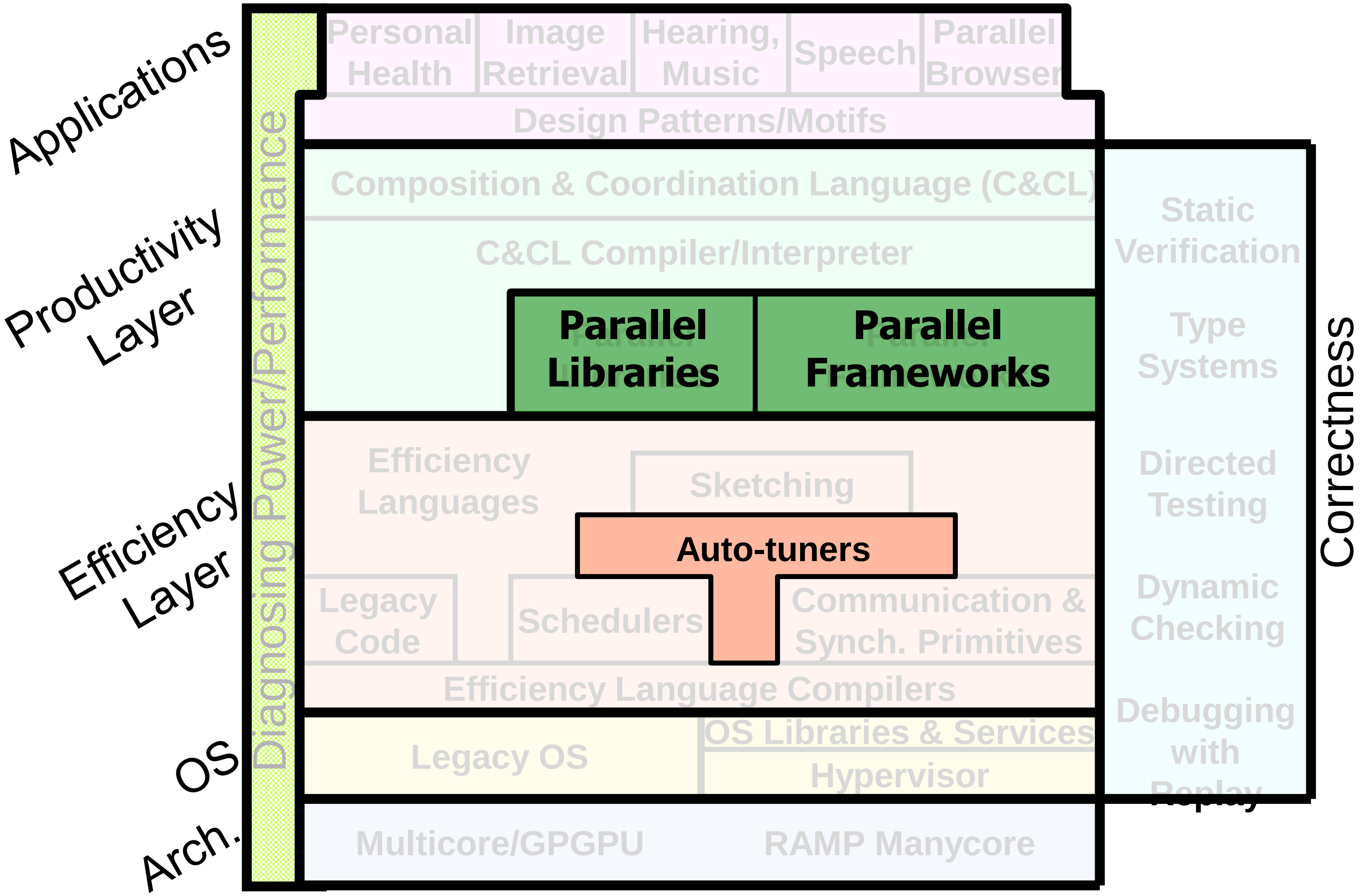
Code Generators for Stencil Auto-tuning

Shoaib Kamil with Cy Chan, John Shalf,
Sam Williams, Kaushik Datta, Katherine Yelick,
Jim Demmel, Leonid Oliker

