Chemora!*

Computational Hierarchy for Engineering Model-Oriented Re-adjustable Applications [tm]

Chemora Kernel Mapping Optimization

Steven R. Brandt[1] David M. Koppelman[2] Yue Hu[2]

Louisiana State University
Baton Rouge, LA U.S.A.

[1] Division of Computer Science
[2] Division of Electrical & Computer Engineering

Center for Computation and Technology

*This work supported in part under US NSF grant ACI-1265449.

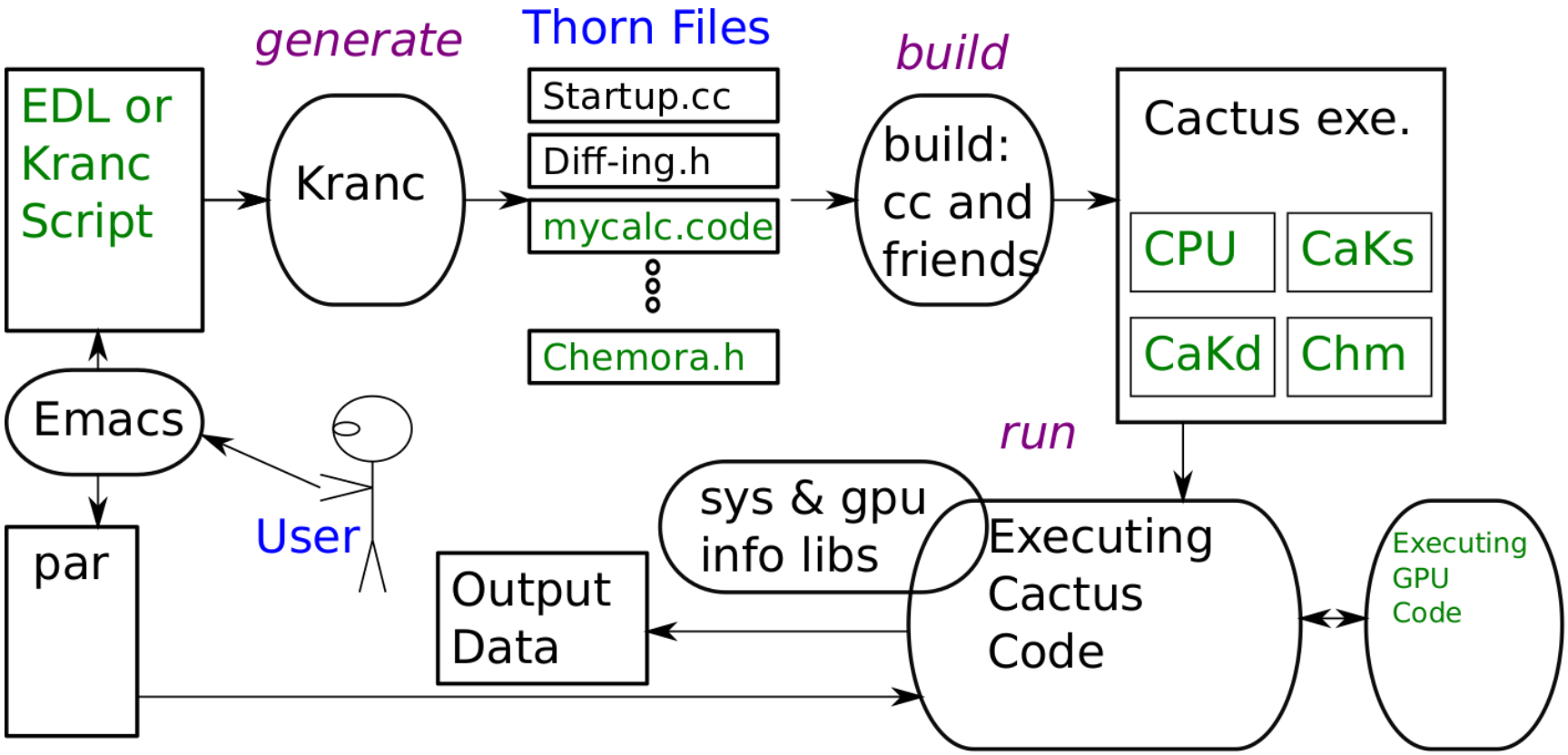
Chemora! Goals

Someday in the near future

> Write a differential equation description of some system . . .
> . . . enter a simple command . . .
> . . . the Chemora system will generate highly tuned code. . .
> . . . tailored to your system and simulation parameters.

Overview

From EDL to GPU Code



Formatted 13:01, 27 August 2015 from chemora.

et15-3

From EDL to GPU Code

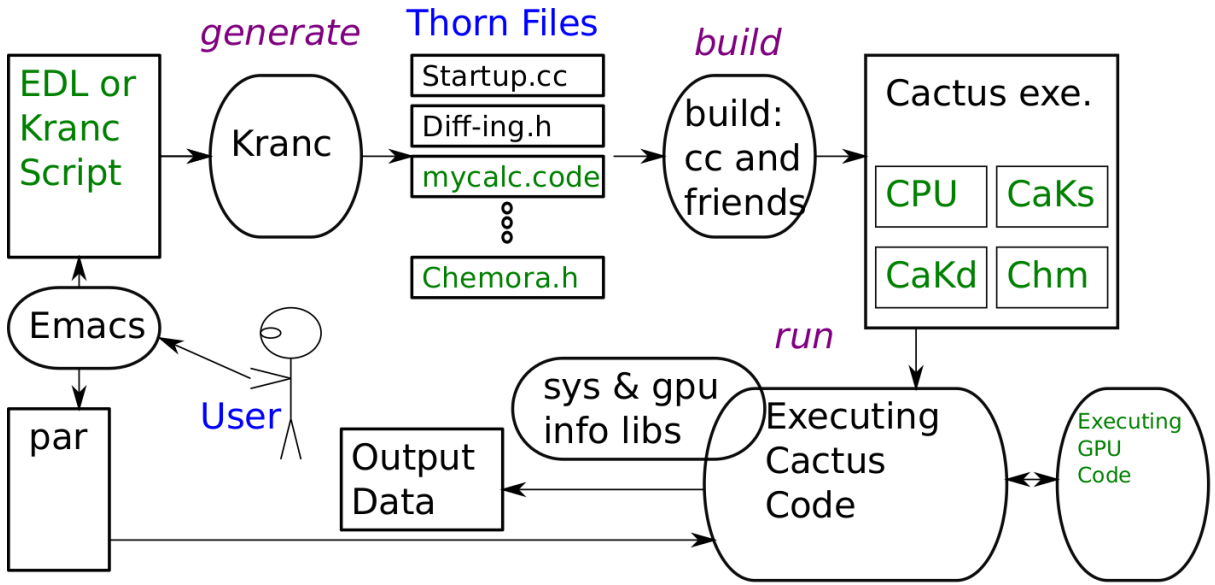
User writes EDL or Kranc script.

User writes parameter file describing simulation.

User runs Kranc to *generate* a Cactus thorn.

User *builds* Cactus to generate a Cactus executable.

User *runs* Cactus executable.



et15-4

Cactus Executable Contents

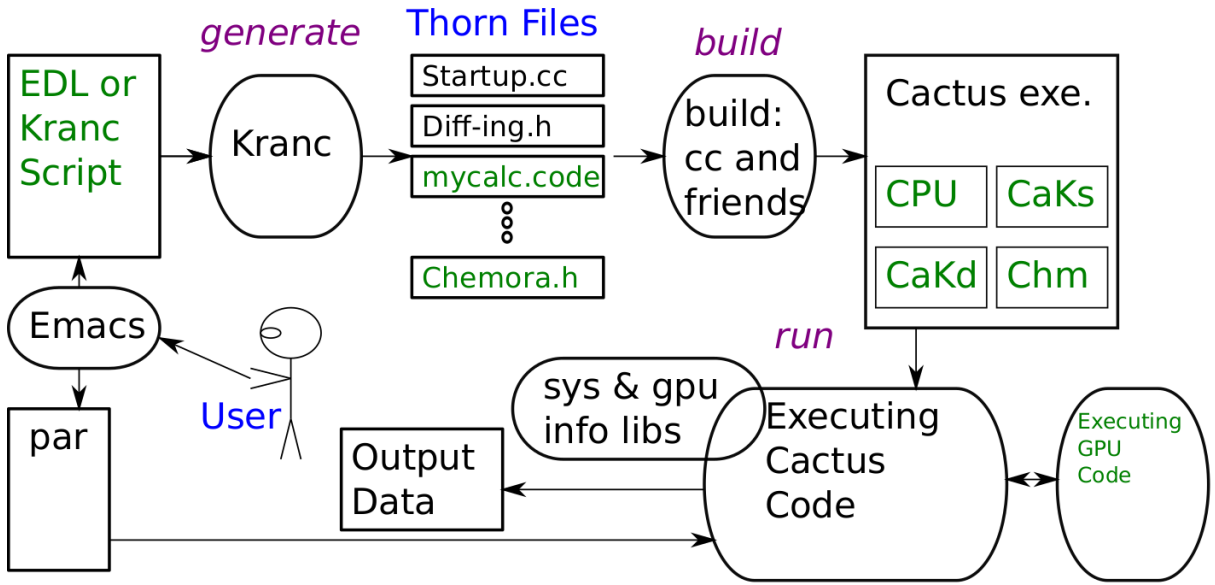
Executable has simulation code in four forms:

CPU code.

