Code Generators for Stencil Auto-tuning

Shoaib Kamil with Cy Chan, Sam Williams, Kaushik Datta, John Shalf, Katherine Yelick, Jim Demmel, Leonid Oliker
Where this fits in Parlab

- Personal Health
- Image Retrieval
- Hearing, Music
- Speech
- Parallel Browser
- Design Patterns/Motifs
- Composition & Coordination Language (C&CL)
- C&CL Compiler/Interpreter
- Parallel Libraries
- Parallel Frameworks
- Sketching
- Auto-tuners
- Legacy Code
- Schedulers
- Communication & Synch. Primitives
- Efficiency Language Compilers
- Legacy OS
- OS Libraries & Services
- Hypervisor
- Multicore/GPGPU
- RAMP Manycore
- Static Verification
- Type Systems
- Directed Testing
- Dynamic Checking
- Debugging with Replay
- Efficiency Languages
- Diagnosing Power/Performance
- Productivity Layer
- Efficiency Layer
- Diagnosing Power/Performance
- Applications
Conventional Optimization

- Take one kernel/application
  - Perform some analysis
  - Research literature for appropriate optimizations
  - Implement some of them by hand-optimizing for one target machine
  - Iterate

- Result:
  Improve performance for one kernel on one computer
Conventional Auto-tuning

- Automate the code generation and tuning process
  - Perform some analysis of the kernel
  - Research literature for appropriate optimizations
  - Implement a code generator and search mechanism
  - Explore optimization space

- Result:
  - Improve performance for **one** kernel on **many** computers
    - Provides *performance portability*

- Downside:
  - Autotuner creation time is substantial
  - Must reinvent the wheel for every kernel
Motif-specific Frameworks for Auto-tuning

- Programmers express calculation in high-level way
- Kernel represented internally in abstract form
- Auto-tuning system uses code transformation and generation to implement domain-specific transformations

Result:
Significantly improve performance for many kernels in a domain on many computers.
- Obtain performance portability without sacrificing productivity
Outline

1. Stencils
2. Framework
3. Performance/Productivity Results
4. Stencils in High Level Dynamic Languages
5. Conclusions
What’s a stencil?

- Nearest neighbor computations on structured grids (1D...ND array)

- Weights can be constant or vary depending on space, time, or data

- Used in applications such as PDE solvers, astrophysics, climate simulation, image filtering

- Auto-tuning target: kernels with *separate read and write arrays*
Studied Kernels

- **read_array[ ]**: u
- **x dimension**: y

**xy product**

- **write_array[ ]**: u'
- **Divergence**

**Laplacian**

- **read_array[ ][ ]**: x
- **x dimension**: y
- **xy product**: z

- **write_array[ ]**: u
Studied Kernels

**Gradient**
- `read_array[]`
- `write_array[][]`
- `x dimension`
- `xy product`

**Bilateral Filter**
- `read_array[]`
- `filter_array[]`
- `write_array[]`
- `lookup`
Studied Kernels
Studied Kernels
Outline

1. Stencils
2. Framework
3. Performance/Productivity Results
4. Stencils in High Level Dynamic Languages
5. Conclusions
Auto-tuner Overview

Perl Script
Auto-tuner Overview

Perl Script

- Code Generators
  - FORTRAN
  - C with pthreads
  - CUDA

- Myriad of equivalent, optimized, implementations (plus test harness)

- Search Engines
  - in context of specific problem

- Best performing implementation and configuration parameters

- Optimized Library
Auto-tuner Overview

Tuner

Reference Implementation

Code Generators

.f95
FORTRAN
C with pthreads
CUDA

Myriad of equivalent, optimized, implementations (plus test harness)

Search Engines

.best performing implementation and configuration parameters

Optimized Library

15
Auto-tuner Overview

- Input is a high-level description/implementation of the kernel
- Framework parses into an internal representation
- Strategy Engines + Backend Code Generators optimize & generate candidate implementations
- End result: an optimized library containing best implementation
Many-/Multi-core Strategy Engine

- Multicore strategy engine divides computation into cache blocks and distributes blocks over cores

- We use a single-program, multiple-data (SPMD) model implemented with POSIX Threads (Pthreads)

- All threads created at the beginning of the application

- Tuner produces initialization routine that exploits first-touch policy to ensure proper NUMA-aware allocation
Many-/Multi-core Strategy Engine

