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

BROWN

Outline

Accelerated Computing?

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

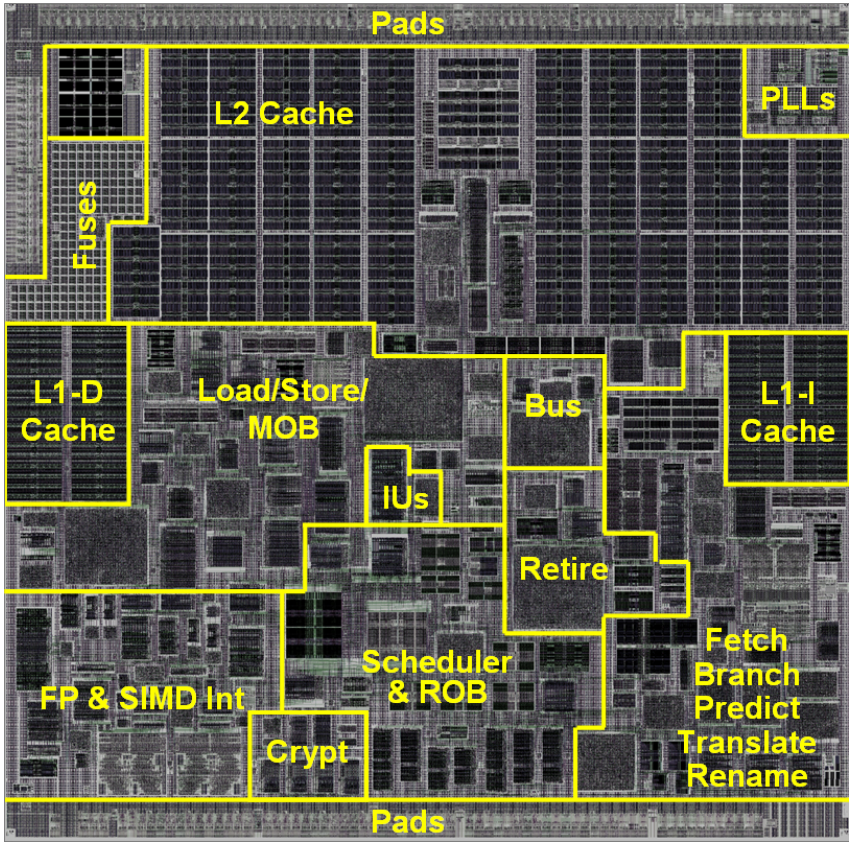
BROWN

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

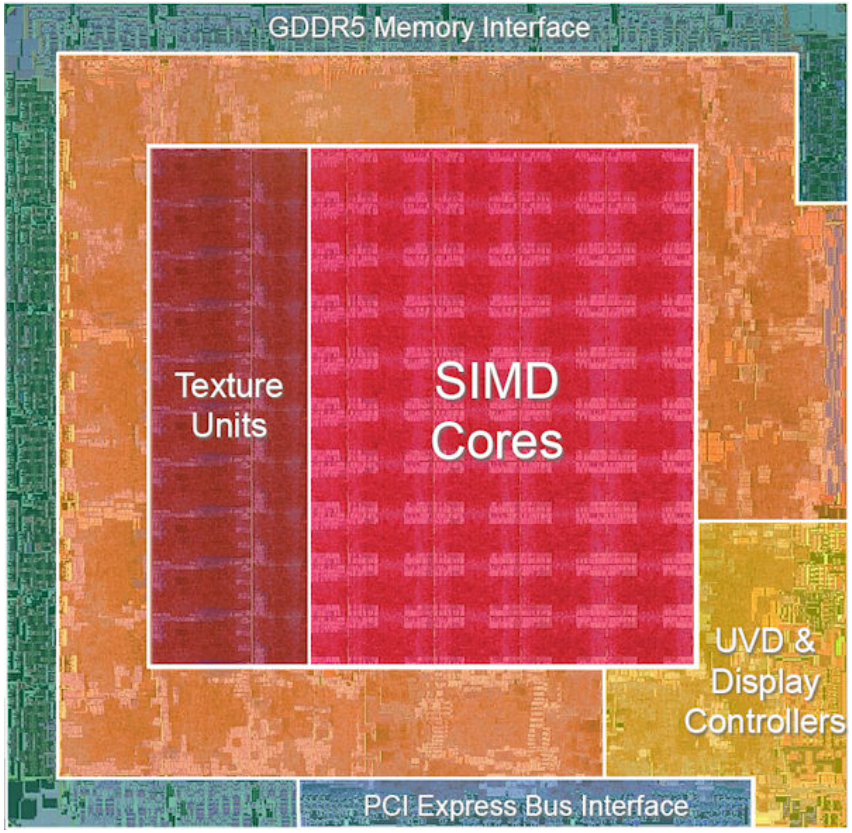
BROWN

CPU Chip Real Estate



Die floorplan: VIA Isaiah (2008).
65 nm, 4 SP ops at a time, 1 MiB L2.

GPU Chip Real Estate



Die floorplan: AMD RV770 (2008).
55 nm, 800 SP ops at a time.

Questions?

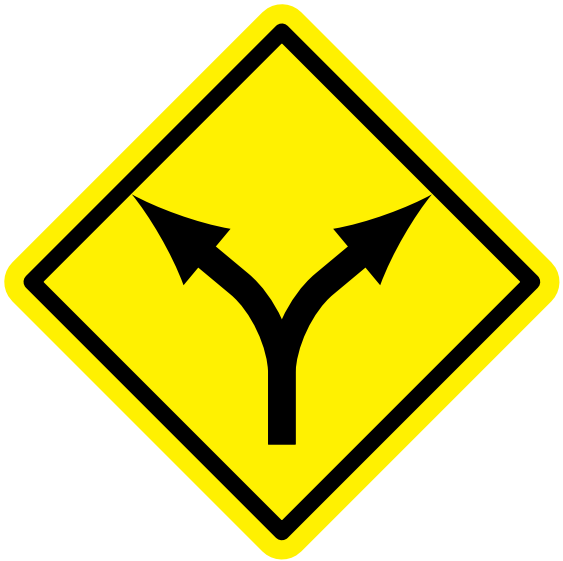
?

Outline

BROWN

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.



BROWN

Scripting: Means

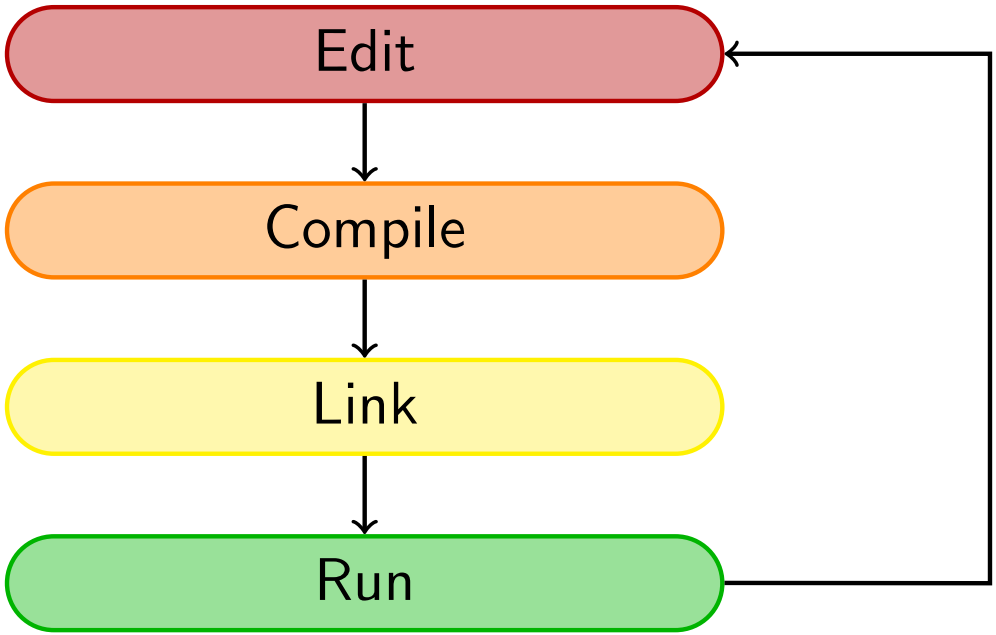
A scripting language...

- is discoverable and interactive.
- has comprehensive built-in functionality.
- manages resources automatically.
- encourages abstraction.
- works well for "gluing" lower-level blocks together.

