1 , multiplies< double >() );
```

For parallelization, the binary operation has to be associative and commutative!

Calculate the norm of a vector $n = \sum x_i^2$

```
template <typename T>
struct square_functor {

  __host__ __device__
  T operator()( const T& x ) const {
    return x*x:
  }

};

thrust::transform_reduce( x.begin(), x.end(), square_functor< double >(),
                          0.0 , thrust::plus< double >() );
```

Recap Thrust Algorithms

- Generic (type and operator independent)
- Automatically dispatched to CPU or GPU based on used container type (at compile time)
- Compatible with STL containers
- Easy to switch between CPU and GPU computation, basically by changing one line of code:

```
typedef thrust::device_vector< double > container_type:
typedef thrust::host_vector< double > container_type:
```

Fancy Iterators

- Up to now: normal iterators based on actual data: `x.begin()`
- Fancy iterators behave like normal iterators, but do not (necessarily) depend on actual data blocks
- Algorithms do not feel a difference
- Example:
  - `constant_iterator`
  - `counting_iterator`
  - `transform_iterator`
  - `permutation_iterator`
  - `zip_iterator`

Counting Iterator

The `counting_iterator` mimics an array with sequential values

```
// create iterators
counting_iterator<int> begin(10);
counting_iterator<int> end = begin + 3;

begin[0] // returns 10
begin[1] // returns 11
begin[100] // returns 110

// sum of [begin, end)
thrust::reduce(begin, end); // returns 33 (i.e. 10 + 11 + 12)

thrust::device_vector<int> x(begin, end); // fills new vector x with 10,11,12
```

Transform Iterator

The `transform_iterator` creates a transformed sequence from data.

```
// initialize vector
device_vector<int> vec(3);
vec[0] = 10; vec[1] = 20; vec[2] = 30;

// create iterator (type omitted)
begin = make_transform_iterator(vec.begin(), thrust::negate<int>());
end = make_transform_iterator(vec.end(), thrust::negate<int>());

begin[0] // returns -10
begin[1] // returns -20
begin[2] // returns -30

// sum of [begin, end)
thrust::reduce(begin, end); // returns -60 (i.e. -10 + -20 + -30)
```

Zip Iterator

The `zip_iterator` joins a number of iterators into a single iterator.

Example: inplace saxpy: $\vec{y} \mathrel{+}= a\vec{x}$

```
template <typename T>
struct saxpy_inplace_functor
{
  const T a;

  saxpy_functor(T _a) : a(_a) {}

  template< typename Tuple >
  __host__ __device__
  void operator()(Tuple t) const {
    thrust::get<1>(t) += a * thrust::get<0>(t);
  }
};

thrust::for_each(
    thrust::make_zip_iterator( thrust::make_tuple( x.begin(), y.begin() ) ),
    thrust::make_zip_iterator( thrust::make_tuple( x.end(), y.end() ) ),
    saxpy_inplace_functor< double >( a ) );

//while( begin != end ) f( *begin++ );
```

Best Practice

- Use as few functors as possible - less but more complicated functors are usually faster.
- Use structures of arrays instead of arrays of structures.
- Use implicit sequences, like `constant_iterator` or `counting_iterator`

Numerical Integration of an ODE

Numerical time evolution according to the Hamiltonian:

$$H = \sum \frac{p_k^2}{2} + \omega_k^2 \frac{q_k^2}{2} + \frac{1}{4}(q_{k+1} - q_k)^4$$

$k = 0 \ldots N-1$

Equations of motion:

$$\dot{q}_k = f(p_k) = p_k$$
$$\dot{p}_k = g(q_k) = -\omega_k^2 q_k - (q_k - q_{k-1})^3 + (q_{k+1} - q_k)^3$$

With periodic boundary conditions: $q_0 = q_N$, $p_0 = p_N$.

Numerical Integration of an ODE

Numerical time evolution according to the Hamiltonian:

$$H = \sum \frac{p_k^2}{2} + \omega_k^2 \frac{q_k^2}{2} + \frac{1}{4}(q_{k+1} - q_k)^4$$

$k = 0 \ldots N-1$
Equations of motion:

$$\dot{q}_k = f(p_k) = p_k$$
$$\dot{p}_k = g(q_k) = -\omega_k^2 q_k - (q_k - q_{k-1})^3 + (q_{k+1} - q_k)^3$$

With periodic boundary conditions: $q_0 = q_N$, $p_0 = p_N$.

First step: numerical evaluation of the rhs $g_k := g(q_k)$.

Compute the rhs

For each value $g_k := g(q_k)$ we need q_k, q_{k-1}, q_{k+1}, ω_k.