- Strategy Engine explores a number of auto-tuning optimizations:
  - loop unrolling/register blocking
  - cache blocking
  - constant propagation / common subexpression elimination

Future Work:
- cache bypass (e.g. `movntpd`)
- software prefetching
- SIMD intrinsics
- data structure transformations
CUDA Strategy Engine

- Strategy Engine parallelizes stencils using CUDA
- Exploit spatial locality by ensuring adjacent CUDA threads operate on adjacent memory locations
- Memory coalescing

Auto-tuning
- Explore shape of CUDA thread block
- Like register blocking optimization in Multi-core

Future Work:
- Exploit temporal locality
  - Properly use memory in all levels of the hierarchy
Outline

1. Stencils
2. Framework
3. Performance/Productivity Results
4. Stencils in High Level Dynamic Languages
5. Conclusions
Results Key

- **serial reference**
  - Original code in Fortran

- **Auto-parallelization**
  - **Auto-parallelized** using the stencil framework (no tuning)

- **Auto-NUMA**
  - Auto-parallelized **plus NUMA optimization**

- **Auto-tuning**
  - **Auto-tuned** and auto-parallelized using the stencil framework

- **STREAM Predicted**
  - Memory-bound **performance predicted using OpenMP STREAM benchmark**

- **OpenMP Comparison**
  - Performance of a NUMA-aware auto-parallelized with **OpenMP version of the original code**
Laplacian Results

- Auto-parallelization by itself does not scale well on CPUs
  - requires NUMA-aware alloc to get decent performance
- our auto-parallelizer gets equal or better performance than OpenMP

- Overall speedups of up to 22x on Nehalem (vs. serial reference), 1.5x on GTX280
Divergence Results

- Less benefit from auto-tuning on cache-based architectures here
  - As we expect based on arithmetic intensity
- Overall speedups of up to 13x on Victoria Falls, 2x on GTX280
Gradient Results

- Heavily memory-bound, so architectures with high memory BW get higher performance
- Overall speedups of up to 8.1x on Nehalem, 1.7x on GTX280
Bilateral Filter Results (r=3)

- Heavily compute-bound, plus lookup for filter weights
  - Most of auto-tuning benefit comes from better innermost-loop

- Overall speedups of 14.8x for Barcelona, 20.7x for Nehalem

- Near linear speedup as cores increase
Outline

1. Stencils
2. Framework
3. Performance/Productivity Results
4. Stencils in High Level Dynamic Languages
5. Conclusions
High Level Languages

Common complaint from domain scientists: too much overhead in experimenting with kernels

- Must manage memory, array layouts, etc

Languages like Ruby & Python support high-level programming with frameworks & libraries

What would productive *parallel* stencil support look like in Ruby?

- Must deal with lack of thread-safety in interpreter
One Approach

- Solution: write stencil in Ruby
  - Use conventions to simplify code structure

Then, **transparently**:

- Use Ruby’s introspective nature to parse code
- Dynamically translate to C, compile, link, and execute translated code on Ruby data structure

- Only translate the stencil kernel: the rest is still in pure Ruby
Example

```ruby
class LaplacianKernel < JacobiKernel
def kernel(in_grid, out_grid)
in_grid.each_interior do |center|
in_grid.neighbors(center,1).each do |x|
  out_grid[center] = out_grid[center] + 0.2 * in_grid[x]
end
end
end
end
```

When the Ruby program calls `kernel()` this is automatically generated, compiled, and run.
Results

- Comparable performance to OpenMP+C
- First execution takes more time (JITing)
  - Subsequent executions are fast

Example: Laplacian on Nehalem (25 iterations)

- Ruby performance is between C+OpenMP and C+OpenMP+NUMA
- Ruby version is not NUMA-aware
- Multicore stencil support in Ruby is >500x faster than a pure Ruby implementation
Summary

- Summer 2008 Retreat: feedback that auto-tuners are not very auto
- Winter 2008 Retreat: Presented idea of auto-tuners for a class of kernels
  - Serial results for 1 kernel
- Now: Parallel stencil auto-tuning for many kernels on many architectures
- Obtain performance and platform portability
- High level dynamic languages can use same techniques to produce portable efficient code
- Lots of future work: better CUDA/OpenCL support, widen class of supported stencils
Acknowledgements

- Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227) as well as ASCR Office in the DOE Office of Science under contract #AC02-05CH11231.

- Wes Bethel and Visualization Group at LBNL for serial Bilateral Filter code and source data.