GPU Metaprogramming applied to High Order DG and Loop Generation

Andreas Klöckner

Division of Applied Mathematics
Brown University

August 19, 2009
Thanks

- Jan Hesthaven (Brown)
- Tim Warburton (Rice)
- Akil Narayan (Brown)
- PyCUDA contributors
- Nvidia Corporation
Outline

1. GPU 101
2. GPU Scripting
3. DG on GPUs
4. Perspectives
Outline

1. GPU 101
   - What and Why?

2. GPU Scripting

3. DG on GPUs

4. Perspectives
Accelerated Computing?

- Design target for CPUs:
  - Make a single thread very fast
  - Hide latency through large caches
  - Predict, speculate
Accelerated Computing?

- Design target for CPUs:
  - Make a single thread very fast
  - Hide latency through large caches
  - Predict, speculate

- Accelerated/Stream Computing takes a different approach:
  - Throughput matters—single threads do not
  - Hide latency through parallelism
  - Let programmer deal with “raw” storage hierarchy
CPU Chip Real Estate

65 nm, 4 SP ops at a time, 1 MiB L2.

Andreas Klöckner
Applied Math · Brown University
GPU Metaprogramming applied to High Order DG and Loop Generation
GPU Chip Real Estate

55 nm, 800 SP ops at a time.
Questions?
Outline

1. GPU 101
2. GPU Scripting
   - Abstracting away the annoying parts
   - GPU Metaprogramming
3. DG on GPUs
4. Perspectives

Andreas Klöckner
Applied Math · Brown University
GPU Metaprogramming applied to High Order DG and Loop Generation
How are High-Performance Codes constructed?

- "Traditional" Construction of High-Performance Codes:
  - C/C++/Fortran
  - Libraries

- "Alternative" Construction of High-Performance Codes:
  - Scripting for ‘brains’
  - GPUs for ‘inner loops’

- Play to the strengths of each programming environment.
A scripting language... 
- is discoverable and interactive.
- has comprehensive built-in functionality.
- manages resources automatically.
- encourages abstraction.
- works well for “gluing” lower-level blocks together.
Scripting: Interpreted, not Compiled

Program creation workflow:

1. Edit
2. Compile
3. Link
4. Run
Scripting: Interpreted, not Compiled

Program creation workflow:

1. Edit
2. Compile → Crossed out
3. Link
4. Run

Abstracting away the annoying parts

Andreas Klöckner
Applied Math · Brown University

GPU Metaprogramming applied to High Order DG and Loop Generation
Scripting: Interpreted, not Compiled

Program creation workflow:

- Edit
- Compile (crossed out)
- Link (crossed out)
- Run
Why do Scripting for GPUs?

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput

→ complement each other
Why do Scripting for GPUs?

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput

→ complement each other

- CPU: largely restricted to control tasks (≈1000/sec)
  - Scripting fast enough
Why do Scripting for GPUs?

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput
  → complement each other
- CPU: largely restricted to control tasks (~1000/sec)
  - Scripting fast enough
- Python + CUDA = PyCUDA
PyCUDA Philosophy

- Provide complete access
- Automatically manage resources
- Provide abstractions
- Allow interactive use
- Check for and report errors automatically
- Integrate tightly with numpy
PyCUDA: Vital Information

- [http://mathema.tician.de/software/pycuda](http://mathema.tician.de/software/pycuda)
- Complete documentation
- MIT License
  (no warranty, free for all use)
- Requires: numpy, Boost C++, Python 2.4+.
- Support via mailing list.
PyCUDA: Vital Information

- [ ] http://mathema.tician.de/software/pycuda
- Complete documentation
- MIT License
  (no warranty, free for all use)
- Requires: numpy, Boost C++, Python 2.4+
- Support via mailing list.
- Sister project: PyOpenCL (TBR soon)
In GPU scripting, GPU code does not need to be a compile-time constant.
In **GPU scripting**, GPU code does *not* need to be a compile-time constant.

