Raising the Level of GPU Computing

Michael Garland

NVIDIA Research
Many processors each supporting many hardware threads

On-chip memory near processors

Shared global memory space (external DRAM)
```c
__host__
void example()
{
    int B = 128,
    P = ceil(n/B);
    saxpy<<<P,B>>>(n, a, x, y);
}

__global__
void saxpy(int n, float a,
            float *x, float *y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if( i<n ) y[i] = a * x[i] + y[i];
}
```
CUDA C: System language of GPU

- Low-level execution model
- Minimal virtualization of hardware
- Exposes hardware features
  - shared memory, constant memory, textures, …
  - intrinsics: fast math, atomics, intra-warp voting, …

i.e., it’s C … and we can build lots of stuff on top of it
How to Program with Parallelism?

- **Parallel libraries**
  - BLAS, FFT, LAPACK, Scan/Reduce, Sort, …

- **Parallel frameworks / skeletons**
  - OpenGL & Direct3D
  - MapReduce

- **Parallel languages / toolkits**
  - NESL, *Lisp, StreamIt, DPCE, Data Parallel Haskell, …
  - CUDA, MPI, OpenMP, TBB, …
Expanding the foundation

- Libraries for common algorithms
  - CUBLAS and CUFFT in CUDA Toolkit
  - CUDPP: Data Parallel Primitives (e.g., scan & sort)
  - CUSP: Sparse Matrix Methods (e.g., SpMV, CG)
  - *and many others ...*

- API bindings for other languages
  - Fortran, Python, Java, .NET, ...

- Compilers for other languages
  - PGI Fortran, ...

© 2009 NVIDIA Corporation
Building up from the foundation

- Not everyone wants a low-level environment
- Not everyone will engage in heroic code optimization
- Many favor productivity over *maximal* performance
  - they’re satisfied with *good* performance
- How do we help these people?
Position Statement

Scaling to 10 threads isn’t interesting

Scaling to 10,000 threads requires data parallelism

Robust, general auto-parallelization will not happen
  p.s. I would love to be proven wrong

Programmers should write programs where data decomposition implicitly yields parallelism

Libraries/compilers should exploit that parallelism
Kernel Generation via C++ Template Meta-Programming

User: Array-wide expression

Result = Mult(B, Add(
    A.shift(1,0,0), A.shift(-1,0,0)));

Expression tree represented via C++ templates

Generated: per-thread kernel code

Result.at(i,j,k) = B.at(i,j,k) * 
    (A.at(i+1,j,k) + A.at(i-1,j,k));
Example: 3D Laplacian

```c
Grid3DDeviceF d_src, d_result;
// init d_src to something
d_result =
    read_shift<1,0,0>(d_src) + read_shift<-1,0,0>(d_src) +
    read_shift<0,1,0>(d_src) + read_shift<0,-1,0>(d_src) +
    read_shift<0,0,1>(d_src) + read_shift<0,0,-1>(d_src) +
    read(d_src) * constant(-6.0f);
```

<table>
<thead>
<tr>
<th></th>
<th>Auto-generated kernel</th>
<th>Hand-written kernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Time</td>
<td>8.164 ms</td>
<td>7.814 ms</td>
</tr>
<tr>
<td>Register Count</td>
<td>15</td>
<td>9</td>
</tr>
<tr>
<td>Instruction Count</td>
<td>159</td>
<td>69</td>
</tr>
<tr>
<td>Lines of User Code</td>
<td>5</td>
<td>40</td>
</tr>
</tbody>
</table>

© 2009 NVIDIA Corporation
Jared Hoberock and Nathan Bell
NVIDIA Research

THRUST
int main(void)
{
  // generate random data on the host
  thrust::host_vector<int> h_vec(1000000);
  thrust::generate(h_vec.begin(), h_vec.end(), rand);

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

  // sort 140M 32b keys/sec on GT200
  thrust::sort(d_vec.begin(), d_vec.end());

  return 0;
}
Example: Level 1 BLAS operations

- Define some function object types
  
  ```
  struct scale_and_add;  struct square;  struct plus;
  ```

- Apply some Thrust algorithms
  
  ```
  void saxpy(int n, float alpha, float *x, float *y){
    thrust::transform(x, x+n, y, y, scale_and_add(alpha));
  }

  float snrm2(int n, float *x) {
    return sqrt(thrust::transform_reduce(x, x+n, square(), 0, plus()));
  }
  ```
Templates for compile-time dispatch

- `thrust::sort` will select the right algorithm:
  - radix sort for built-in types (int, float, etc.)
  - merge sort where radix sort cannot be used

- `thrust::reduce` handles data sizes appropriately:
  - e.g., efficient memory access of 8-bit chars vs. 32-bit floats

- Handle `host_vector` & `device_vector` transparently

- Provide user-defined function objects:
  - `thrust::sort(begin, end, my_comparator());`
Bryan Catanzaro and Michael Garland
UC Berkeley & NVIDIA Research

COPPERHEAD
Consider this intrinsically parallel procedure

```python
def saxpy(a, x, y):
    return map(lambda xi, yi: a*xi + yi, x, y)
```

... or for the lambda averse ...

```python
def saxpy(a, x, y):
    return [a*xi + yi for xi, yi in zip(x, y)]
```