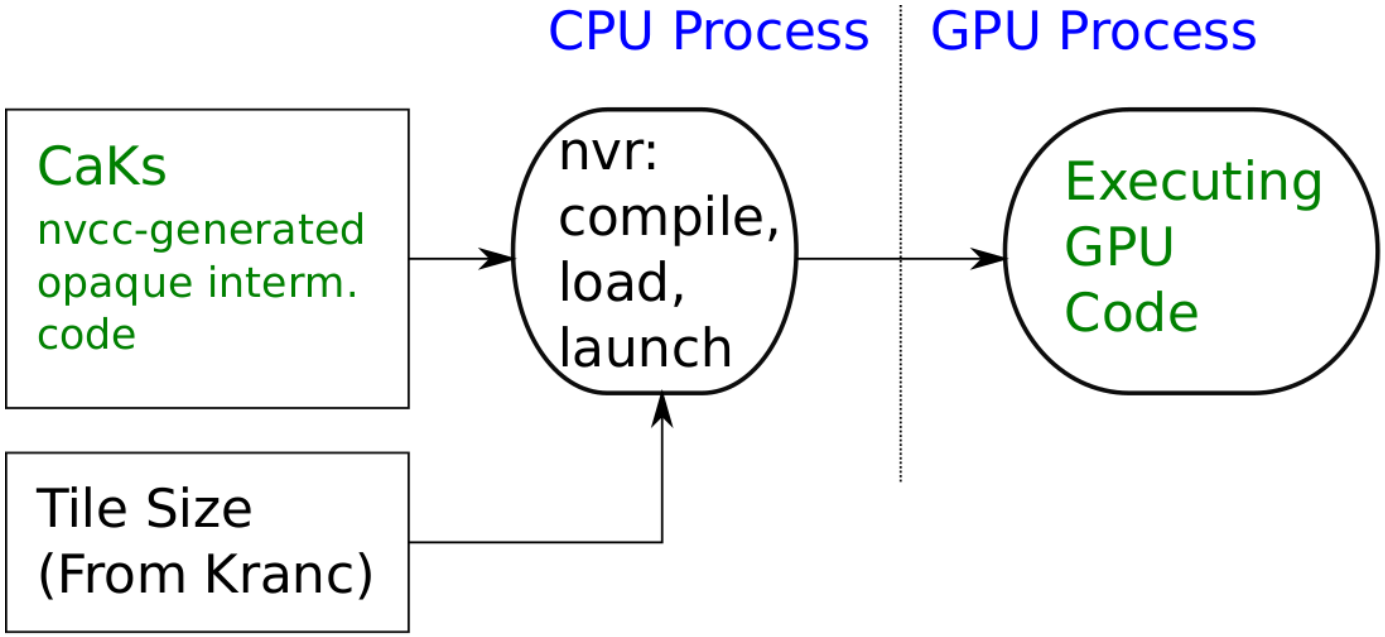
CaKernel static code (CaKs), for GPU.

CaKernel dynamic code (CaKd), for GPU.

Chemora intermediate code (Chm), for GPU.



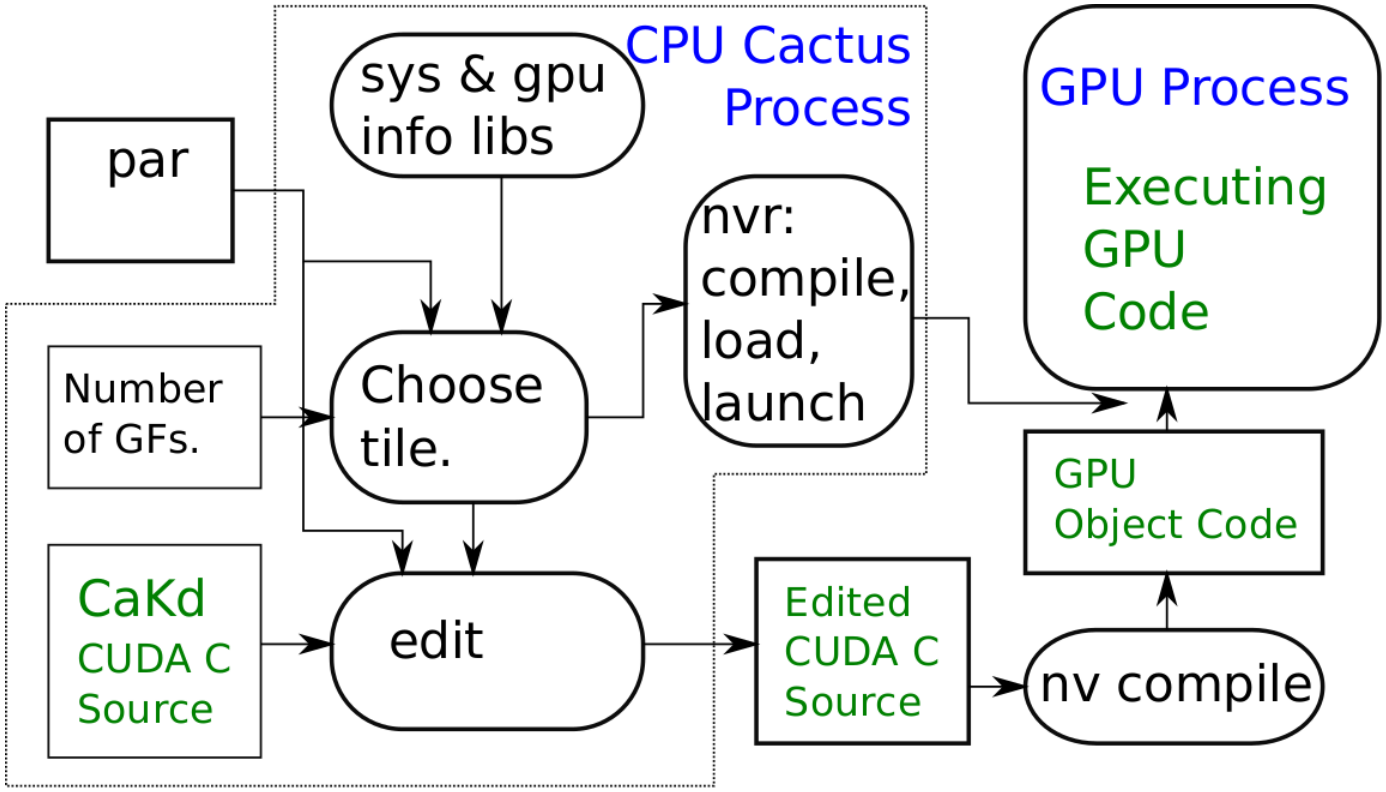
CaKernel Static Code



GPU run of Kranc-generated code using Kranc-chosen tile.

Optimal tile size requires optimally patient user.