Where this fits in Parlab



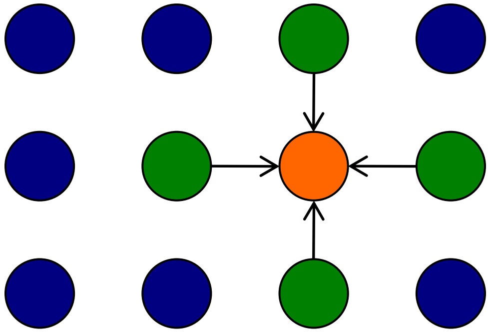
Making Auto-tuners Really Auto

- Search
  - Old: Extremely user guided, or arbitrary orderings of optimizations
  - New: more intelligence (as in talk by K. Datta & A. Ganapathi)

- Code generation
  - Old: Perl scripts (one per kernel) that are essentially "glorified printfs" aka string substitution
  - New: represent kernels abstractly (this talk), extend auto-tuning to motifs that are not well-represented as libraries

What is a Stencil Kernel?

- Many computations on grids with regular structures can be represented as “sweeps” over the grid, where each point in a sweep is a arithmetic combination of the point’s neighbors
- “Kernel” is an instance of a stencil operator
- Want to make Auto-tuners for many (all?) motifs; start with stencils
  - Varied enough but still relatively simple



Stencil Example: 5 point stencil in 2D. Orange point is updated with a combination of the green points.

Goals

- How can we take previous work using a single stencil kernel and build an auto-tuning system like Atlas/PhiPAC/OSKI
  - Relatively user-friendly
  - Automated
  - Works across many stencil kernels
  - Target GPUs, Manycore, SMP
- Too much variability to write a single library that contains all possible kernels (a la OSKI)
- What can we learn about productively writing auto-tuners
- Proof-of-Concept

Overview of Stencil Auto-tuner

- Front end parses stencil kernel using Domain-Specific Language
- Transformation framework applies domain-specific optimizations
- Code generator outputs candidate versions
- Controlled by Search system ("outer loop")
- We do the dirty work of optimizing: Performance Layer programmer can just run the Stencil Auto-tuner to get **performance portability**

DSL's for Stencils?

- Only need a few features:
  - Represent the operation at each point
    - E.g. "add left and right neighbors, multiply by 4"
  - Represent operations at boundaries

- Why not use existing code, say Fortran or Matlab + annotations?

```
do i=2,99,1
 do j=2,99,1
           wr_arry(i,j) = rd_arry(i,j) + rd_arry(i+1,j) -
                                         rd_arry(i,j+1)
 enddo
enddo
```

Domain-Specific Transformations

- Express optimizations as transformations to an abstract representation of the stencil
  - E.g. “Cache Blocking” is a combination of two loop transformations
- Advantage: can arbitrarily mix/combine optimizations
- Domain-specificity means minimal analysis needed
  - Only do transformations we’re sure are correct
  - Example: if we know stencil “footprint” can tell how/whether to cache block

Transformations

- Proof-of-Concept represents stencil kernel as simplified AST
- AST -> AST transformations
- Alternative methods
  - Sketching (some investigation of this already)
  - Low-Level Virtual Machine (LLVM)
  - String substitution/rewriting
    - Probably not powerful enough
  - Others?

Code Generation

- Current backends do AST->code
  - Fortran (serial)
  - C (serial, several different array representations)
  - C + pthreads
  - CUDA
- Parallel backends represent parallelization as just another transformation
- In progress/Consideration
  - C + Cell-specific pthreads
  - OpenCL in progress
  - UPC? Fortress or X10?

Search

- Search system goals
  - Should automatically extract annotated kernels
  - Then run candidate implementations
  - Gather "winning" implementations (one per kernel) into a library that can be called by the application

- Currently, the "outer loop" runs candidate implementations and uses fastest
- Semi-exhaustive search (e.g. powers of 2 cache blocking)
- More intelligent search needed?
  - Hill climbing? (a la CG)
  - Machine learning? (see talk by K. Datta & A. Ganapathi)