BROWN

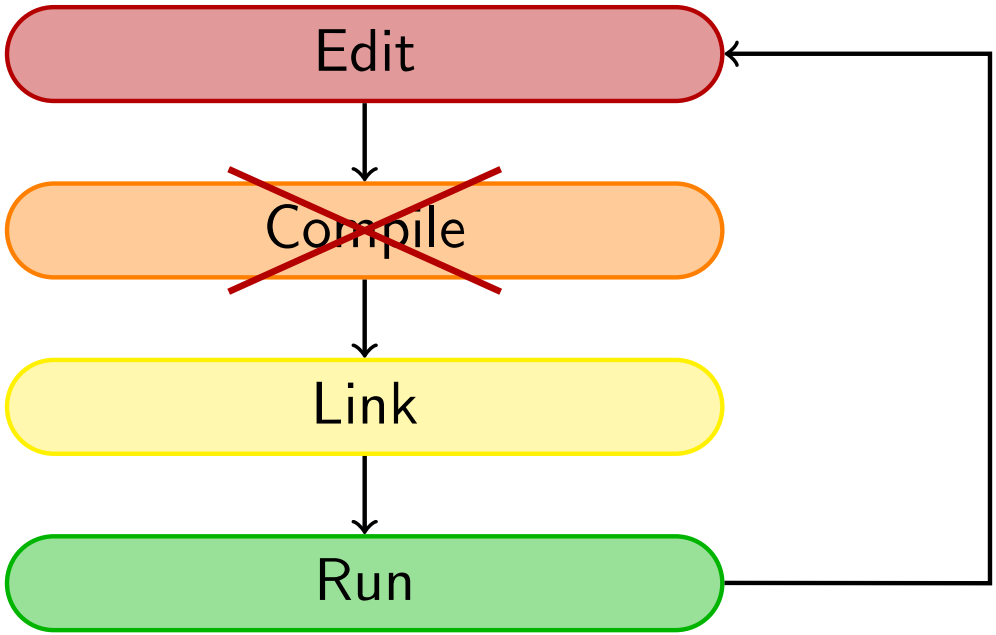
Scripting: Interpreted, not Compiled

Program creation workflow:



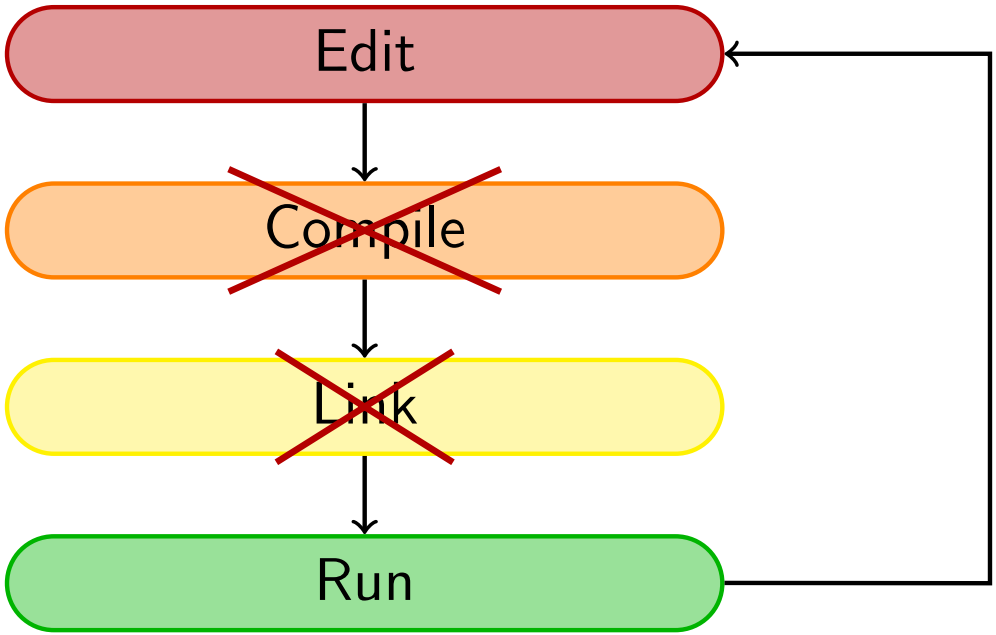
Scripting: Interpreted, not Compiled

Program creation workflow:



Scripting: Interpreted, not Compiled

Program creation workflow:



Why do Scripting for GPUs?

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

$\longrightarrow$ complement each other

BROWN

Why do Scripting for GPUs?

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

  $\longrightarrow$ complement each other
- CPU: largely restricted to control tasks ($\sim$1000/sec)
  - Scripting fast enough

BROWN

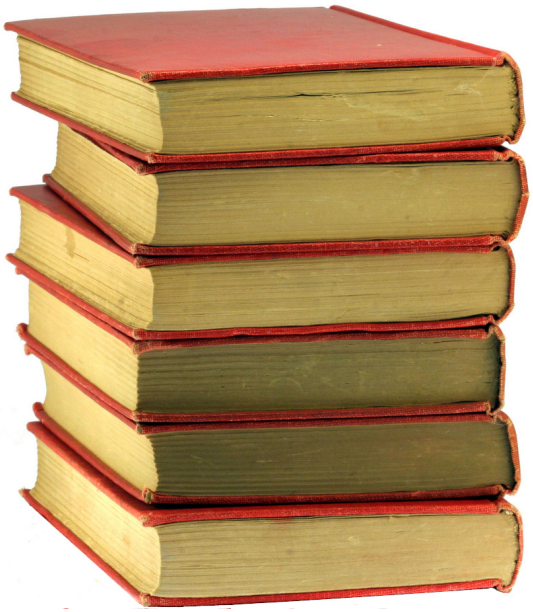
Why do Scripting for GPUs?

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

  $\longrightarrow$ complement each other
- CPU: largely restricted to control tasks ($\sim$1000/sec)
  - Scripting fast enough
- Python + CUDA = **PyCUDA**

BROWN

PyCUDA Philosophy



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

BROWN

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.