```
template< class ValueType >
struct compacton_functor
{
    typedef ValueType value_type;

    template< class Tuple >
    __host__ __device__
    void operator() ( Tuple t )
    {   // the tuple contains q_k, q_k-1, q_k+1, omega_k, g_k
        value_type omega = thrust::get<3>(t);
        value_type x1 = thrust::get<0>(t);
        value_type x2 = thrust::get<0>(t) - thrust::get<1>(t);
        value_type x3 = thrust::get<2>(t) - thrust::get<0>(t);

        thrust::get<4>(t) = -omega * x1 - x2*x2*x2 + x3*x3*x3;
    }
};
```

Compute the rhs

How to get the neighbours? Use permutation iterators!

```
// create integer sequences with permutation indices
prev = [ N-1 , 0 , 1 , 2 , ... , N-2 ]
next = [ 1 , 2 , 3 , ... , N-1 , 0 ]

begin = thrust::make_permutation_iterator( q.begin(), prev.begin() );
end = thrust::make_permutation_iterator( q.begin(), prev.end() );
// creates a sequence that gives q_N-1 , q_0 , q_1 , q_2 , ... , q_N-1

begin = thrust::make_permutation_iterator( q.begin(), next.begin() );
end = thrust::make_permutation_iterator( q.begin(), next.end() );
// creates a sequence that gives q_1 , q_2 , ... , q_N-1 , q_N-2
```

Compute the rhs

Putting things together:

```
thrust::for_each(
    thrust::make_zip_iterator(
        thrust::make_tuple(
            q.begin(),
            thrust::make_permutation_iterator( q.begin() , prev.begin() ),
            thrust::make_permutation_iterator( q.begin() , next.begin() ),
            omega.begin() ,
            g.begin() ) ),
    thrust::make_zip_iterator(
        thrust::make_tuple(
            q.end(),
            thrust::make_permutation_iterator( q.begin() , prev.end() ),
            thrust::make_permutation_iterator( q.begin() , next.end() ),
            omega.end() ,
            g.end() ) ),
    compacton_functor< double >()
);
```

Timestep – Symplectic Euler

Index n counts the timesteps:

$$p_k^{n+1} = p_k^n + g(q_k)\Delta t$$
$$q_k^{n+1} = q_k^n + p_k^{n+1}\Delta t$$

Preserves phase-space volume.

Timestep – Symplectic Euler

Inplace timesteps using the saxpy functor:

```
dpdt = g( q );

thrust::for_each(
    thrust::make_zip_iterator(
        thrust::make_tuple( dpdt.begin() , p.begin() ) ),
    thrust::make_zip_iterator(
        thrust::make_tuple( dpdt.end() , p.end() ) ),
    saxpy_inplace_functor< double > ( dt ) );

thrust::for_each(
    thrust::make_zip_iterator(
        thrust::make_tuple( p.begin() , q.begin() ) ),
    thrust::make_zip_iterator(
        thrust::make_tuple( p.end() , q.end() ) ),
    saxpy_inplace_functor< double > ( dt ) );
```

More complicated (symplectic) Runge-Kutta methods can be implemented in a similar fashion.

```
for( size_t l=0 ; l<6 ; ++l ) {
  thrust::for_each(
    thrust::make_zip_iterator(
      thrust::make_tuple(
        q.begin() ,
        p.begin() ) ),
    thrust::make_zip_iterator(
      thrust::make_tuple(
        q.end() ,
        p.end() ) ),
      increment_functor2< value_type > ( rk_a[l]*dt ) );

  sys( q , m_dpdt );

  thrust::for_each(
    thrust::make_zip_iterator(
      thrust::make_tuple(
        p.begin() ,
        m_dpdt.begin() ) ),
    thrust::make_zip_iterator(
      thrust::make_tuple(
        p.end() ,
        m_dpdt.end() ) ),
      increment_functor2< value_type > ( rk_b[l]*dt ) );
}
```

Conclusions

- Thrust is a very powerful way to employ GPU for numerical computations.
- Using Thrust algorithms, you can use CUDA without knowing anything about CUDA.
- My measurements for a GTX280 show a gain of about 20x over a single 3GHz core with double precision.
- But: high performance is only reached for very large systems – vector size $\sim$ 200.000.
- Debugging and program design is more challenging.

Compatibility Issues

gcc, CUDA and Thrust have to work together
Configurations I know to be working:

- OpenSuse 11.1, gcc 4.3.2, CUDA 3.1, Thrust 1.2.1
- OpenSuse 11.2, gcc 4.4.1, CUDA 3.2, Thrust 1.3
- OpenSuse 11.2, gcc 4.4.1, CUDA 4RC2, Thrust included

Links

- `http://www.nvidia.com/object/cuda_home_new.html`
- `http://code.google.com/p/thrust/`