Status of Proof-of-Concept

- Few months of effort
- Implemented most transformations from K. Datta & S. Williams SC08 paper
- Many lessons learned
  - Writing code is important!
  - Testing with real-world kernels & applications
    - Use not only microbenchmarks like Stencil Probe
    - Green Flash's climate application has many stencils-> many tests
  - Implementation in Lisp ("the L-word!")
    - Higher order functions make composition easier
    - Simple tree representation as lists
    - Mostly-functional programming
    - Don't worry: Lisp -> C then linked with C libraries

Auto-tuning Example

- Operators extracted from climate code before and after auto-tuning

```
 do k=0,km,1
  do iprime=1,nside,1
    do i=2,im2nghost-1,1
    ia  = i + ii(iprime)
      do j=2,jm2nghost-1,1
         ja  = j + jj(iprime)
        buoyancy_gen(i,j,iprime,k)
           =-1.0*g*(theta(ia,ja,k) -
           theta(i,j,k))
          /(theta00(k)*el(iprime))
      enddo
    enddo
  enddo
enddo
```

```
do G14906=0,km,4
do G14907=1,nside,6
do G14908=2,im2nghost - 1,25
do G14909=2,jm2nghost - 1,25
do k=G14906,G14906 + 1,1
do iprime=G14907,G14907 + 5,1
do i=G14908,G14908 + 24,1
ia = i + ii(iprime)
do j=G14909,G14909 + 24,1
ja = j + jj(iprime)
buoyancy_gen(i,j,iprime,k) = -1.0 * g * theta(ia,ja,k) - theta(i,j,k) /
                  theta00(k) * el(iprime)
enddo
enddo
enddo
enddo
enddo
enddo
enddo
do G14907=1,nside,6
do G14908=2,im2nghost - 1,25
do G14909=2,jm2nghost - 1,25
do k=G14906 + 2,G14906 + 3,1
do iprime=G14907,G14907 + 5,1
do i=G14908,G14908 + 24,1
ia = i + ii(iprime)
do j=G14909,G14909 + 24,1
ja = j + jj(iprime)
buoyancy_gen(i,j,iprime,k) = -1.0 * g * theta(ia,ja,k) - theta(i,j,k) /
                  theta00(k) * el(iprime)
enddo
enddo
enddo
enddo
enddo
enddo
enddo
enddo
```

Examples

- Heat

  Aout(i,j,k) = A(i,j,k) + factor*(A(i+1,j,k) + A(i-1,j,k) + A(i,j+1,k) + A(i,j-1,k) + A(i,j,k+1) + A(i,j,k-1))