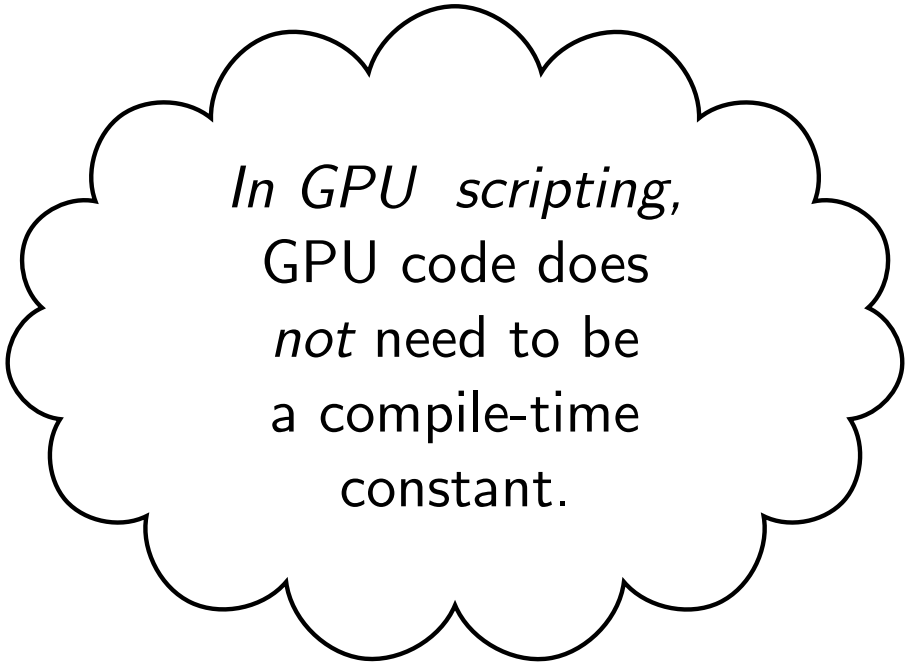
BROWN

PyCUDA: Vital Information

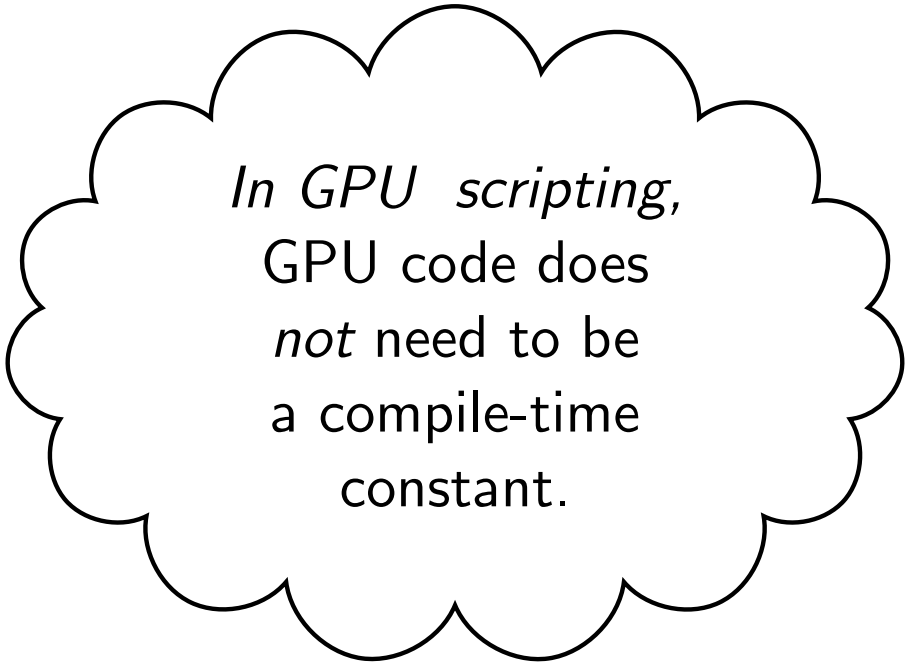
- `http://mathema.tician.de/software/pycuda`
- Complete documentation
- MIT License
  (no warranty, free for all use)
- Requires: numpy, Boost C++,
  Python 2.4+.
- Support via mailing list.
- Sister project: PyOpenCL (TBR soon)

BROWN

Metaprogramming



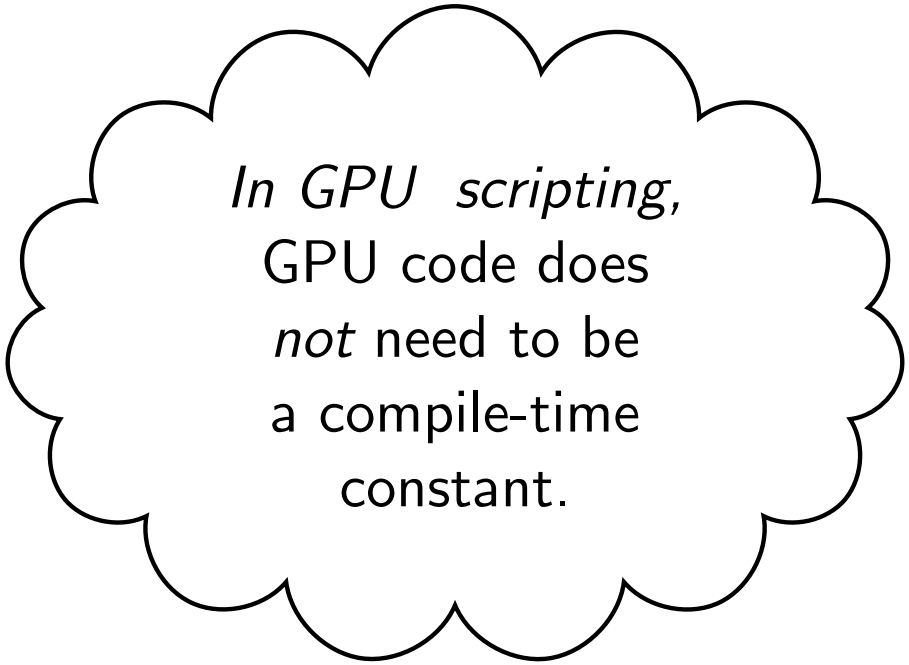
Metaprogramming



(Key: Code is data–it *wants* to be reasoned about at run time)

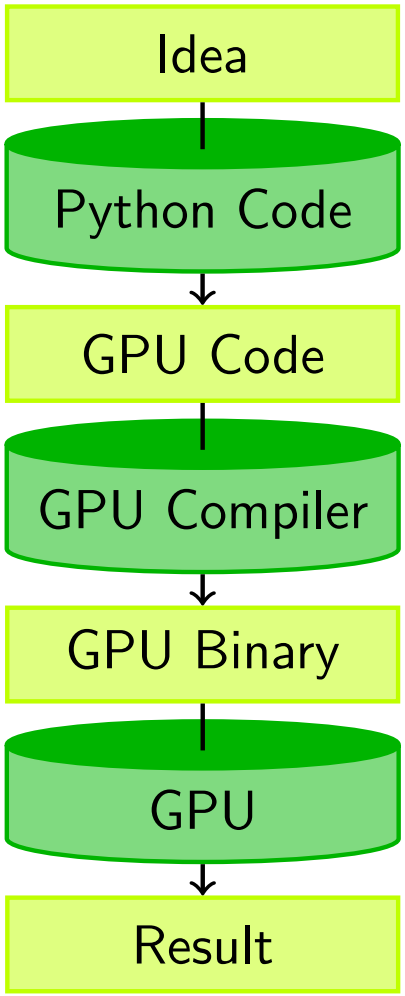
Metaprogramming

Idea



(Key: Code is data–it *wants* to be reasoned about at run time)

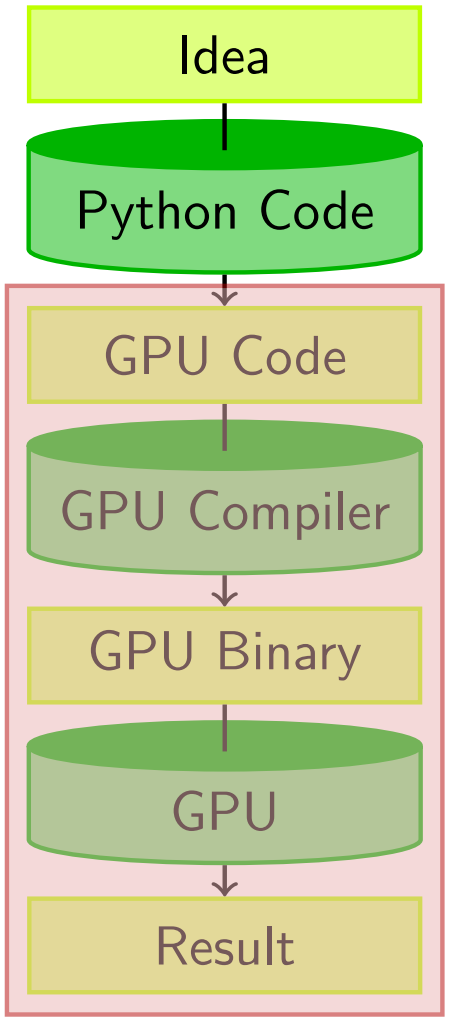
Metaprogramming



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)

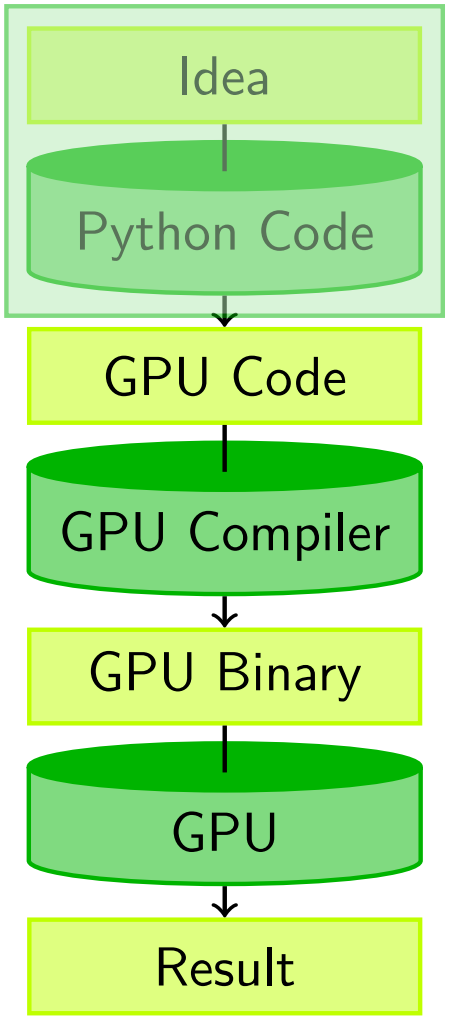
Metaprogramming



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)