(Key: Code is data—it *wants* to be reasoned about at run time)
In GPU scripting, GPU code does \textit{not} need to be a compile-time constant.

(Key: Code is data—it \textit{wants} to be reasoned about at run time)
In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data—it *wants* to be reasoned about at run time)
In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
In GPU scripting, GPU code does \textit{not} need to be a compile-time constant.

(Key: Code is data—it \textit{wants} to be reasoned about at run time)
Machine-generated Code

Why machine-generate code?

- Automated Tuning (cf. ATLAS, FFTW)
- Data types
- Specialize code for given problem
- Constants faster than variables (→ register pressure)
- Loop Unrolling
PyCUDA: Support for Metaprogramming

- Access to resource use data
  - such as register count, shared mem, ...
- $\mu$s-precision GPU timing
- Built-in heuristics ("occupancy" etc.)
- codepy:
  - Build C syntax trees from Python
  - Generates readable, indented (GPU) C code
- Or use a templating engine (many available)
Questions?
Outline

1. GPU 101
2. GPU Scripting
3. DG on GPUs
   - Introduction
   - DG and Metaprogramming
   - Results
4. Perspectives

Andreas Klöckner
Applied Math · Brown University
GPU Metaprogramming applied to High Order DG and Loop Generation
Discontinuous Galerkin Method

Let $\Omega := \bigcup_i D_k \subset \mathbb{R}^d$. 
Discontinuous Galerkin Method

Let $\Omega := \bigcup_i D_k \subset \mathbb{R}^d$.

**Goal**

Solve a *conservation law* on $\Omega$:

$$u_t + \nabla \cdot F(u) = 0$$

*Example*: Maxwell's Equations

Maxwell's Equations: EM field: $E(x, t), H(x, t)$ on $\Omega$ governed by

$$\frac{\partial E}{\partial t} - \frac{1}{\varepsilon} \nabla \times H = -j\varepsilon,$$

$$\frac{\partial H}{\partial t} + \frac{1}{\mu} \nabla \times E = 0,$$

$$\nabla \cdot E = \rho \varepsilon,$$

$$\nabla \cdot H = 0.$$
Discontinuous Galerkin Method

Let $\Omega := \bigcup_i D_k \subset \mathbb{R}^d$.

Goal

Solve a conservation law on $\Omega$:

$$u_t + \nabla \cdot F(u) = 0$$

Example

Maxwell’s Equations: EM field: $E(x, t), H(x, t)$ on $\Omega$ governed by

$$\partial_t E - \frac{1}{\varepsilon} \nabla \times H = -\frac{j}{\varepsilon},$$

$$\nabla \cdot E = \frac{\rho}{\varepsilon},$$

$$\partial_t H + \frac{1}{\mu} \nabla \times E = 0,$$

$$\nabla \cdot H = 0.$$
Discontinuous Galerkin Method

Multiply by test function, integrate by parts:

\[
0 = \int_{D_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, dx = \int_{D_k} u_t \varphi - F(u) \cdot \nabla \varphi \, dx + \int_{\partial D_k} (\hat{n} \cdot F)^* \varphi \, dS_x,
\]

Integrate by parts again, substitute in basis functions, introduce elementwise differentiation and “lifting” matrices \( D, L \):

\[
\partial_t u^k = - \sum_{\nu} D^{\partial \nu, k} [F(u^k)] + L^k [\hat{n} \cdot F - (\hat{n} \cdot F)^*]|A \subset \partial D_k.
\]

For straight-sided simplicial elements:
Reduce \( D^{\partial \nu} \) and \( L \) to reference matrices.
Decomposition of a DG operator into Subtasks

DG's execution decomposes into two (mostly) separate branches:

\[ F(u^k) \quad \text{Flux Lifting} \quad \partial_t u^k \]

- **Flux Gather**
- **Flux Lifting**
- **Local Differentiation**

Green: Element-local parts of the DG operator.