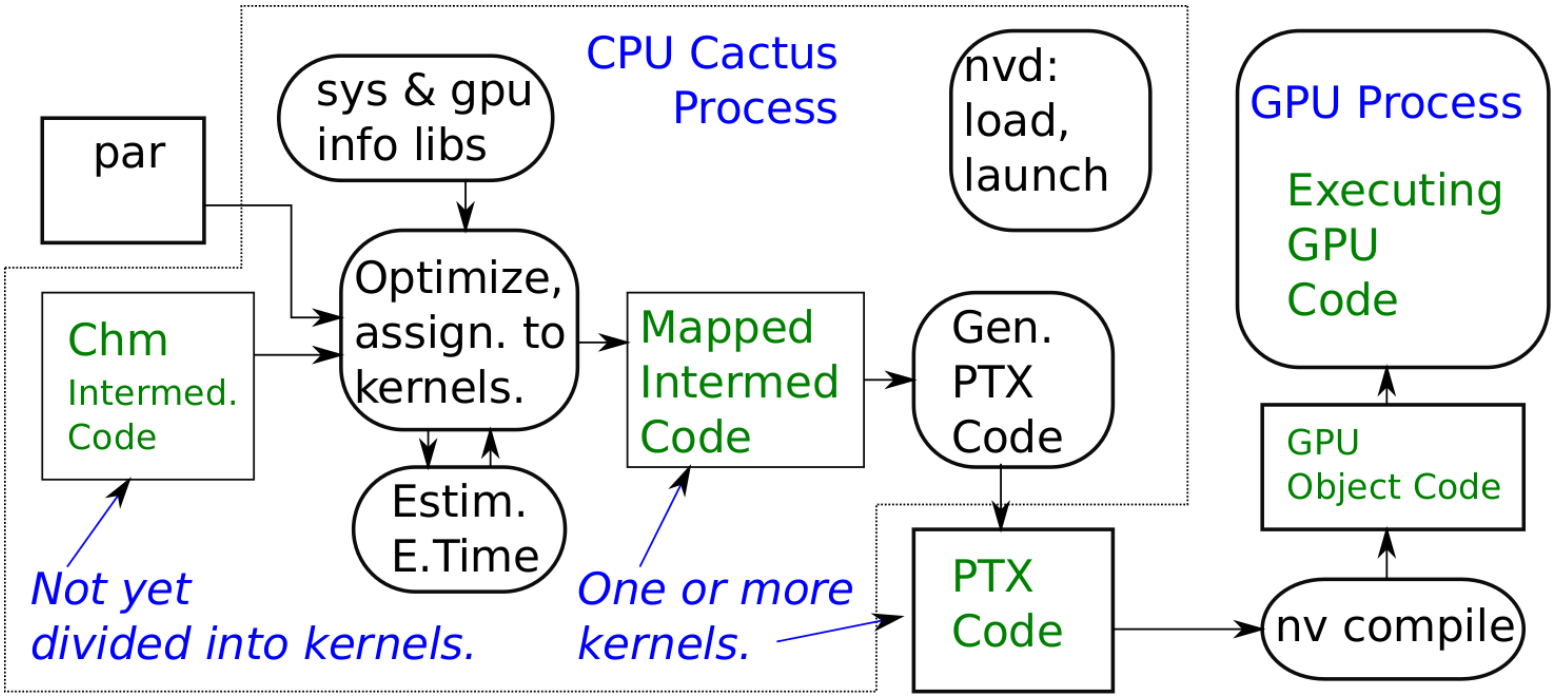
CaKernel Dynamic Code



Tile sizes chosen dynamically, based on system and application parameters.

Code dynamically compiled using tile size and app parameters.

Chemora!



Calculation dynamically mapped to kernels based on app and sys params.

Emit code in PTX to avoid compiler front end sillyness.

Chemora's Optimization of a Calculation

Elements of a Calculation

→ Operates on a grid point, say (i, j, k).

→ Reads grid functions at small *offsets* from grid point, say $(i-1, j, k)$.

→ Performs arithmetic operations.

→ Writes grid functions at grid point.

Mathematically: $x_{i,j,k} = 7a_{i,j,k} + 3a_{i-1,j,k} + b_{i,j+1,k}$

As C Code:

```
I3D[x,0,0,0] = 7 * I3D[a,0,0,0] + 3 * I3D[a,-1,0,0] + I3D[b,0,1,0];
```

Execution of a calculation.

Calculation must operate over a grid of points, for example:

```
for ( int i=0; i<imax; i++ )
 for ( int j=0; j<jmax; j++ )
  for ( int k=0; k<kmax; k++ )

   I3D[x,0,0,0] = 7 * I3D[a,0,0,0] + 3 * I3D[a,-1,0,0] + I3D[b,0,1,0];
```

In a perfect world:

```
#pragma smartmp parallelize target auto

for ( int i=0; i<imax; i++ )
 for ( int j=0; j<jmax; j++ )
  for ( int k=0; k<kmax; k++ )

   I3D[x,0,0,0] = 7 * I3D[a,0,0,0] + 3 * I3D[a,-1,0,0] + I3D[b,0,1,0];
```

Fundamental Performance Limits

Saturation Analysis

Hardware Limiters

FP Rate: $\theta_{\text{fp}} = 1\,\text{TFLOPS}$.

Data Rate: $\theta_{\text{data}} = 1\,\text{GB/s}$.

Problem Requirements

FP: $p_{\text{fp}} = 10$ FP operations per active grid point.

Data (r+w): 4 grid functions, 32 B per grid point.

Grid Size: n^2 points, stencil size 1×1.

Formatted 13:01, 27 August 2015 from chemora.

et15-12

Execution Time Limit

Based on FP: $(n-2)^2 10/\theta_{\text{fp}}$.

Based on Data: n^2/θ_{data}.

$\Rightarrow$Nothing is wasted.

Working Set v. Parallelism

Locality Domain:

The collection of high-speed storage available to a set of threads.

For this talk, locality domain imposed by the *CUDA Multiprocessor*.

Working Set:

The set of data that will soon be reused by some piece of code.

Accommodating locality domains imposes re-use and communication overheads.

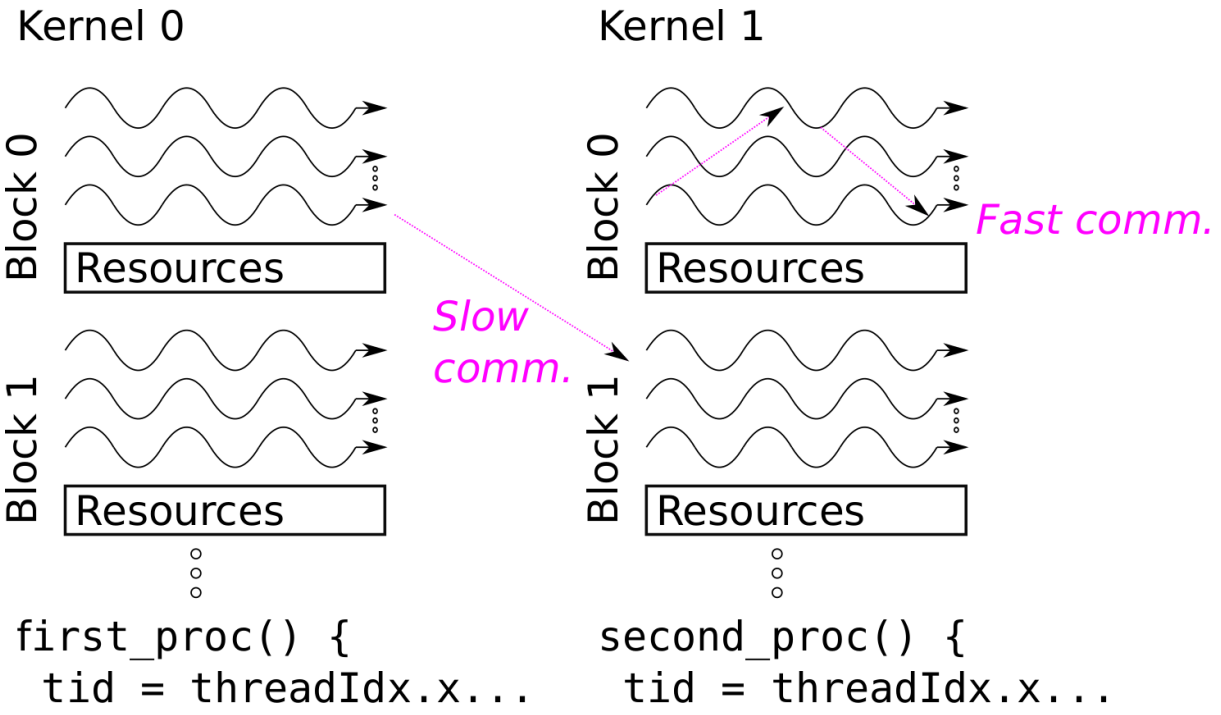
Spatial accommodation: utilizing all available MPs.

Temporal accommodation: using the same MP multiple times.

NVIDIA GPU Organization Highlights

NVIDIA CUDA GPU code executes in units called *kernels* which consists of threads organized into a *blocks*.

All threads in a kernel start execution at the same place.



Formatted 13:01, 27 August 2015 from chemora.