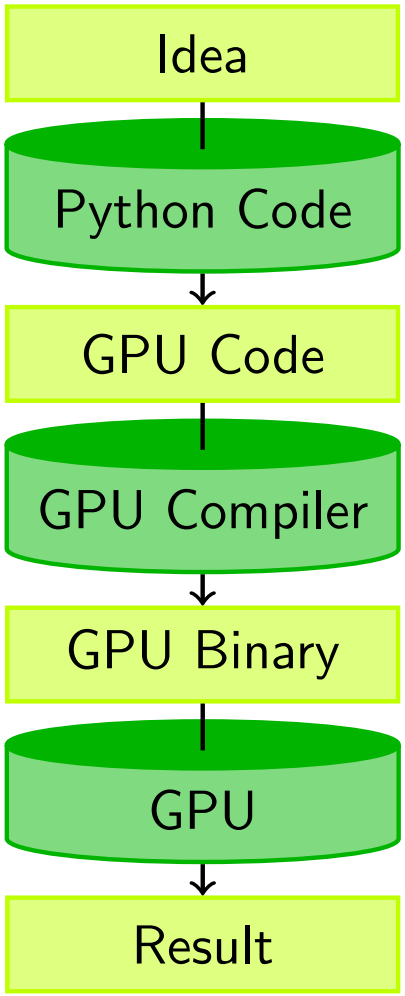
Metaprogramming



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)

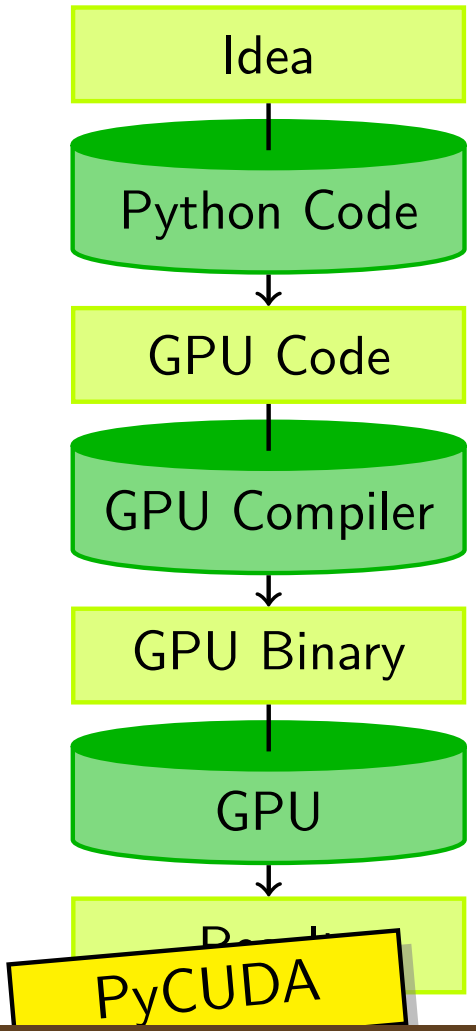
Metaprogramming



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)

Metaprogramming



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)

Machine-generated Code

Why machine-generate code?

- Automated Tuning
  (cf. ATLAS, FFTW)
- Data types
- Specialize code for given problem
- Constants faster than variables
  ($\rightarrow$ register pressure)
- Loop Unrolling

BROWN

PyCUDA: Support for Metaprogramming

- Access to resource use data
  - such as register count, shared mem, ...
- μs-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)

BROWN

Questions?

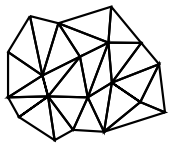
?

Outline

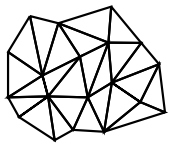
BROWN

Discontinuous Galerkin Method

Let $\Omega := \bigcup_i \mathrm{D}_k \subset \mathbb{R}^d$.



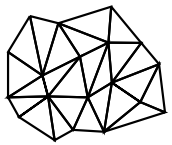
Discontinuous Galerkin Method

Let $\Omega := \bigcup_i \mathrm{D}_k \subset \mathbb{R}^d$. 

Goal

Solve a *conservation law* on Ω: $u_t + \nabla \cdot F(u) = 0$

Discontinuous Galerkin Method

Let $\Omega := \bigcup_i \mathrm{D}_k \subset \mathbb{R}^d$. 

Goal

Solve a *conservation law* on Ω: $u_t + \nabla \cdot F(u) = 0$

Example

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

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

$$\nabla \cdot E = \frac{\rho}{\varepsilon}, \qquad \nabla \cdot H = 0.$$

Discontinuous Galerkin Method

Multiply by test function, integrate by parts:

$$\begin{aligned} 0 &= \int_{\mathrm{D}_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, \mathrm{d}x \\ &= \int_{\mathrm{D}_k} u_t \varphi - F(u) \cdot \nabla \varphi \, \mathrm{d}x + \int_{\partial \mathrm{D}_k} (\hat{n} \cdot F)^* \varphi \, \mathrm{d}S_x, \end{aligned}$$

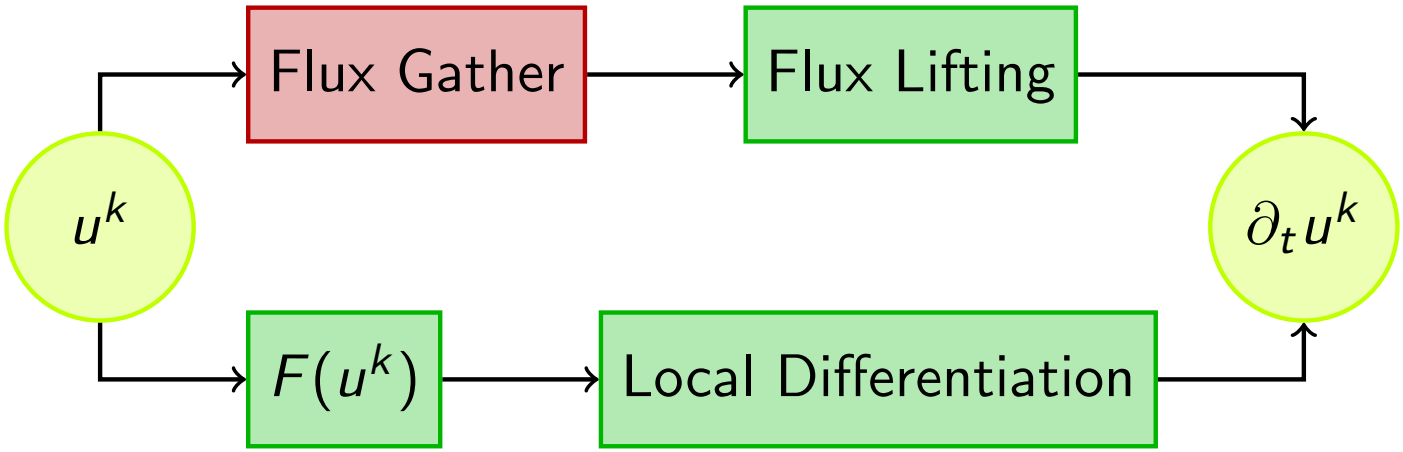
Integrate by parts again, subsitute in basis functions, introduce elementwise differentiation and "lifting" matrices D, L:

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

For straight-sided simplicial elements:
Reduce D^{∂_ν} and L to reference matrices.

Decomposition of a DG operator into Subtasks

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



Green: Element-local parts of the DG operator.

Note: Explicit timestepping, nodal representation.

BROWN

DG on GPUs: Possible Advantages

DG on GPUs: Why?

- GPUs have deep Memory Hierarchy
  - The majority of DG is local.
- Compute Bandwidth $\gg$ 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 mostly)
  - DG has built-in granularities, too ($\neq 2^n$ mostly)
- Much data reuse in DG (or any matrix product workload)
  - Very little cache memory available ($\sim$ 20 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

Metaprogramming DG: Flux Terms

$$0 = \int_{\mathrm{D}_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, \mathrm{d}x - \underbrace{\int_{\partial \mathrm{D}_k} [\hat{n} \cdot F - (\hat{n} \cdot F)^*] \varphi \, \mathrm{d}S_x}_{\text{Flux term}}$$

Metaprogramming DG: Flux Terms

$$0 = \int_{\mathrm{D}_k} u_t \varphi + [\nabla \cdot F(u)] \varphi \, \mathrm{d}x - \underbrace{\int_{\partial \mathrm{D}_k} [\hat{n} \cdot F - (\hat{n} \cdot F)^*] \varphi \, \mathrm{d}S_x}_{\text{Flux term}}$$

Flux terms:

- vary by problem
- expression specified by user
- evaluated pointwise

Metaprogramming DG: Flux Terms Example

Example: Fluxes for Maxwell's Equations

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

Metaprogramming DG: Flux Terms Example

Example: Fluxes for Maxwell's Equations

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

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

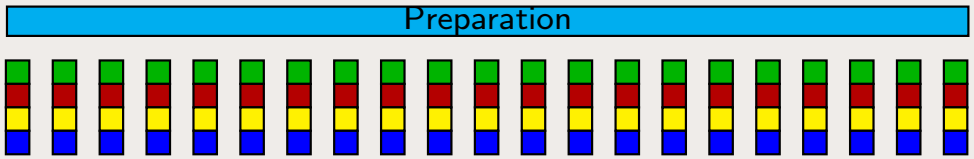
$$\hat{n} \cdot (F - F^*)_E := \frac{1}{2} [\hat{n} \times ([\![H]\!] - \alpha \hat{n} \times [\![E]\!])]$$