**Note:** Explicit timestepping, nodal representation.
DG on GPUs: Possible Advantages

DG on GPUs: Why?

- GPUs have deep Memory Hierarchy
  - The majority of DG is local.
- Compute Bandwidth ≫ Memory Bandwidth
  - DG is arithmetically intense.
- GPUs favor dense data.
  - Local parts of the DG operator are dense.
DG on GPUs: Challenges

What makes DG on GPUs challenging?

- GPUs have preferred granularities \((2^n\text{ mostly})\)
  - DG has built-in granularities, too \((\neq 2^n\text{ mostly})\)
- Much data reuse in DG (or any matrix product workload)
  - Very little cache memory available \((\sim 20\text{ KiB})\)
  - Cache managed by user (!)
- Loop slicing can have a large impact
  - GPUs introduce several more slicing axes
  - Loop slicing (= “computation layout”) determines memory layout for other steps.
Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms
Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms
- Automated Tuning:
  - Memory layout
  - Loop slicing
  - Gather granularity
Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms

- Automated Tuning:
  - Memory layout
  - Loop slicing
  - Gather granularity

- Constants instead of variables:
  - Dimensionality
  - Polynomial degree
  - Element properties
  - Matrix sizes
Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms
- Automated Tuning:
  - Memory layout
  - Loop slicing
  - Gather granularity
- Constants instead of variables:
  - Dimensionality
  - Polynomial degree
  - Element properties
  - Matrix sizes
- Loop Unrolling
Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms (*)

- Automated Tuning:
  - Memory layout
  - Loop slicing (*)
  - Gather granularity

- Constants instead of variables:
  - Dimensionality
  - Polynomial degree
  - Element properties
  - Matrix sizes

- Loop Unrolling
0 = \int_{D_k} u_t \phi + [\nabla \cdot F(u)] \phi \, dx - \int_{\partial D_k} [\hat{n} \cdot F - (\hat{n} \cdot F)^*] \phi \, dS_x

\text{Flux term}
Metaprogramming DG: Flux Terms

\[ 0 = \int_{D_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, dx - \int_{\partial D_k} [\hat{n} \cdot F - (\hat{n} \cdot F)^*] \varphi \, dS_x \]

Flux terms:
- vary by problem
- expression specified by user
- evaluated pointwise
Metaprogramming DG: Flux Terms Example

**Example:** Fluxes for Maxwell’s Equations

\[\hat{n} \cdot (F - F^*)_E := \frac{1}{2} \left[\hat{n} \times ([H] - \alpha \hat{n} \times [E])\right]\]
Metaprogramming DG: Flux Terms Example

**Example:** Fluxes for Maxwell’s Equations

\[ \hat{n} \cdot (F - F^*)_E := \frac{1}{2} [\hat{n} \times ([H] - \alpha \hat{n} \times [E])] \]

**User writes:** Vectorial statement in math. notation

```
flux = 1/2*cross(normal, h.int-h.ext -alpha*cross(normal, e.int-e.ext))
```
Metaprogramming DG: Flux Terms Example

**Example:** Fluxes for Maxwell’s Equations

\[
\hat{n} \cdot (F - F^*)_E := \frac{1}{2} [\hat{n} \times ([H] - \alpha \hat{n} \times [E])]
\]

**We generate:** Scalar evaluator in C

```c
a_flux += ( (((val_a_field5 - val_b_field5) * fpair) -> normal[2])
   - ((val_a_field4 - val_b_field4) * fpair) -> normal[0])
+ (val_a_field0 - val_b_field0) * fpair -> normal[0]
   - (((val_a_field4 - val_b_field4) * fpair) -> normal[1])
   - ((val_a_field1 - val_b_field1) * fpair) -> normal[2])
+ (val_a_field3 - val_b_field3) * fpair -> normal[1]
) * value_type (0.5);
```
Loop Slicing on the GPU: A Pattern