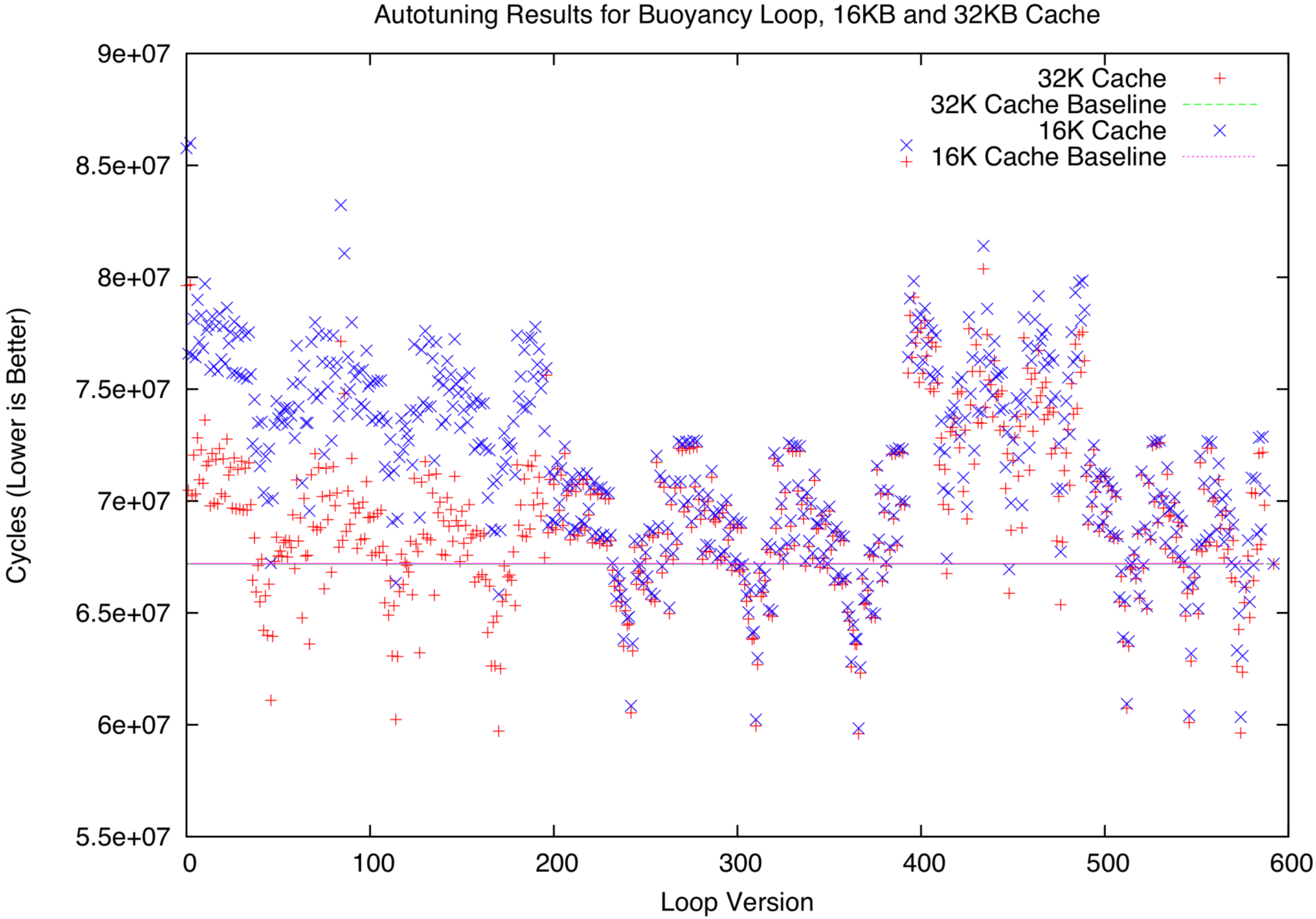
- Divergence (from application)

```
xout(i,j) =                                                   &
        (   v_weights(1,2,1,i  ,j  ) * xin(1,1,i  ,j  )       &
          - v_weights(1,1,1,i  ,j  ) * xin(2,1,i  ,j  )       &
          + v_weights(1,2,2,i  ,j  ) * xin(1,2,i  ,j  )       &
          - v_weights(1,1,2,i  ,j  ) * xin(2,2,i  ,j  )       &
          + v_weights(2,2,1,i+1,j  ) * xin(1,1,i+1,j  )       &
          - v_weights(2,1,1,i+1,j  ) * xin(2,1,i+1,j  )       &
          + v_weights(2,2,2,i+1,j+1) * xin(1,2,i+1,j+1)       &
          - v_weights(2,1,2,i+1,j+1) * xin(2,2,i+1,j+1)       &
          + v_weights(3,2,1,i+1,j+1) * xin(1,1,i+1,j+1)       &
          - v_weights(3,1,1,i+1,j+1) * xin(2,1,i+1,j+1)       &
          + v_weights(3,2,2,i  ,j+1) * xin(1,2,i  ,j+1)       &
          - v_weights(3,1,2,i  ,j+1) * xin(2,2,i  ,j+1) )     &
          * area_inv(i,j)
```