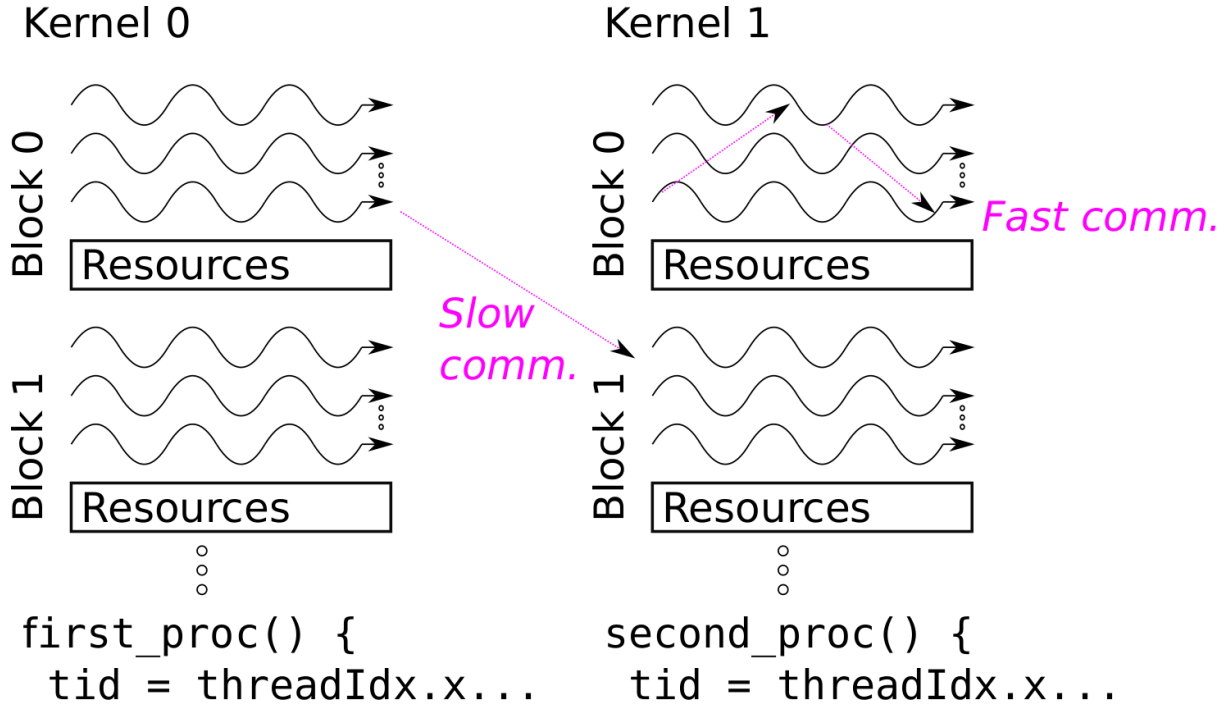
et15-15

NVIDIA GPU Organization Highlights

Blocks are assigned to *multiprocessors* (MPs) for execution.

Threads in a block share an MP's *resources*.

Resources include *registers* and *shared memory* . . .
. . . both of which serve the same purpose as the L1 cache in a CPU.

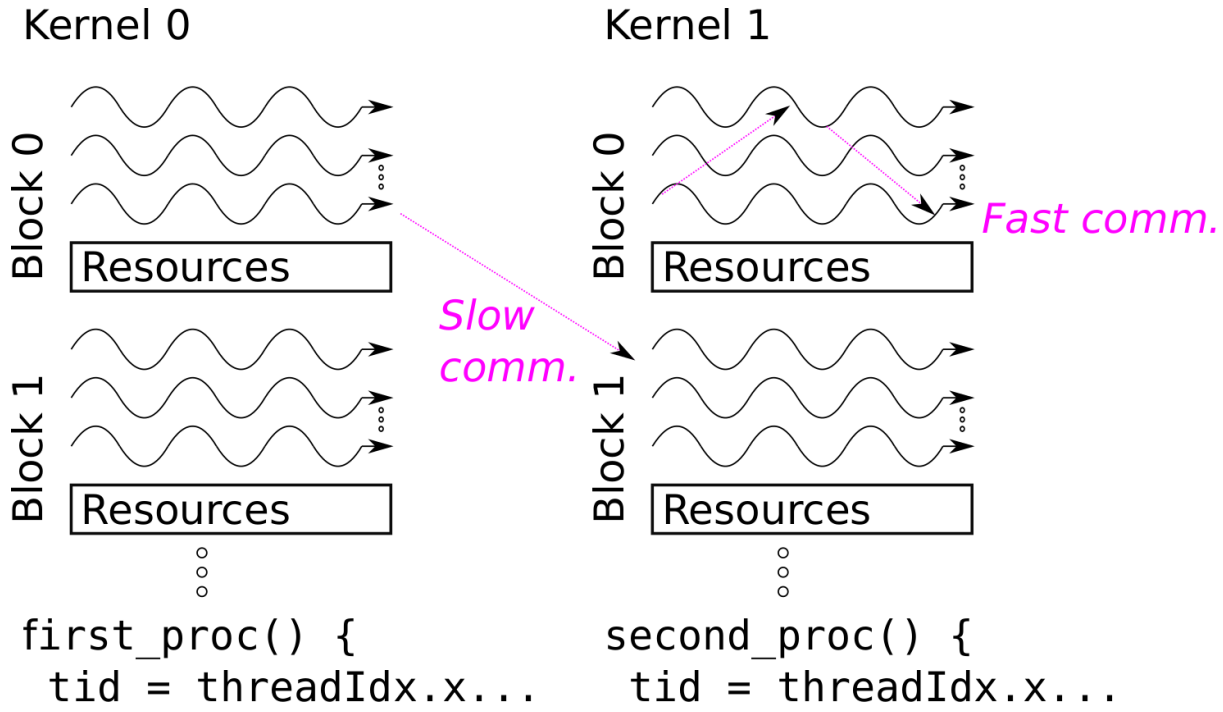


Formatted 13:01, 27 August 2015 from chemora.
et15-16

NVIDIA GPU Organization Highlights

There are no latency reduction and hiding mechanisms . . .

. . . other than a huge number of threads.



Formatted 13:01, 27 August 2015 from chemora.

et15-17

Tradeoffs In Kernel Size (Two Senses)

Increasing the number of threads in a block.

:-) Reduces stencil buffering overhead.

:-) Hides more latency.

But, increasing the number of instructions in a thread.

:-(Reduces the maximum number of threads.

:-O Can lead to register spill/spills ...

... which are very costly on a GPU.

Degrees of Freedom for Generating Code

Assignment of iterations to threads and thread block shape.

Buffering of re-used values.

Mapping of calculation to multiple kernels.

Reassociation to reduce communication and storage size.

Etc.

How Chemora Helps

Quickly explore a large space of configurations.

Speed is due to our *model-driven autotuning* technique.

The Chemora! Approach

Model-Driven Autotuning

- Develop an accurate execution time model.
- Generate code alternatives and execution configurations.
- Use model to choose between alternatives. (Tile shape, loop fission.)
- Generate code after finishing model-driven search.

Those Other Guys

Execution-Driven Autotuning:

An autotuning method in which configuration alternatives are chosen using a sample run.

- Generate execution configurations.
- Run (*static*) or compile and run (*dynamic*) . . .
  . . . code and measure execution time.
- Choose configuration based on measured execution time.

Kamil 10 IPDS

Zhang 12 CGO

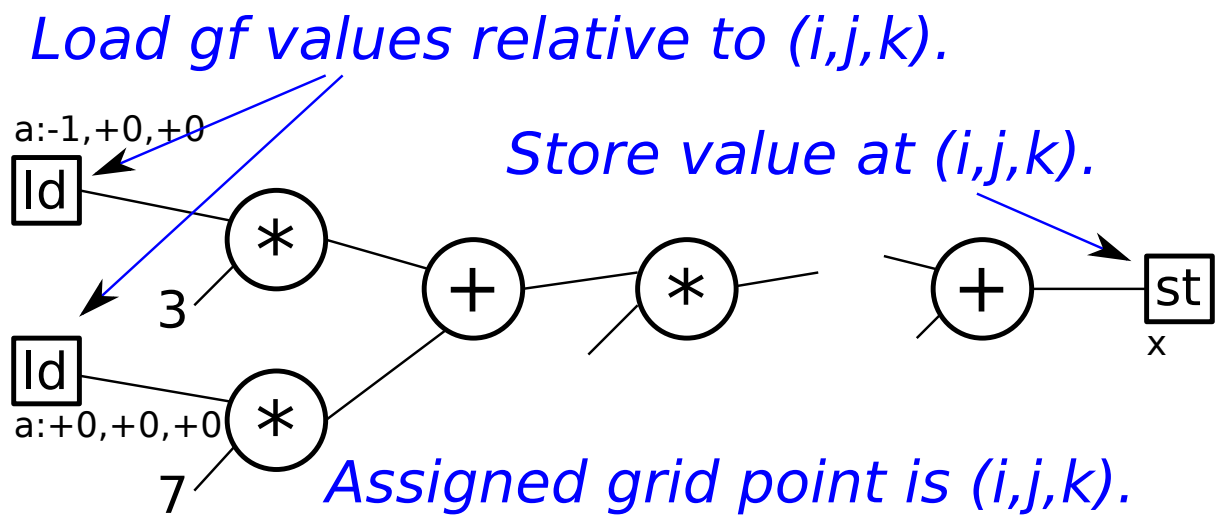
Williams 11 SC

Some Terminology

Calculation:

A DAG whose sources load grid function values at some offset from the assigned grid point, whose interior nodes perform arithmetic operations and whose sinks store grid function values at the assigned grid point.