**Setting:** \( N \) independent work units + preparation

**Question:** How should one assign work units to threads?
Loop Slicing on the GPU: A Pattern

**Setting:** $N$ independent work units + preparation

**Question:** How should one assign work units to threads?

$w_S$: in sequence

$\downarrow$ Thread
Loop Slicing on the GPU: A Pattern

**Setting:** \( N \) independent work units + preparation

**Question:** How should one assign work units to threads?

\( w_s: \) in sequence

\( w_p: \) in parallel
Loop Slicing on the GPU: A Pattern

Setting: \( N \) independent work units + preparation

Question: How should one assign work units to threads?

\( w_s: \) in sequence

\( w_i: \) "inline-parallel"

\( w_p: \) in parallel
Loop Slicing on the GPU: A Pattern

**Setting:** $N$ independent work units + preparation

**Question:** How should one assign work units to threads?

$w_s$: in sequence

$w_i$: “inline-parallel”

$w_p$: in parallel

(amortize preparation)
Loop Slicing on the GPU: A Pattern

**Setting:** $N$ independent work units + preparation

**Question:** How should one assign work units to threads?

$w_s$: in sequence  
$w_i$: “inline-parallel”  
$w_p$: in parallel

( amortize preparation)  
(exploit register space)

Andreas Klöckner  
Applied Math · Brown University

GPU Metaprogramming applied to High Order DG and Loop Generation
Loop Slicing for Differentiation

Local differentiation, matrix-in-shared, order 4, with microblocking point size denotes $w_i \in \{1, \ldots, 4\}$

Execution time [ms] Local differentiation, matrix-in-shared, order 4, with microblocking point size denotes $w_i \in \{1, \ldots, 4\}$
Nvidia GTX280 vs. single core of Intel Core 2 Duo E8400

![Graph comparing GFlops/s and Speedup Factor with Polynomial Order N for GPU and CPU.](https://www.example.com/graph.png)

Andreas Klöckner
Applied Math · Brown University

GPU Metaprogramming applied to High Order DG and Loop Generation
Memory Bandwidth on a GTX 280

![Graph showing memory bandwidth for different operations and polynomial orders.](image)

- **Axes:**
  - Y-axis: Global Memory Bandwidth [GB/s]
  - X-axis: Polynomial Order $N$

- **Legend:**
  - **Gather**
  - **Lift**
  - **Diff**
  - **Assy.**
  - **Peak**

- **Operations:**
  - Gather
  - Lift
  - Diff
  - Assy.

- **Polynomial Orders:**
  - 20, 40, 60, 80, 100, 120, 140, 160, 180, 200

- **Memory Bandwidth Trends:**
  - Increase with increasing polynomial order
  - Variations between operations

**Note:**
- GPUs perform better with higher polynomial orders.
- Memory bandwidth peaks at certain orders and then decreases.

---

Andreas Klöckner
Applied Math · Brown University

GPU Metaprogramming applied to High Order DG and Loop Generation
16 T10s vs. 64 = 8 × 2 × 4 Xeon E5472

Flop Rates and Speedups: 16 GPUs vs 64 CPU cores

<table>
<thead>
<tr>
<th>Polynomial Order N</th>
<th>GPU</th>
<th>CPU</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1000</td>
<td>500</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2000</td>
<td>1000</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3000</td>
<td>1500</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>4000</td>
<td>2000</td>
<td>2</td>
</tr>
</tbody>
</table>

GFlops/s

Speedup Factor

Andreas Klöckner
Applied Math · Brown University

GPU Metaprogramming applied to High Order DG and Loop Generation
Double Precision, T10 vs Xeon (Poisson)

Performance: Double Precision Poisson Solver
Unpreconditioned CG with IP DG on $K=18068$ elements

Andreas Klöckner
Applied Math · Brown University

GPU Metaprogramming applied to High Order DG and Loop Generation
“Real-World” Scattering Calculation