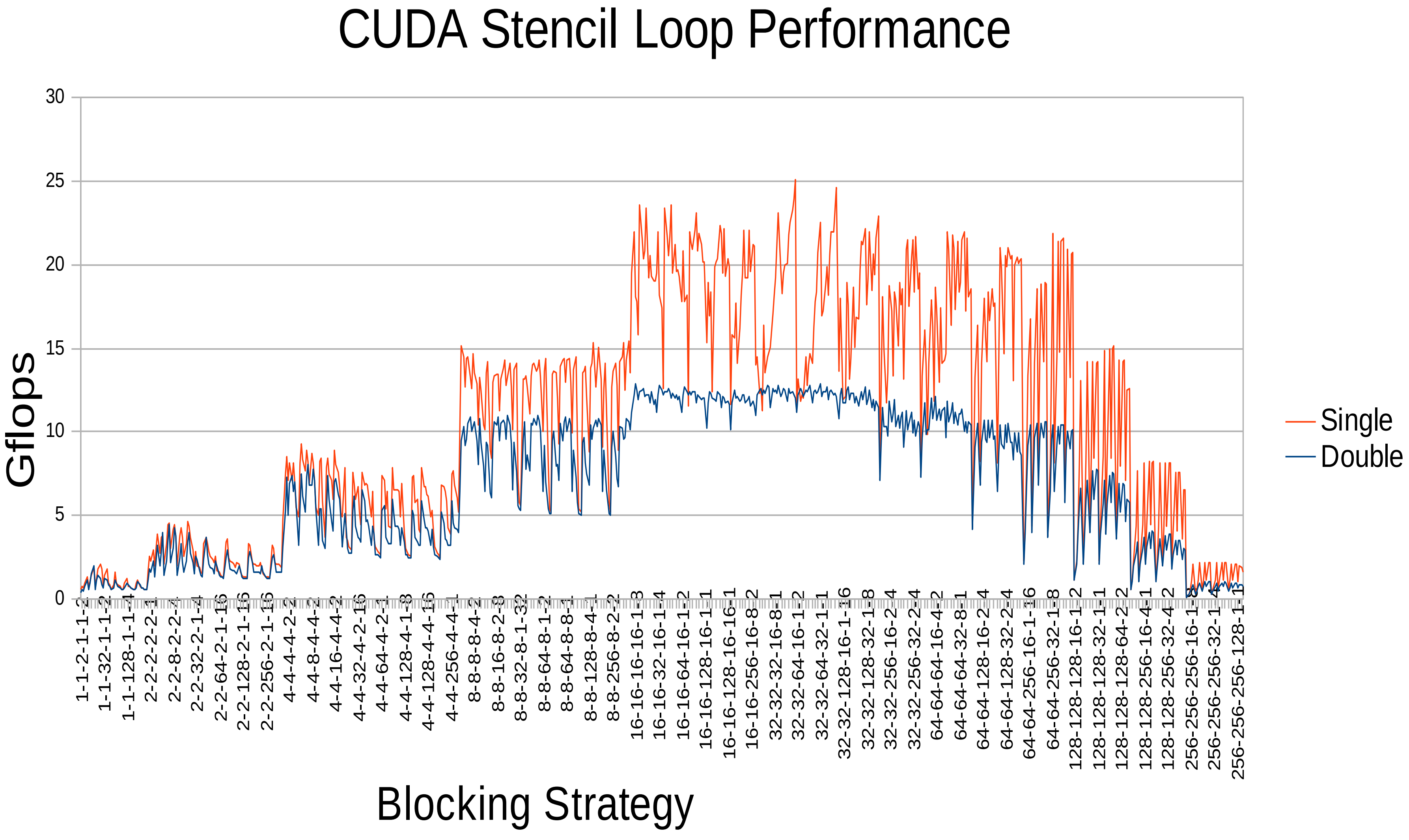
Examples

- Stencil Auto-tuner generates optimized code for these kernels on
  - Nvidia GTX280
  - Pthreads + C on Intel, AMD, and Sun Victoria Falls with 2 different indexing strategies
  - Serial Fortran