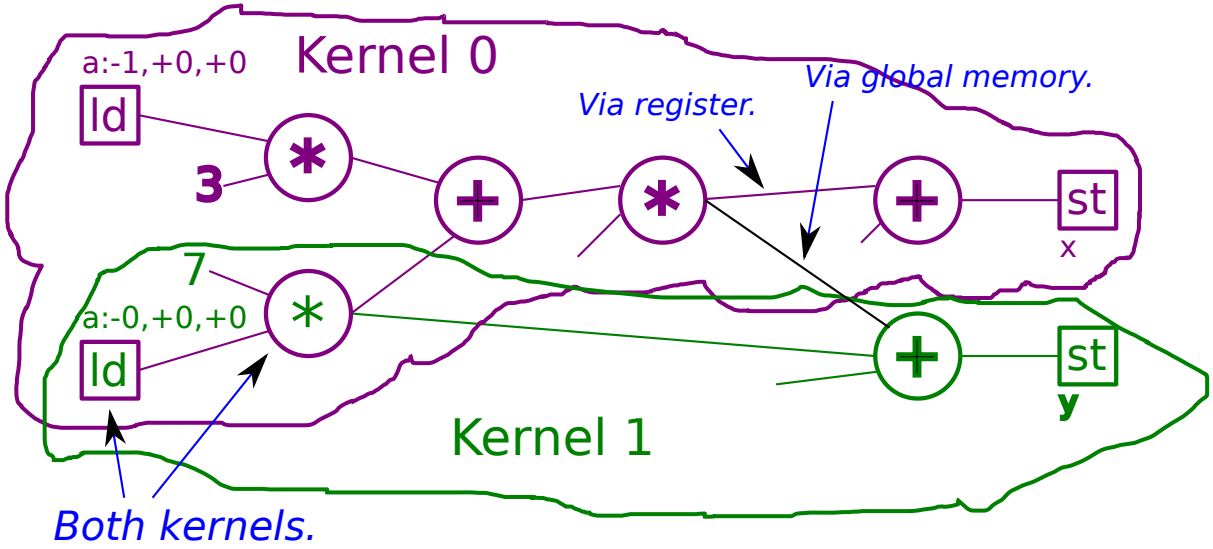
Example:



Mapping:

An assignment of nodes in a calculation to kernels,
the choosing of CUDA block dimensions,
the assignment of grid space to threads,
and other details needed to delineate generated code.

Example:



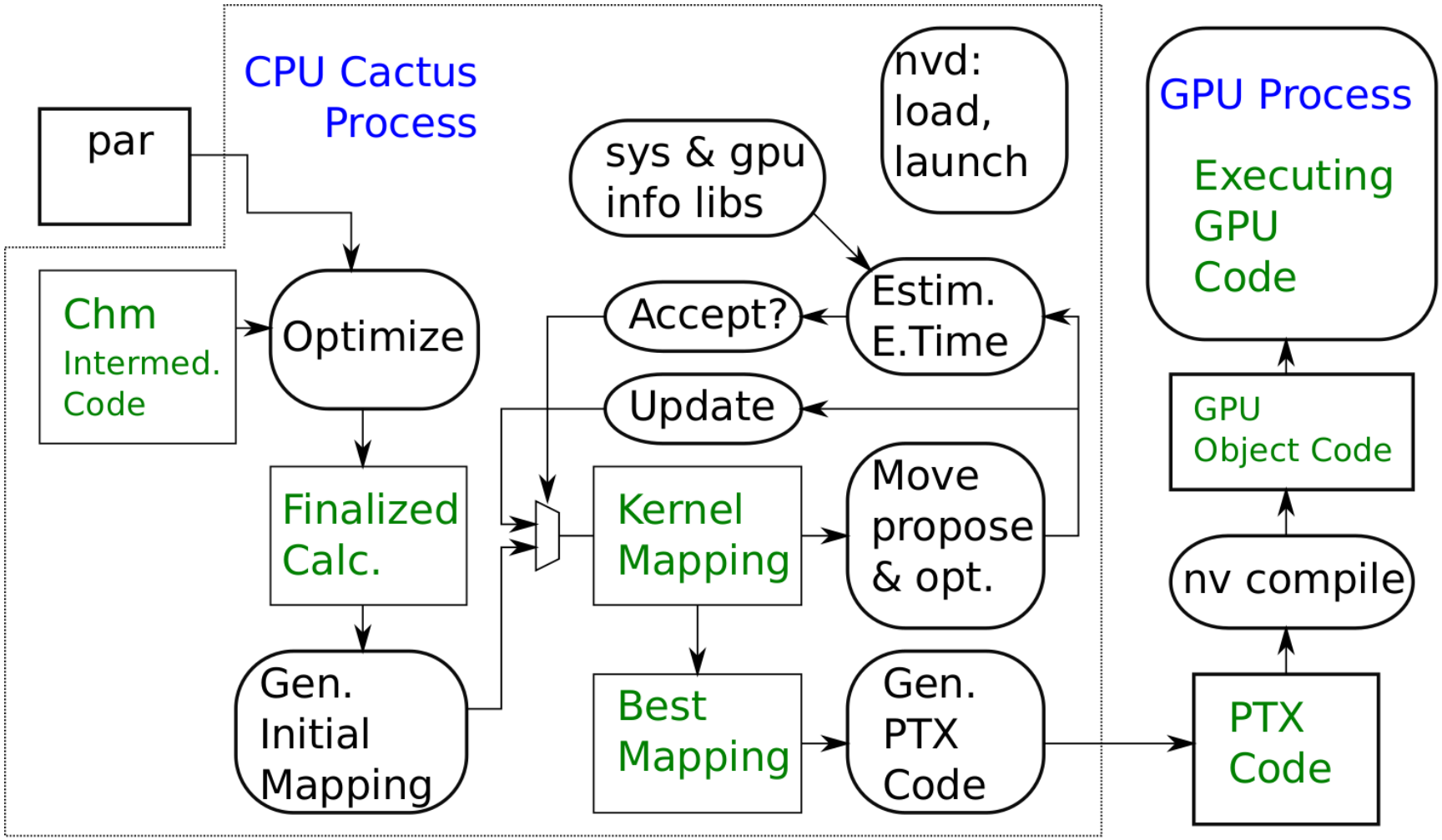
Kernel 0: Block 256 threads. Tile: $64 \times 4 \times 1$ threads.

Kernel 1: Block 1024 threads. Tile: $128 \times 8 \times 1$ threads.

Code Generation:

The process of generating source code given some mapping of a calculation.

The Chemora Approach



et15-26

Advantages of Chemora's Model-Driven Autotuning

Much faster than execution-driven autotuning.

Refinement of the model helps refine search.

Mapping Generation

Goal: Fast search.

Steps:

- Node assignment.
- Optimization.
- Tile configuration.
- Execution time estimation.

Node Assignment

Objective: Assign each node to one or more kernels.

Starting point can be a calculation or an existing mapping.

Calc → Mapping

Single kernel (all nodes to one kernel).

Multiple kernels, one GF store per kernel.

Random assignment to k kernels.

Mapping → Mapping

A *proposed move* specifies nodes to move or copy.

Move Proposal Optimization

Add to proposal for correctness.

Dead code elimination.

Tile Configuration

Determine:

- Method used to access grid function values.
- Number of threads, thread assignment and iteration.

Currently uses a deterministic procedure

Procedure

- Based on stencil determine tile shape and iteration direction.
- Determine thread limit imposed by register use.
- Determine tile size imposed by shared memory use.

Performed in a deterministic fashion—for now.

Execution Time Estimation

Based on move proposal, not on a complete re-tallying.

Importance of Execution Time Estimation

et15-32

Performance Model

Purpose: Estimate execution time of a kernel.

Applied to all kernels to get an overall execution time.

Based on hardware model, not on curve fitting.

An indispensable part of the optimization process.

et15-33

Performance Model

Three Kernel Components Plus Launch Overhead

- s_i: Instruction Issue Time
- s_d: Off-Chip Data Volume
- s_L: Iteration Latency
- s_k: Kernel Launch Overhead

Formatted 13:01, 27 August 2015 from chemora.
et15-34

Performance Model—Instruction Issue

Scaled instruction issue time denoted s_i.

This is the easy one.

Based on estimated tally of instructions.

Easy because of PTX compiler's predictability.

$s_i = \sum_{c \in C} \frac{n_c}{\theta_c}$.

where C is a set of instruction classes ...
... n_c is the number of instruction in a calculation of class c ...
... and θ_c is the target device MP throughput for c.

Example

Calculation has 21 MADD instructions.

MP has throughput of 64 MADD insn per cycle.

$s_i = \frac{21}{64} + \cdots$

et15-36

Off-Chip Data Volume

Assume 0% L2 Cache Hit Ratio

Account for size of ghost zone.

Account for buffering.

Easy because we aren't using the RO cache yet.

Iteration Latency

Quick and dirty estimate (so far).

Based on mix of instructions, not dependence chains.

Scaled based on the number of threads in a block.

On the flight back: Use dataflow graph to get latency.