We generate: Scalar evaluator in C

```
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

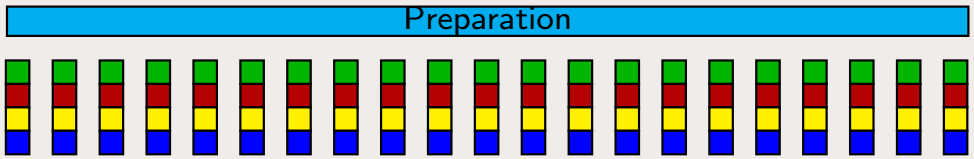
Setting: N independent work units + preparation



Question: How should one assign work units to threads?

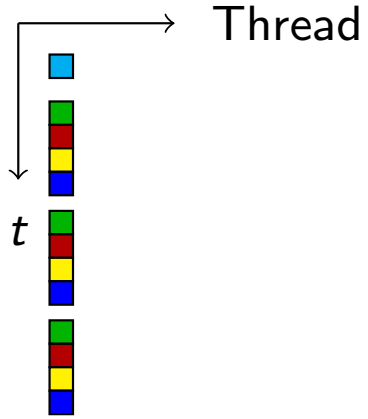
Loop Slicing on the GPU: A Pattern

Setting: N independent work units + preparation



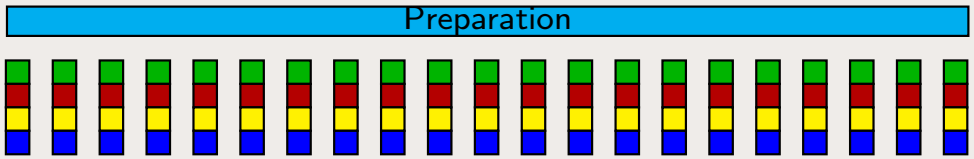
Question: How should one assign work units to threads?

w_s: in sequence



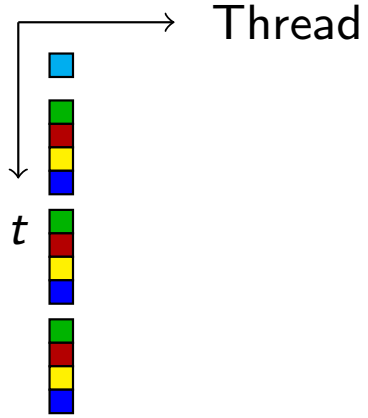
Loop Slicing on the GPU: A Pattern

Setting: N independent work units + preparation

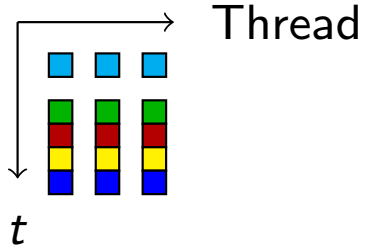


Question: How should one assign work units to threads?

w_s: in sequence

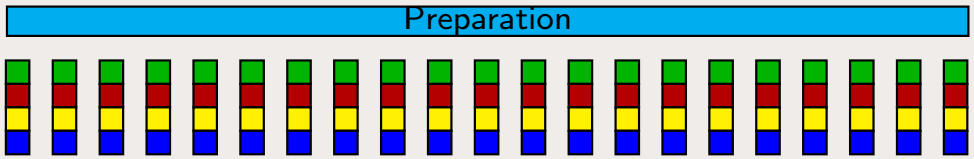


w_p: in parallel



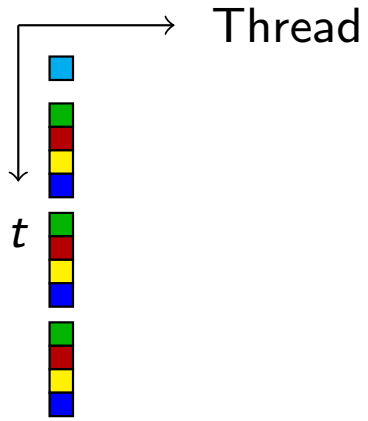
Loop Slicing on the GPU: A Pattern

Setting: N independent work units + preparation

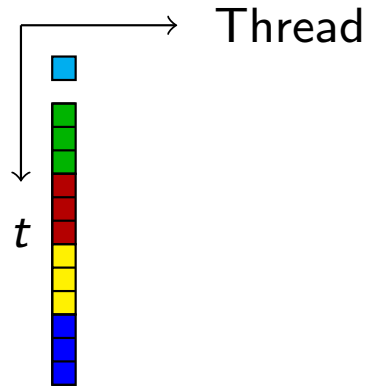


Question: How should one assign work units to threads?

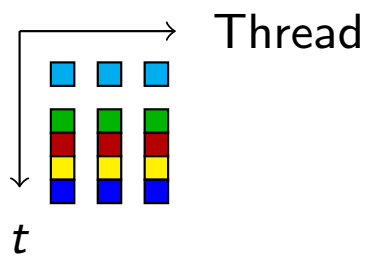