Performance Example: Serial

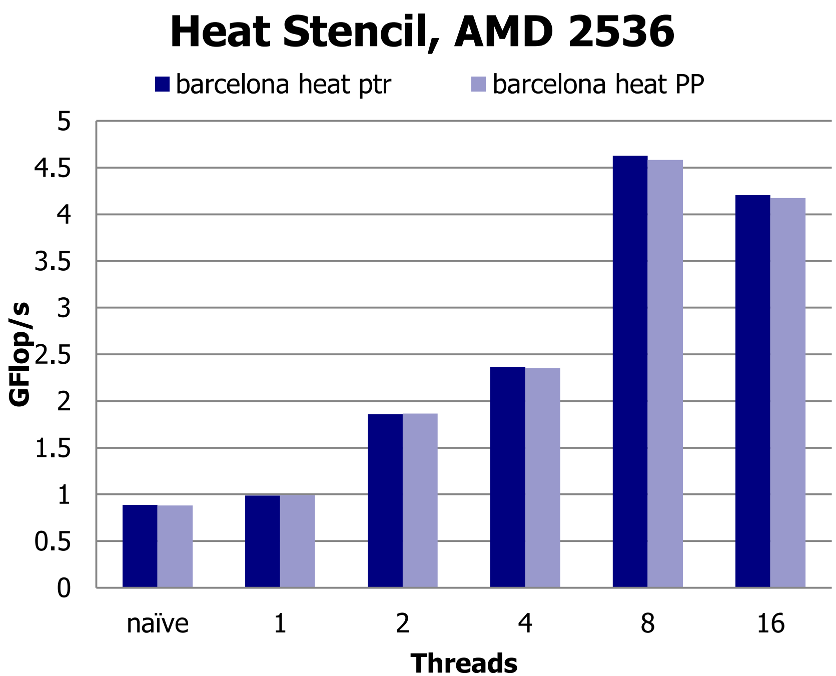


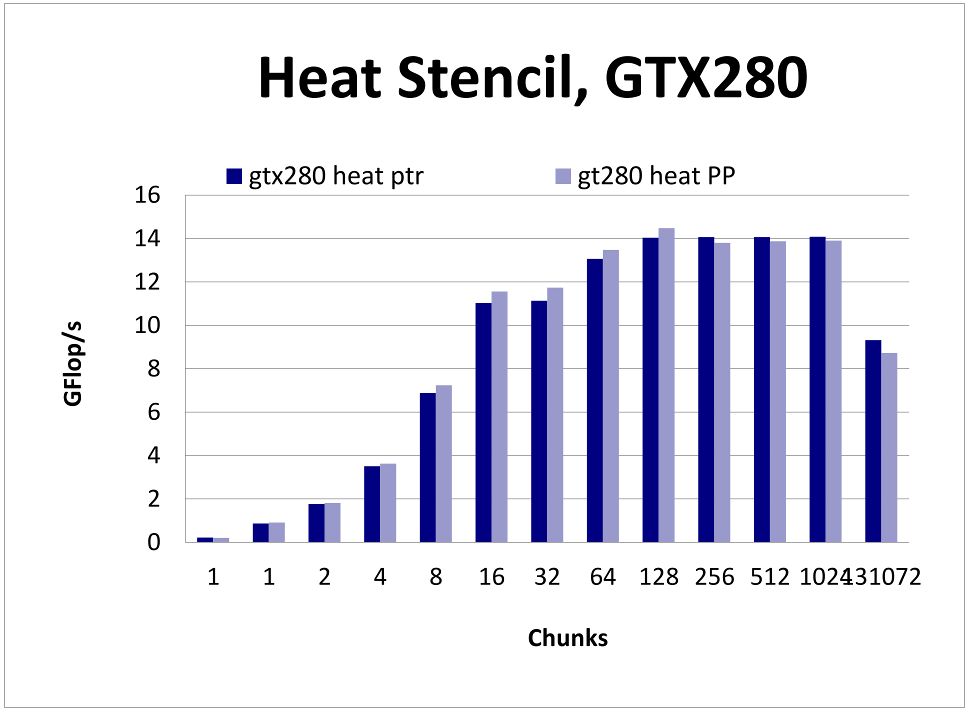
Performance Example: CUDA



Parallel Performance Results

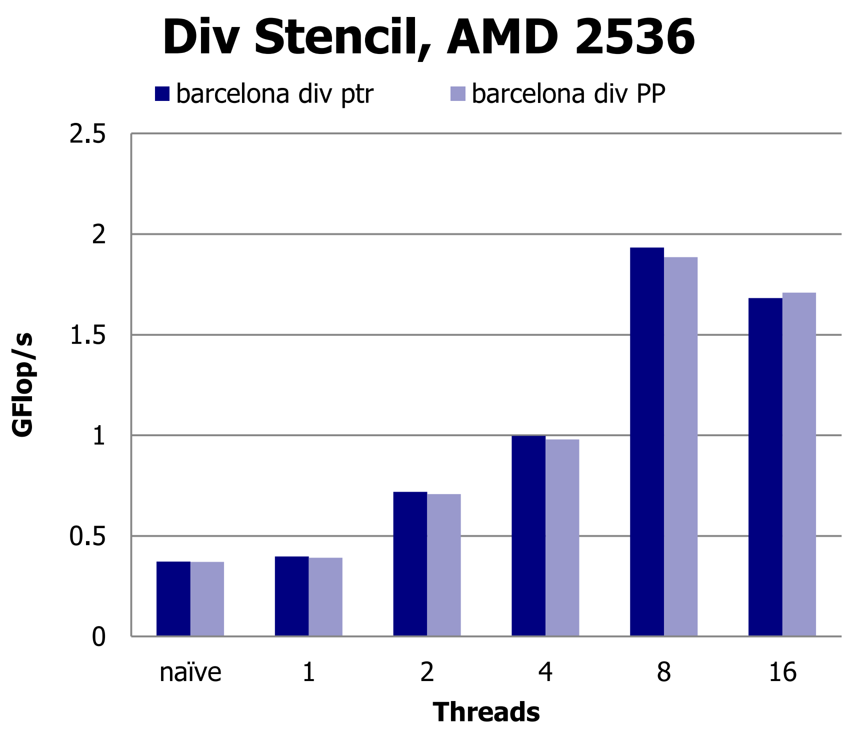
- Heat Equation: 7pt 3D stencil from Stencil Probe

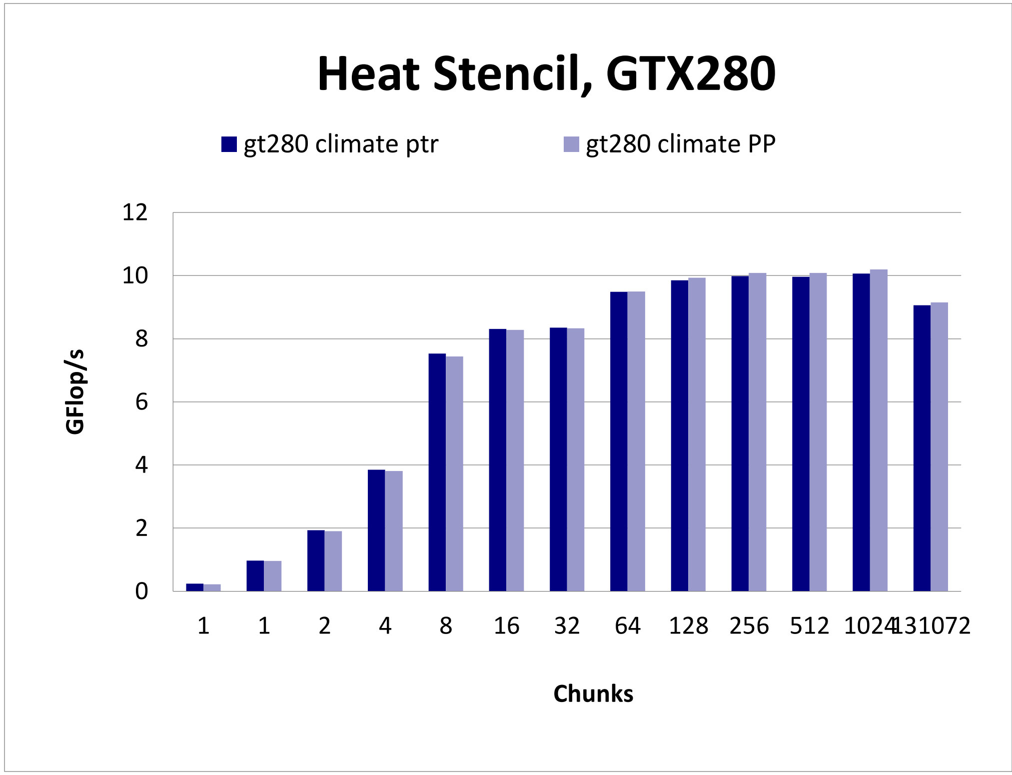




Parallel Performance Results

- 2D Divergence Kernel from Climate Application





Summary & Conclusions

- Auto-tuners should really be automatic
- Presented a framework & proof-of-concept that implements Stencil Auto-tuner
- Showed we can speed up real-world stencil kernels
- Writing Auto-tuners
  - Use better methods than string substitution
  - Make more general
  - Better search
- Performance portability for large class of stencil kernels