Execution Time Estimate

Scaling of Estimate

Components scaled to cycles. . .
. . . per grid point. . .
. . . per multiprocessor. . .
. . . per time step.

Let s_d denote the scaled time for off-chip data.

Total time for off-chip data is $E\frac{N}{M}s_d$ . . .
. . . where E is the number of time steps. . .
. . . N is the number of grid points. . .
. . . and M is the number of multiprocessors.

Scaling of Latency

Let L denote latency for computing one grid point.

Scaled latency is L/τ, where τ is the number of threads in a block.

Estimated Execution Time

Let s_k denote the scaled kernel launch overhead.

Ideal Formula:

$$T = s_k + \max\{s_d, s_L, s_i\}$$

This assumes one component does not affect the other.

Current Formula:

$$T = s_k + \max\{s_d, s_L, s_i\} + 0.35(s_d + s_L + s_i - \max\{s_d, s_L, s_i\})$$

Code Generation

Original Target: CUDA C

Used by CaKernel (static and dynamic).

Used in earlier version of Chemora.

But difficult to coax compiler to generate tight code.

Compiler would emit bloated code sequences for computing addresses and iteration.

Current Chemora Target: PTX (CUDA Intermediate Form)

Assembler-like.

Need to compute byte addresses for data accesses.

But, fewer "surprise" code sequences.

Performance Evaluation Methodology

Hardware

NVIDIA Tesla K20c (Kepler generation, CC 3.5).

0.71 GHz, 5119 MiB global memory, 1280 kiB L2 cache.

Memory/L2 208.0 GB/s.

Build and OS Software

CUDA 7.0, V7.0.27

Data Collection

Kernel timing data collected using `cuEventRecord`.

Timing collected for kernel launch only . . .
. . . CPU $\leftrightarrow$ GPU data transfer omitted.

ML_BSSN Versions

Fixed:

Calculations have their natural size.

By Hand:

Calculations split by hand to improve performance.

BSSN Simulation Parameters

Ten iterations.

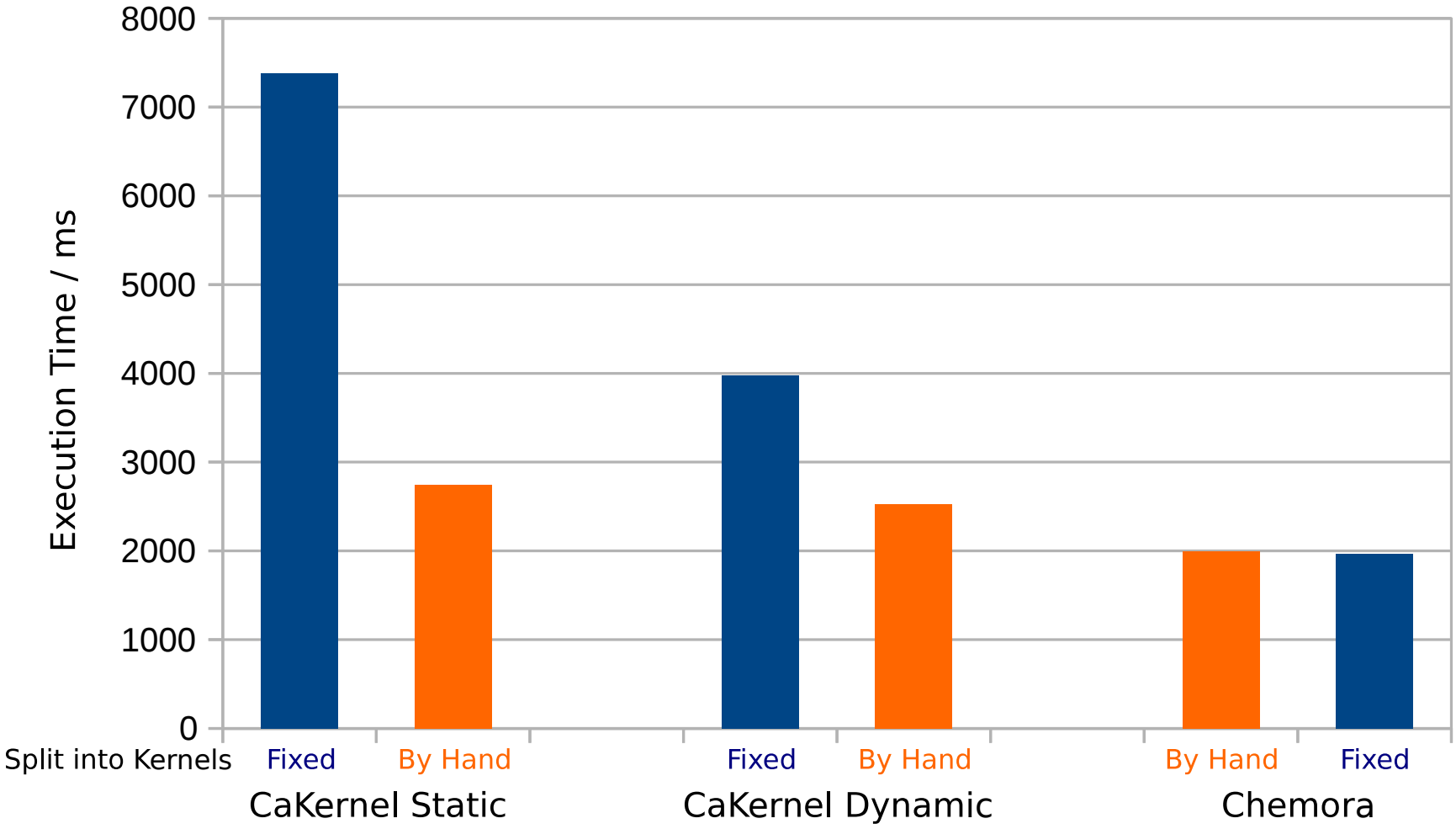
100^3 grid.

More BSSN Details.

```
ADMBase::dtshift_evolution_method = "ML_BSSN"

ML_BSSN::harmonicN           = 1      # 1+log
ML_BSSN::harmonicF           = 2.0    # 1+log
ML_BSSN::ShiftGammaCoeff     = 0.75
ML_BSSN::BetaDriver          = 1.0
ML_BSSN::LapseAdvectionCoeff = 1.0
ML_BSSN::ShiftAdvectionCoeff = 1.0

ML_BSSN::MinimumLapse                  = 1.0e-8
ML_BSSN::conformalMethod               = 1 # 1 for W, 0 for phi
ML_BSSN::my_initial_boundary_condition = "extrapolate-gammas"
ML_BSSN::my_rhs_boundary_condition     = "static" # Radiative does not work with CaKernel yet
ML_BSSN::apply_dissipation             = "never"

# This thorn has been built only with 8th order finite differencing

Boundary::radpower                     = 2
ADMBase::metric_type = "physical"

CoordBase::domainsize                   = minmax

CoordBase::boundary_size_x_lower        = 5
CoordBase::boundary_size_y_lower        = 5
CoordBase::boundary_size_z_lower        = 5
CoordBase::boundary_shiftout_x_lower    = 1
CoordBase::boundary_shiftout_y_lower    = 1
CoordBase::boundary_shiftout_z_lower    = 1

CoordBase::boundary_size_x_upper        = 5
CoordBase::boundary_size_y_upper        = 5
CoordBase::boundary_size_z_upper        = 5
CoordBase::boundary_shiftout_x_upper    = 0
CoordBase::boundary_shiftout_y_upper    = 0
CoordBase::boundary_shiftout_z_upper    = 0

CartGrid3D::type                        = "coordbase"
CartGrid3D::domain                      = "full"
CartGrid3D::avoid_origin                = "no"

CoordBase::xmin                         = 0
CoordBase::ymin                         = 0
CoordBase::zmin                         = 0

CoordBase::xmax                         = 100
CoordBase::ymax                         = 100
CoordBase::zmax                         = 100

CoordBase::dx                           = 1
CoordBase::dy                           = 1
CoordBase::dz                           = 1
```