w_s: in sequence



w_i: "inline-parallel"



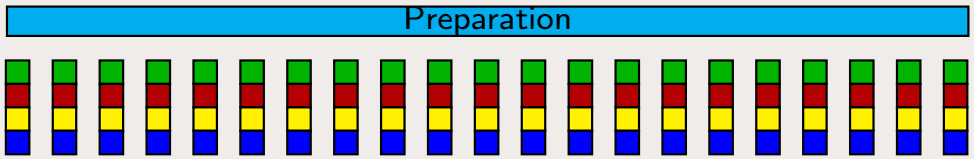
w_p: in parallel




BROWN

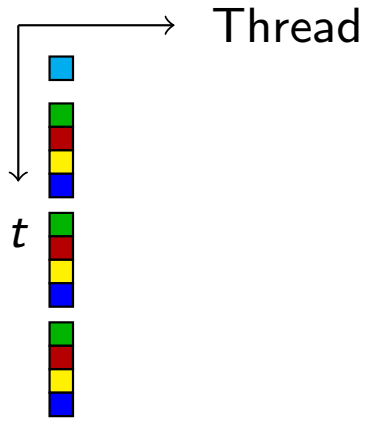
Loop Slicing on the GPU: A Pattern

Setting: N independent work units + preparation

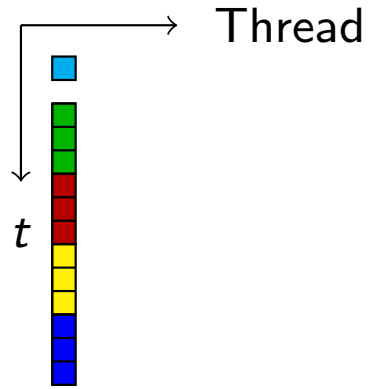


Question: How should one assign work units to threads?

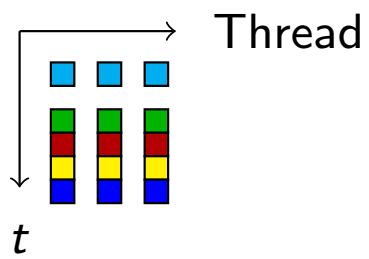
w_s: in sequence



w_i: "inline-parallel"



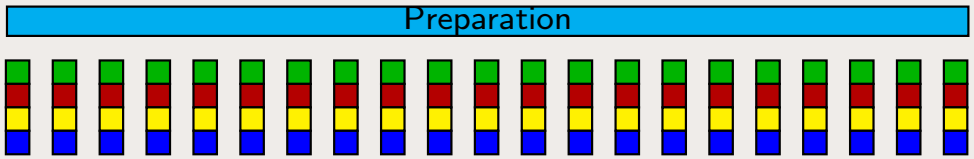
w_p: in parallel



(amortize preparation)

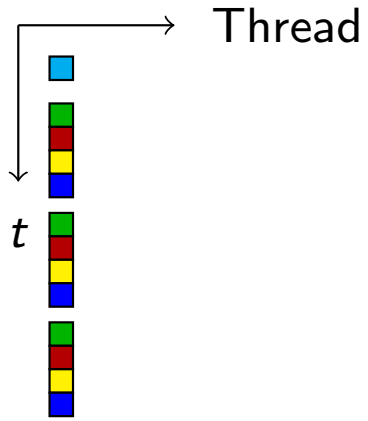
Loop Slicing on the GPU: A Pattern

Setting: N independent work units + preparation



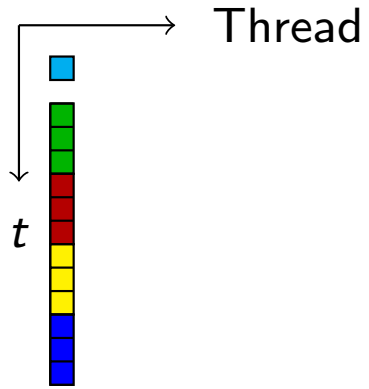
Question: How should one assign work units to threads?

w_s: in sequence



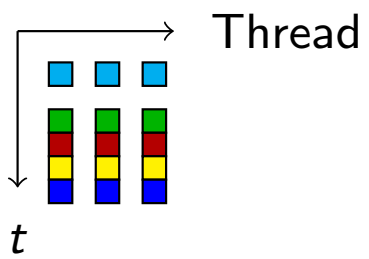
(amortize preparation)

w_i: "inline-parallel"

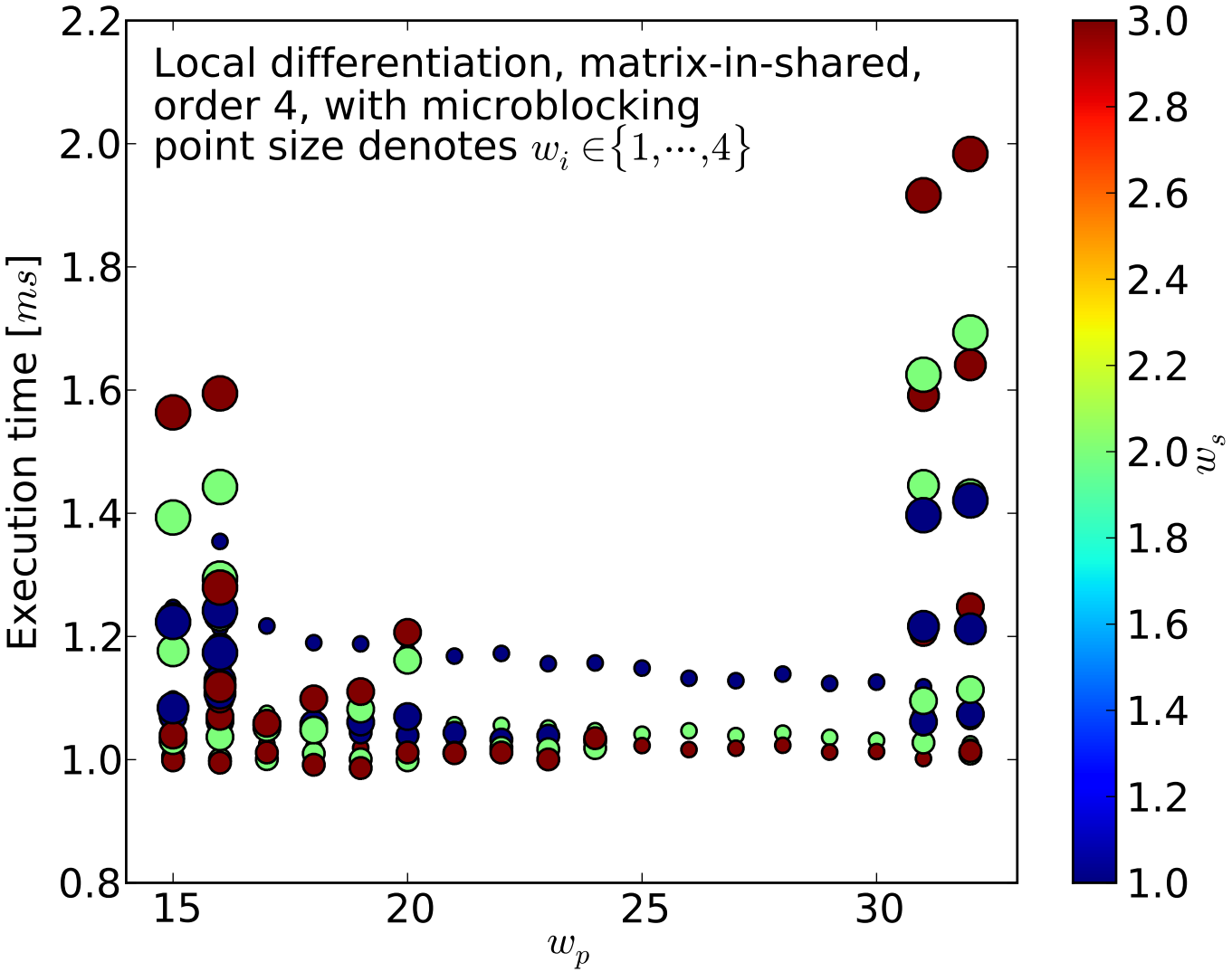


(exploit register space)

w_p: in parallel

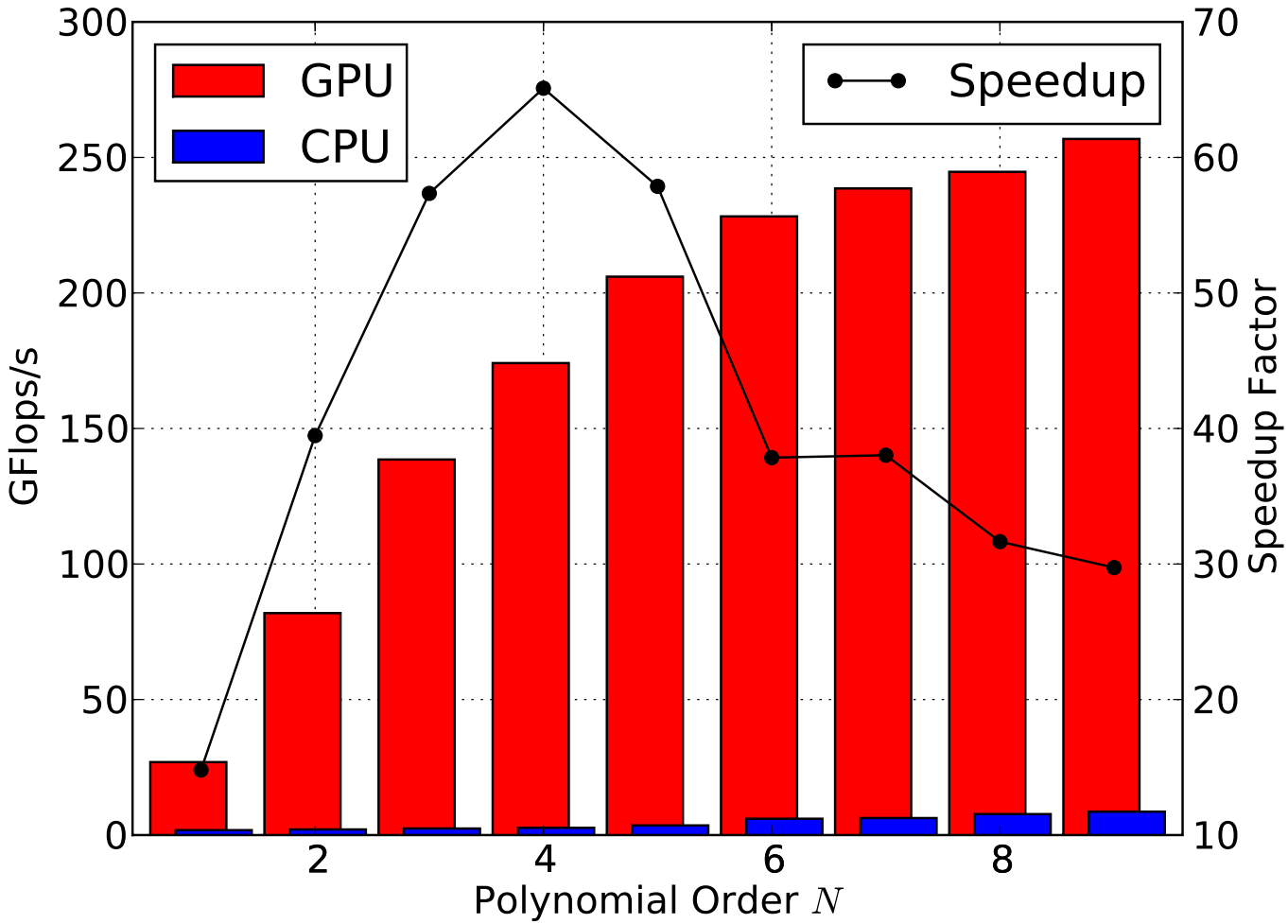


Loop Slicing for Differentiation

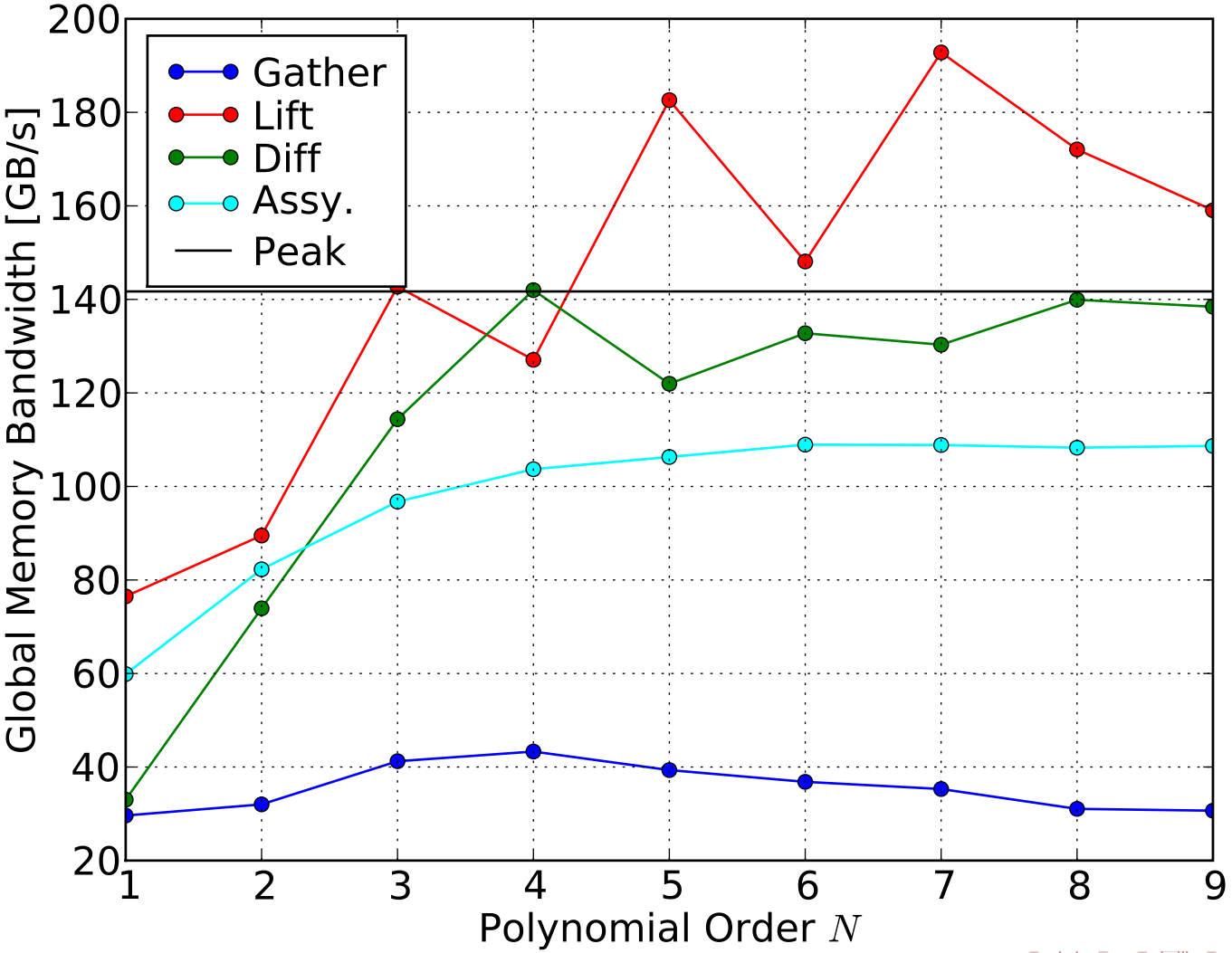