Order \( N = 4 \),
78,745 elements,
2.7M \( \cdot \) 6 DOFs,
single Tesla C1060.
Outline

1. GPU 101
2. GPU Scripting
3. DG on GPUs
4. Perspectives
   - Loo.py
   - Iterative CUDA
   - Conclusions
Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

- Obvious idea: Let the computer do it.
- One way: Smart compilers
Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

- Obvious idea: Let the computer do it.
- One way: Smart compilers
  - GPU programming requires complex tradeoffs
  - Tradeoffs require heuristics
  - Heuristics are fragile
Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

- **Obvious idea:** Let the computer do it.
  - One way: Smart compilers
    - GPU programming requires complex tradeoffs
    - Tradeoffs require heuristics
    - Heuristics are fragile

- **Another way:** Dumb enumeration
  - Enumerate loop slicings
  - Enumerate prefetch options
  - Choose by running resulting code on actual hardware
Empirical GPU loop optimization:

```python
a, b, c, i, j, k = [var(s) for s in "abcijk"]
n = 500
k = make_loop_kernel([LoopDimension("i", n), LoopDimension("j", n), LoopDimension("k", n),
                      ], [
                      (c[i+n*j], a[i+n*k]*b[k+n*j])
                      ])
gen_kwargs = {
              "min_threads": 128,
              "min_blocks": 32,
            }
```

→ Ideal case: Finds 160 GF/s kernel without human intervention.
Loo.py Status

- Limited scope:
  - Require input/output separation
  - Kernels must be expressible using "loopy" model
    (i.e. indices decompose into "output" and "reduction")
  - Enough for DG, LA, FD, ...

Kernel compilation limits trial rate

Non-Goal: Peak performance

Good results currently for dense linear algebra and (some) DG subkernels
Loo.py Status

- Limited scope:
  - Require input/output separation
  - Kernels must be expressible using “loopy” model
    (i.e. indices decompose into “output” and “reduction”)
  - Enough for DG, LA, FD, ...
- Kernel compilation limits trial rate
- Non-Goal: Peak performance
- Good results currently for dense linear algebra and (some) DG subkernels
Iterative CUDA

- GPU-based sparse linear system solver library (CG so far, trivial to add more)
- Built-in GPU Sparse Matrix-Vector multiplication
- Pure C++ on the outside—encapsulates GPU build complexity → Use as “yet another solver library”
- MIT License
- Problem size only limited by matrix storage on GPU
- About $10 \times$ performance gain in SP and DP
- Same functionality also in PyCUDA

\footnote{1PKT format, Bell/Garland 2008}
Conclusions

- Two technologies “ready for prime-time” now:
  - Scripting
  - GPUs

Andreas Klöckner
Applied Math · Brown University

GPU Metaprogramming applied to High Order DG and Loop Generation
Conclusions

- Two technologies “ready for prime-time” now:
  - Scripting
  - GPUs
- GPUs and scripting work well together
  - Enable Metaprogramming
Conclusions

- Two technologies “ready for prime-time” now:
  - Scripting
  - GPUs

- GPUs and scripting work well together
  - Enable Metaprogramming

- Further work in GPU-DG:
  - Other equations (Euler, Navier-Stokes)
  - Curvilinear Elements
  - Local Time Stepping
Where to from here?

PyCUDA, PyOpenCL

→ http://www.dam.brown.edu/people/kloeckner/

CUDA-DG Preprint


→ http://arxiv.org/abs/0901.1024
Thank you for your attention!

http://www.dam.brown.edu/people/kloeckner/

http://arxiv.org/abs/0901.1024
Image Credits

- C870 GPU: Nvidia Corp.
- Isaiah die shot: VIA Technologies
- RV770 die shot: AMD Corp.
- C870 GPU: Nvidia Corp.
- Old Books: flickr.com/ppdigital (CC)
- Floppy disk: flickr.com/ethanhein (CC)
- Machine: flickr.com/13521837@N00 (CC)