Results

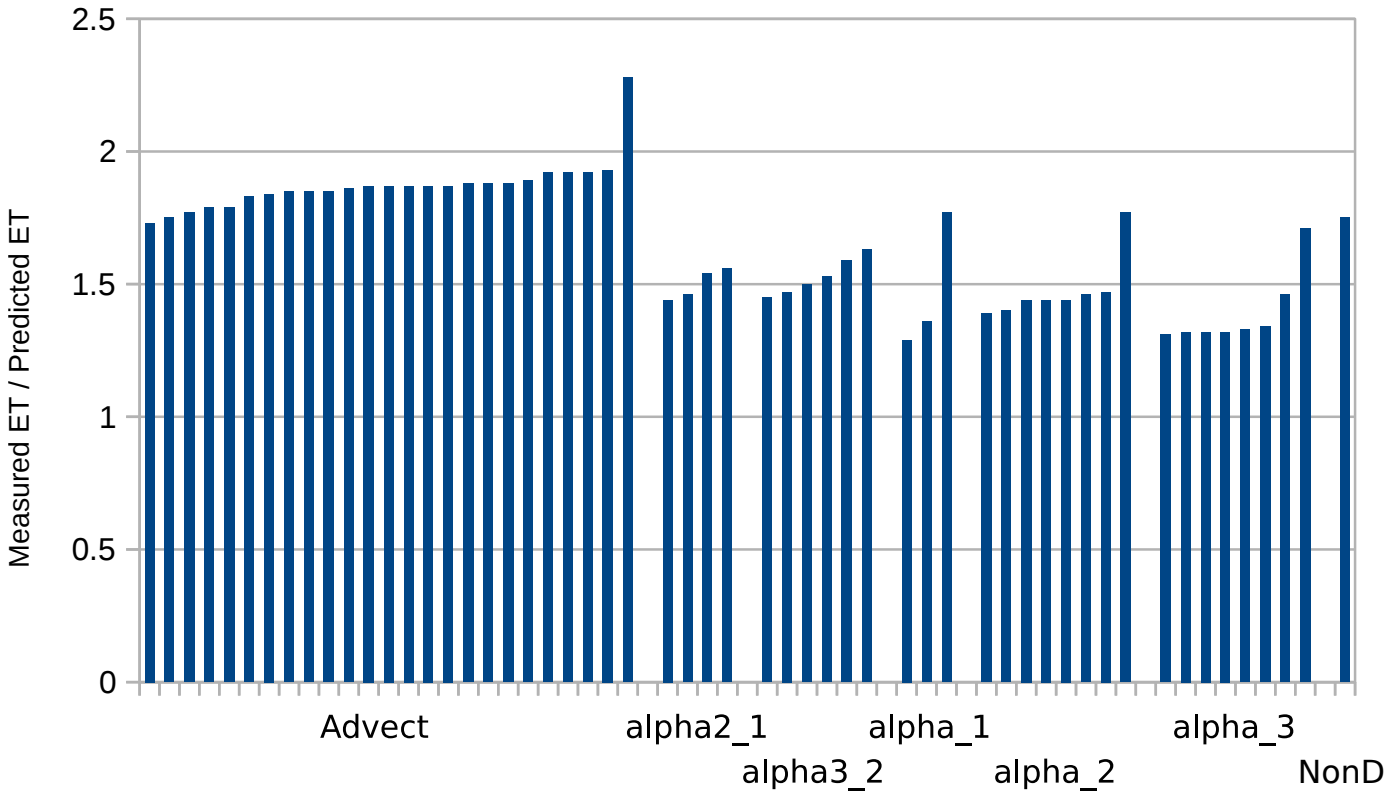
Benefit of: Hand Tuning, Cakernel Dynamic, Chemora



Formatted 13:01, 27 August 2015 from chemora.

et15-45

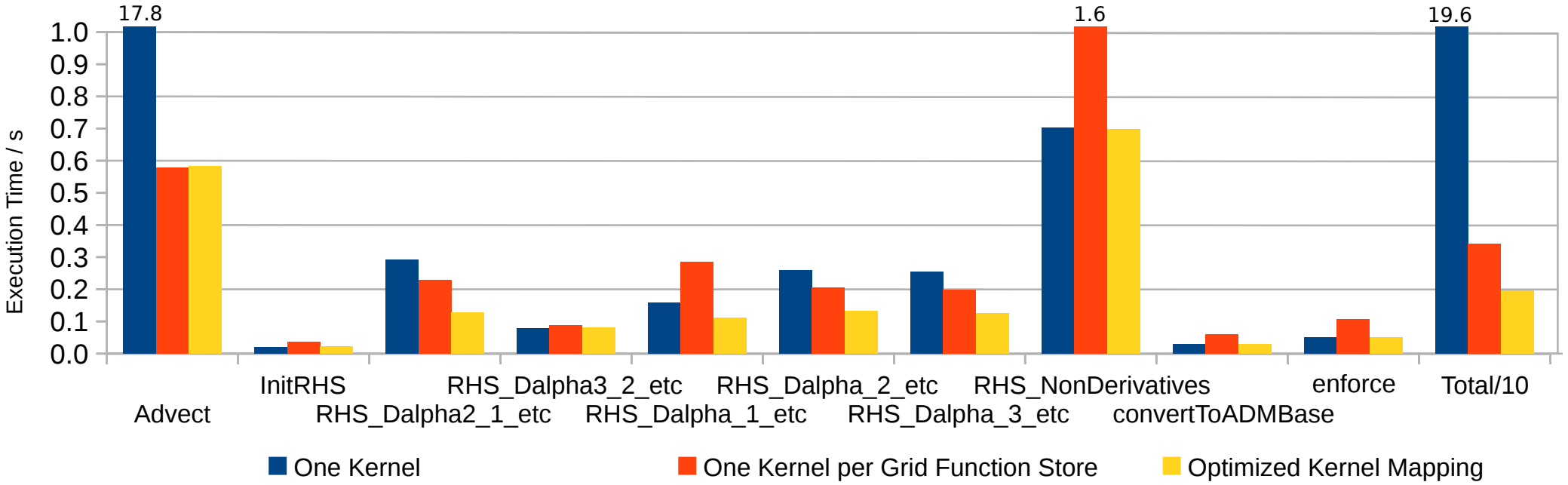
Accuracy of Prediction Model



Within each calculation sorted by accuracy.

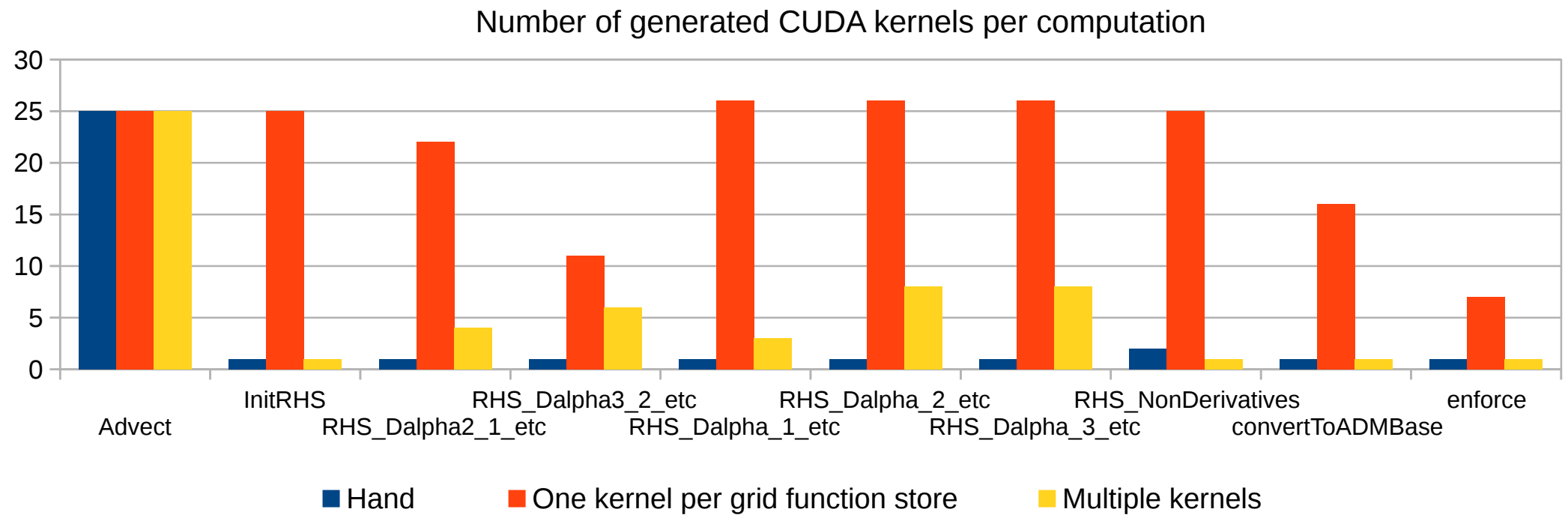
Outliers often due to unanticipated spill/fill code.