BROWN

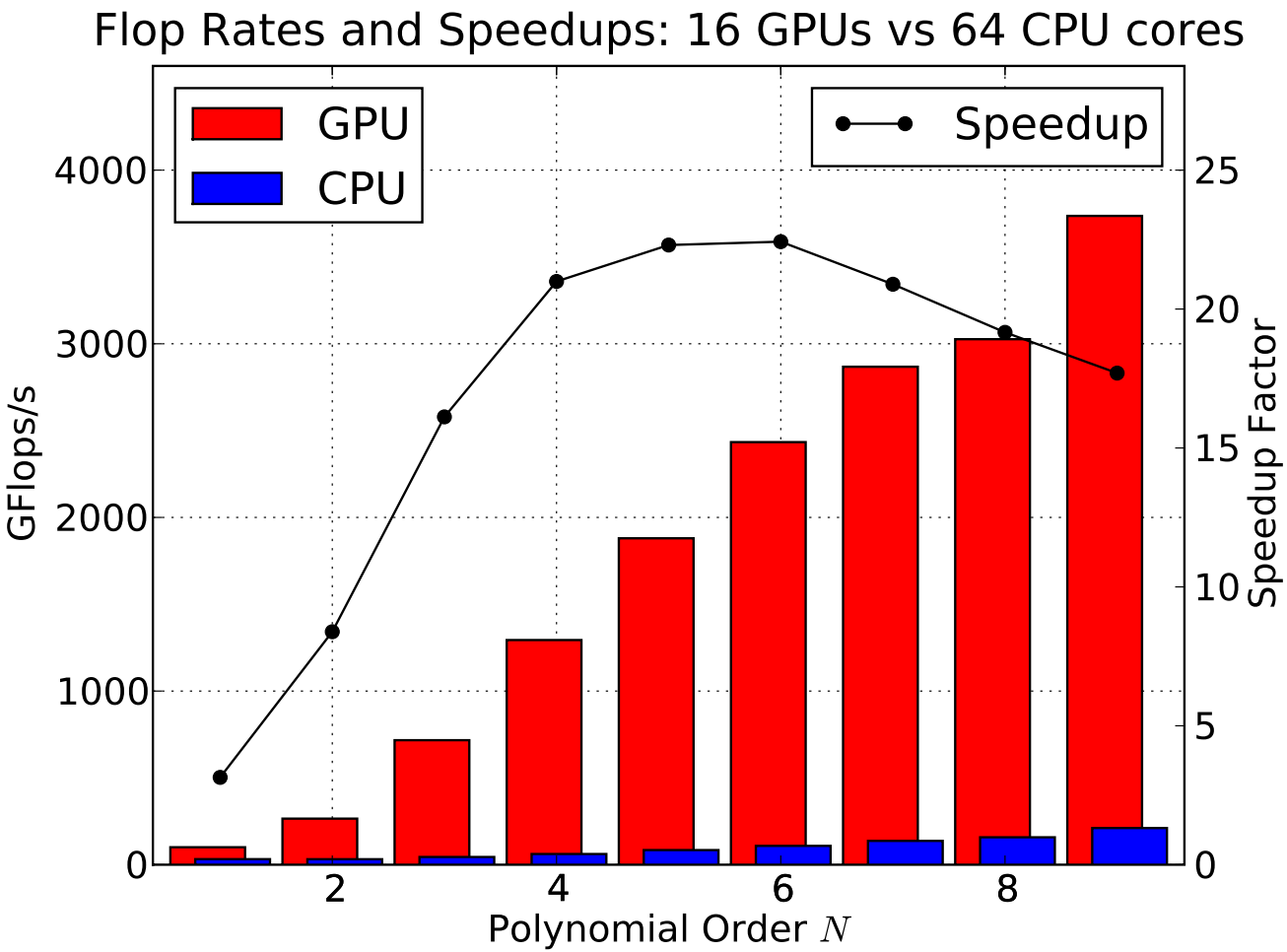
Nvidia GTX280 vs. single core of Intel Core 2 Duo E8400



Memory Bandwidth on a GTX 280

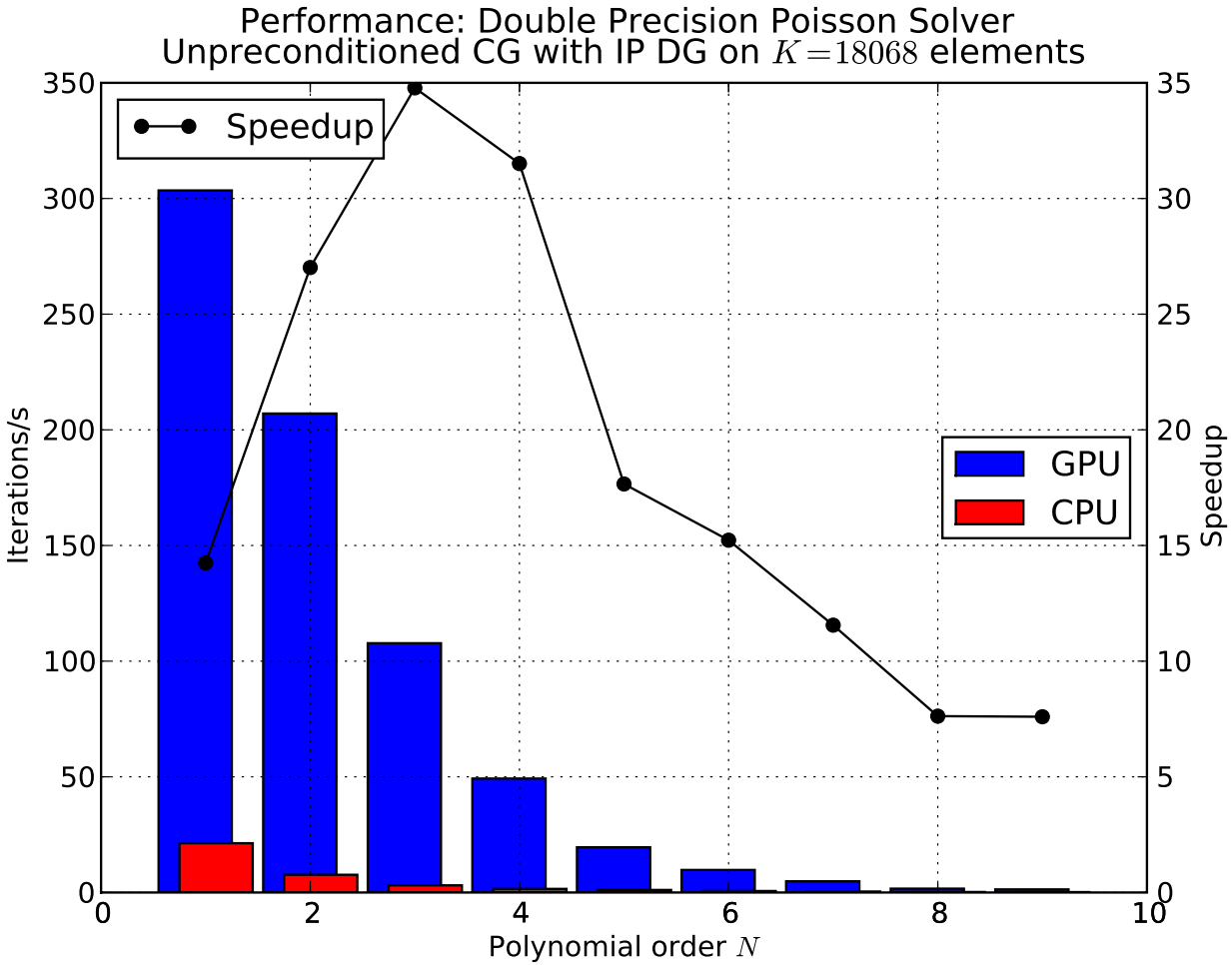


16 T10s vs. $64 = 8 \times 2 \times 4$ Xeon E5472

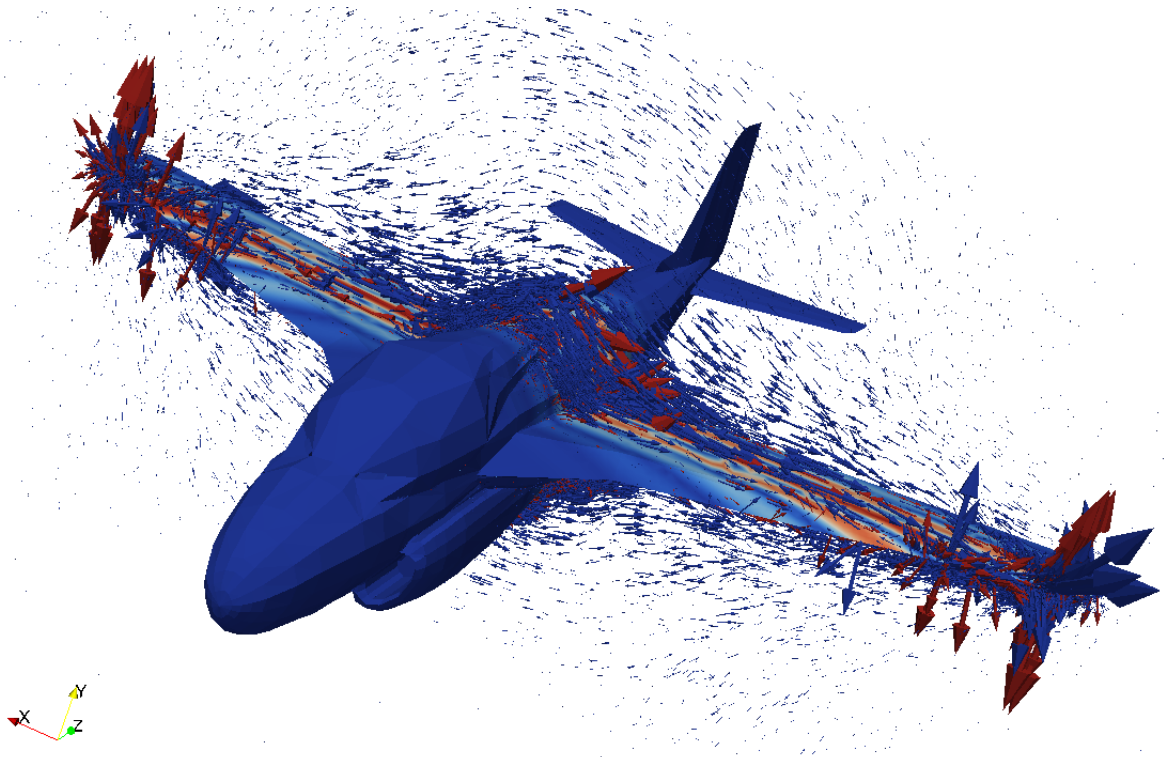


BROWN

Double Precision, T10 vs Xeon (Poisson)



"Real-World" Scattering Calculation



Order $N = 4$,
78745 elements,
$2.7M \cdot 6$ DOFs,
single Tesla C1060.

BROWN

Outline

BROWN

Automating GPU Programming

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

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

BROWN

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

BROWN

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

Loo.py Example

Empirical GPU loop optimization:

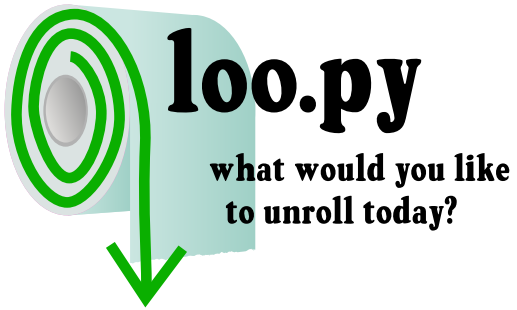
```
a, b, c, i, j, k = [var(s) for s in "abcijk"]
n = 500
k = make_loop_kernel([
    LoopDimension("i", n),
    LoopDimension("j", n),
    LoopDimension("k", n),
    ], [
    (c[i+n*j], a[i+n*k]*b[k+n*j])
    ])

