From Physics Model to Results: An Optimizing Framework for Cross-Architecture Code Generation

Erik Schnetter, Perimeter Institute with M. Blazewicz, I. Hinder, D. Koppelman, S. Brandt, M. Ciznicki, M. Kierzynka, F. Löffler, J. Tao

XSCALE 2012, Chicago (July 15-16, 2012)
Motivation

- In astrophysics (and elsewhere), physical systems often described via systems of partial differential equations (PDEs)
  - example: general relativity, hydrodynamics, electrodynamics, ...
- Solved via grid-based methods
  - space: FD, FV, FE, ...; time: RK, ...
  - adaptive mesh refinement (AMR), multi-block
- Large HPC systems (MPI, OpenMP, GPUs)

http://einstein toolkit.org
Binary Black Hole Merger

Image: I. Hinder, AEI
A Hard Problem

Physics Model (EDL)

Kranc Code Generator

CaKernel Programming Abstractions

Cactus-Carpet Computational Infrastructure

Parallel Programming Environment

C++/OpenMP

CUDA

OpenCL

OpenACC?

TBB?

HPX?

Results

Publication

Monday, 16 July, 12
EDL: Equation Description Language

Scalar Wave Equation:

begin calculation Init
  \[ u = 0 \]
  \[ \rho = A \exp(-1/2 \ (r/W)**2) \]
  \[ v_i = 0 \]
end calculation

begin calculation RHS
  \[ \partial_t u = \rho \]
  \[ \partial_t \rho = \delta^{ij} \partial_i v_j \]
  \[ \partial_t v_i = \partial_i \rho \]
end calculation

begin calculation Energy
  \[ \varepsilon = 1/2 \ (\rho^{**2} + \delta^{ij} v_i v_j) \]
end calculation

- **Aim:** Write systems of PDEs in easily readable manner
- **High-level, LaTeX-like syntax**
- **Better error reporting than e.g. standard Mathematica input**
- **Describe:** variables, equations, initial/boundary conditions, parameters, analysis quantities
- **Also describe discretisation** (e.g. stencils)
Kranc: Automated Code Generation and Optimisation

- Input in Mathematica (e.g. generated from EDL)
- Generate complete Cactus modules (employing AMR, multi-block, ...)
- Customised FD stencils
- Coordinate transformations for multi-block systems

http://kranccode.org/
Kranc: Automated Code Generation and Optimisation

- Automatically check consistency with other components
- High-level optimisations:
  - (semi-automatic:) loop fission, loop fusion
  - loop blocking, unrolling
- Low-level optimisations:
  - CSE, SIMD, OpenMP
- Generate C++, OpenCL, CUDA

http://kranccode.org/
Cactus Framework, Carpet AMR Driver

- **Cactus**: Software framework for HPC
  - all math/physics/computer science located in modules (components)
  - encourages/supports decentralized code development
- **Carpet**: Cactus driver for AMR, multi-block
  - parallelized via MPI and OpenMP
  - parallel I/O (HDF5)

http://cactuscode.org/
http://carpetcode.org/

Einstein Toolkit benchmark: TOV (9 levels)
Cactus Programming
Abstractions

- **Grid Hierarchy**: list of coordinates, boxes etc. describing domain and discretisation
- *grid functions* live on GH
- storage managed by driver component
- **Schedule**: set of active routines and their dependencies
  - routines implement *kernels* (operators applied to grid functions)
  - have associated meta-data (variables read/written, stencil sizes, iteration region, ...)
- Allows high-level **introspection** (debugging, optimisation)
New in Cactus: Accelerator Framework

- Not all code suitable for GPUs
  - administrative code, legacy code, I/O
- Avoid copying data from/to GPU if possible
  - track which routines require/provide what data where (host or device)
- components already provide metadata for this
- can copy data automatically
Dynamic CPU/GPU Optimisations

• Emphasis: automatic, dynamic, fast optimisation for many different kernels (autotuning)

• Don’t know configurations in advance (AMR, run-time parameters, varying code, different systems) – cannot use static autotuning
Dynamic Tile Selection

- Need to *tile* loops to improve cache performance (GPU: to use fast memory)
  - know stencil sizes, shapes
  - know cache characteristics (line size)
  - array size changes dynamically (AMR!)
- Apply heuristics to choose tile sizes, validated by experiments
Lightweight Kernel Generation

- Instruction cache is small – need to generate short code
- reduce number of array pointers, simplify index arithmetic
- dynamic code generation turns run-time parameters into constants
- can even hard-code loop sizes when re-compiling after AMR regridding
Fat Kernel Detection

• Some kernels are still just too large (e.g. Einstein equations have 7,000 flop)
• (semi-automated) loop fission when generating code
• restructure code to reduce number of live variables
• use heuristics to reduce number of threads
Integrated Performance Monitoring

• Use performance monitoring (PAPI, Nvidia Cupti)
  • collect data after running kernel
  • inform user about performance
  • planned: use this also to e.g. identify fat kernels
Choosing CPU Affinity

Forge (hybrid CPU/GPU system) NCSA
Benchmarks

- Input: High-level description of Einstein equations
- Code generated automatically, run in Cactus framework
- 8th order finite differencing (5 ghost points); ~7,000 flop in kernel
- Einstein equations on GPU, other code on CPU
- ~10% efficiency on GPU (cf. ~20% efficiency on CPU); (these are good numbers!)
Scaling Results

![Graph showing scaling results for different configurations of CPUs and GPUs.](image)

- **CPU_{cane}, 2p(12t)/node**
- **CPU_{datura}, 2p(6t)/node**
- **CPU_{cane}, 4p(6t)/node**
- **GPU_{cane}, no x-split**
- **GPU_{cane}, x-split**

The graph illustrates the time per grid point update per GPU (ns) as a function of the number of GPUs or CPUs. The x-axis represents the number of GPUs or CPUs, while the y-axis shows the time per grid point update per GPU.
Conclusion

• Can auto-generate reasonably efficient code for complex equations from high-level input

• Infrastructure portable to (almost) all HPC architectures

• Need to optimise while generating C++/CUDA code – compilers don’t do that