Number of Kernels v. Performance



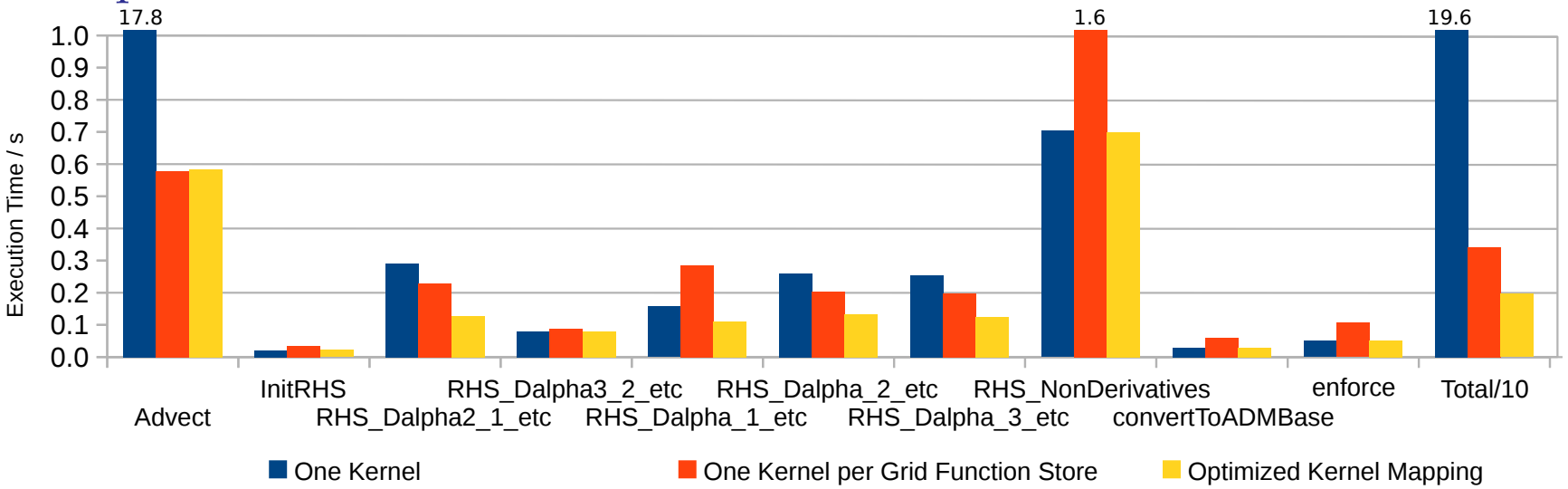
et15-47

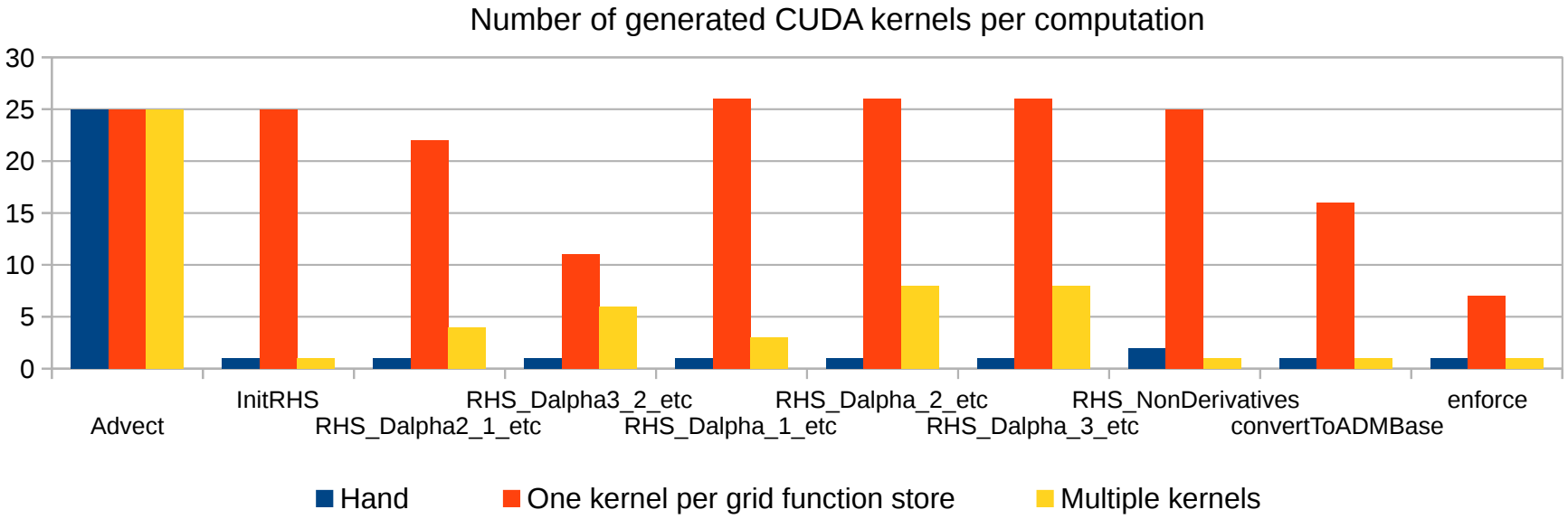
Number of Kernels per Calculation



et15-48

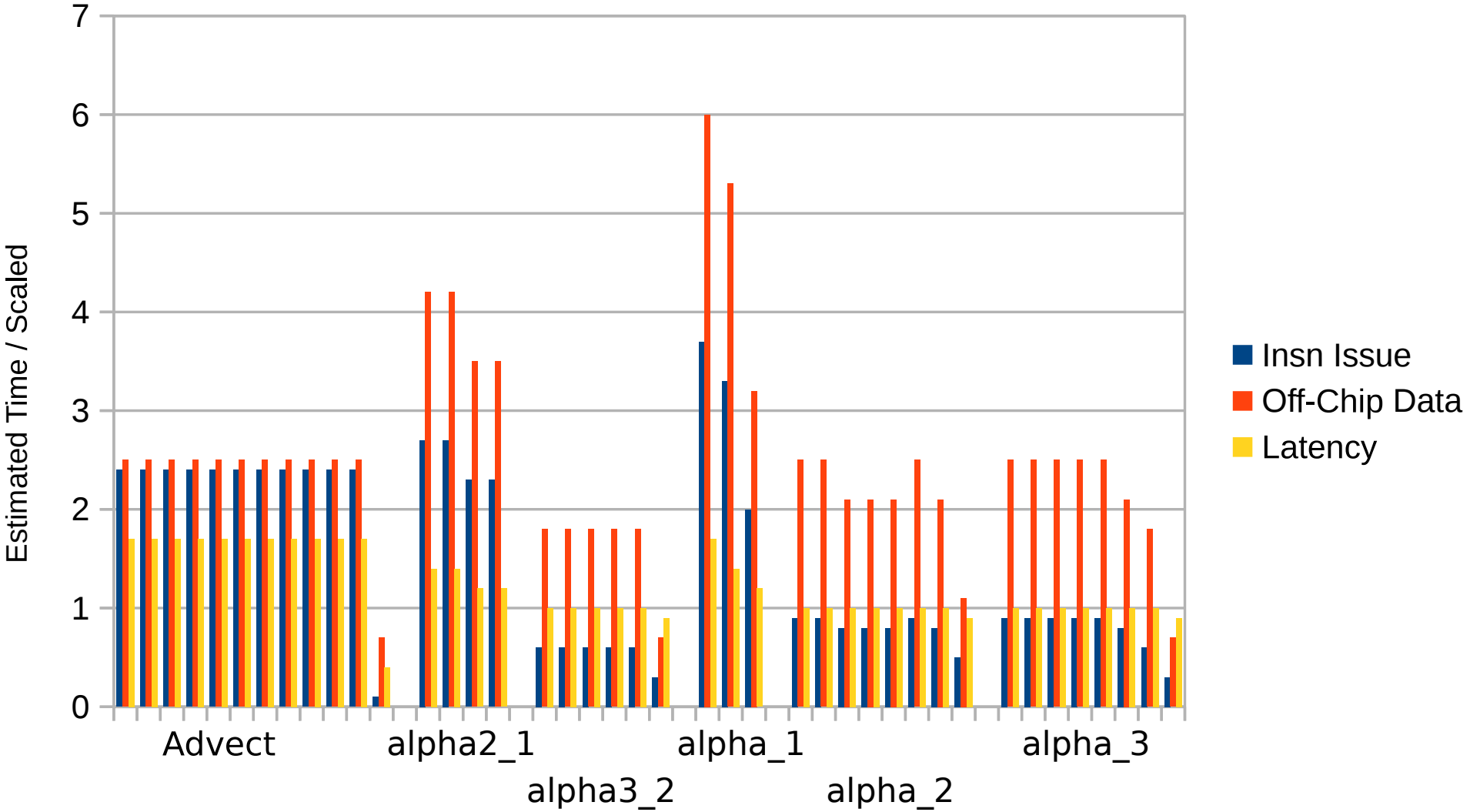
Comparison





et15-49

Execution Time Components



Formatted 13:01, 27 August 2015 from chemora.

et15-50

To Do

Performance Model

Base latency on dependence graph — which we have.

Replace Q&D estimates with true numbers.

Optimization

Rely more on non-stochastic methods.

Code Generation

Reassociate reduction operations to reduce IK storage. (Works on branch).

More optimization of iteration and address computation arithmetic.

Model Scope

Operate on a supercalculation (one that includes dependent differencing operations).

Include CPU/GPU computation.

Thank You

Questions?