gen_kwargs = {
        "min_threads": 128,
        "min_blocks": 32,
        }
```

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

BROWN

Loo.py Status

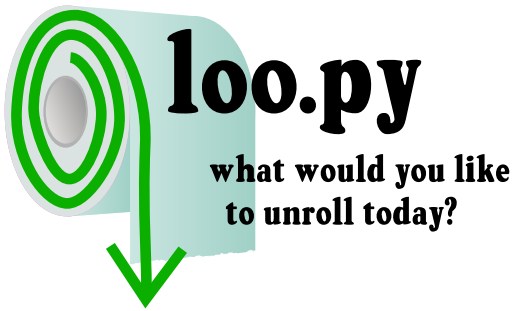
- Limited scope:
  - Require input/output separation
  - Kernels must be expressible using "loopy" model
    (i.e. indices decompose into "output" and "reduction")
  - Enough for DG, LA, FD, ...



BROWN

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



BROWN

Iterative CUDA

- GPU-based sparse linear system solver library (CG so far, trivial to add more)
- Built-in GPU Sparse Matrix-Vector multiplication[1]
- 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× performance gain in SP *and* DP
- Same functionality also in PyCUDA

BROWN

[1] PKT format, Bell/Garland 2008

Conclusions

- Two technologies "ready for prime-time" now:
  - Scripting
  - GPUs

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

AK, T. Warburton, J. Bridge, J.S. Hesthaven, *"Nodal Discontinuous Galerkin Methods on Graphics Processors"*, J. Comp. Phys., to appear.

→ `http://arxiv.org/abs/0901.1024`

BROWN

Questions?

?

Thank you for your attention!

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

http://arxiv.org/abs/0901.1024

▸ image credits

BROWN

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