This procedure is both
- completely valid Python code
- compilable to a corresponding CUDA kernel
Hello GPU programming

```python
» from copperhead import *

» @cu
def saxpy(a, x, y):
    return map(lambda xi,yi: a*xi+yi, x, y)

» x = [1.0, 1.0, 1.0, 1.0]  # can use NumPy or
» y = [0.0, 1.0, 2.0, 3.0]  # CuArrays, too

» gpuResult = saxpy(2.0, x, y)
» cpuResult = saxpy(2.0, x, y, cuEntry=False)
```

© 2009 NVIDIA Corporation
Disclaimer: Work in Progress
Copperhead is a Python subset

- Every procedure is also a valid Python procedure
  - don’t need to sell people on a new language
  - fully functioning sequential environment for free

- Try to maximize productivity
  - interactive algorithm prototyping
  - executable “pseudo-code”

- Other sources of inspiration
  - APL, SETL, MATLAB, …
  - Nesl, Data Parallel Haskell, …

© 2009 NVIDIA Corporation
Procedures must be statically typed

- **Standard Hindley-Milner style type inference**
  - `def plus1(x): return x+1`
  - `plus1 :: int -> int`

- **Supporting parametric polymorphism**
  - `def saxpy(a, x, y):`
    - `return map(lambda xi,yi: a*xi+yi, x, y)`
  - `saxpy :: (a, [a], [a]) -> [a]`

- **And rejecting ill-typed procedures**
  - `def ill_typed(p):`
    - `return 1 if p else True`
Side-effects are forbidden

An acceptable Copperhead procedure:

```python
def saxpy(a, x, y):
    return map(lambda xi, yi: a*xi + yi, x, y)
```

Valid Python but forbidden in Copperhead:

```python
def saxpy(a, x, y):
    for i in indices(y):
        y[i] = a*x[i] + y[i]
    return y
```

parallelization requires *a priori* knowledge about `indices(y)`
Data-driven parallelism

- Parallelism arises from map

- Or primitive procedures built from it
  - reduce
  - scan
  - sort
  - ...

© 2009 NVIDIA Corporation
Data-driven synchronization

Joining previously independent sequences
  » join(map(sort, split(A))

Data access patterns that can’t be statically localized
  » B = gather(map(f, A), indices)
  » total = plus_reduce(B)
Coordinating CUDA routines

Assume we have a block-level reduction primitive

```cpp
template<typename T>
__device__ T plus_reduce_P(sequence<T> values);
```

We can easily build a global reduction procedure

```python
@cu
def plus_reduce(A):
    tiles = split_by_size(A, tilesize)
    partials = map(plus_reduce_P, tiles)
    return plus_reduce_P(partials)
```
Implementing complete algorithms

```python
» def rank(i, A):
    """Count items that sort to left of A[i]""
    nlt = count(operator.lt, A[i], A)
    neq = count(operator.eq, A[i], take(A,i))
    return nlt + neq

» def counting_sort(A):
    ranks = [rank(i,A) for i in indices(A)]
    return permute(ranks, A)
```
Auto-sequentialization

- Compiler chooses sequential vs. parallel
  
  \[
  \text{total} = \text{reduce} (\text{map} (\text{reduce}, \text{split}(A)))
  \]

- Deeply nested primitives are sequentialized

- Note that there is no “correct” answer here
  
  - depends on input size
  - depends on architecture
Automatic kernel fusion/fission

- Compiler infers & schedules “phase” boundaries
- Points where synchronization is required

\[ B = \text{reduce} (\text{map}(A)) \]
\[ D = \text{reduce} (\text{map}(C)) \]
Some broader challenges

- How do we make parallel programming suitable for teaching 1st year CS/CE students?

- How do we make parallel programming “easy” for the majority of programmers?

- How do we enable a scientist to develop code on a desktop and which will run at scale on a cluster?
Questions?

mgarland@nvidia.com

http://www.nvidia.com/research
GPU Technology Conference
Sept 30 – Oct 2, 2009 – The Fairmont San Jose, California

The most significant event in 2009 dedicated to application development on the GPU

- Learn about the seismic shifts happening in computing
- Preview disruptive technologies and emerging applications to stay ahead of imminent trends
- Get tools/techniques to impact mission critical projects now
- Network with experts and peers from across several industries

Event focused on developers, engineers, researchers, senior executives, venture capitalists, press and analysts
Summits at the GPU Technology Conference

Emerging Companies Summit
- For investors, venture capitalists and entrepreneurs
- Recognized as a premier private company showcase

GPU Developers Summit
- For developers, programmers and engineers
- In-depth look at tools and techniques to impact mission-critical work NOW

NVIDIA Research Summit
- For researchers and academics
- Showcase findings and learn about ways to reduce time-to-discovery

Opportunities:
- Call for Submissions
  open now for sessions, tutorials, panels, birds of a feather, posters and moderated roundtables for GPU Developers Summit and NVIDIA Research Summit
- Sponsors / Exhibitors
  have a variety of options to reach influential decision makers across a broad range of fields
- Startup Showcase
  Present your company and technology to potential investors
CUDA Model of Parallelism

- CUDA virtualizes the physical hardware
  - thread is a virtualized scalar processor (registers, PC, state)
  - block is a virtualized multiprocessor (threads, shared mem.)

- Scheduled onto physical hardware without pre-emption
  - threads блокs launch & run to completion
  - blocks should be independent