Leveraging Compiler-Based Tools for Performance Portability

Protonu Basu+, Samuel Williams+, Brian Van Straalen+, Leonid Oliker+, Philip Colella+, Mary Hall*

+ Lawrence Berkeley National Laboratory
* University of Utah
Performance and Productivity Challenge –
GSRB Smooth

/* Laplacian 7-point Variable-Coefficient Stencil */
for (k=0; k<N; k++)
  for (j=0; j<N; j++)
    for (i=0; i<N; i++)
      temp[k][j][i] = b * h2inv * (beta_i[k][j][i+1] * (phi[k][j][i+1] – phi[k][j][i])
                     -beta_i[k][j][i] * (phi[k][j][i] – phi[k][j][i-1])
                     +beta_j[k][j+1][i] * (phi[k][j+1][i] – phi[k][j][i])
                     -beta_j[k][j][i] * (phi[k][j][i] – phi[k][j-1][i])
                     +beta_k[k+1][j][i] * (phi[k+1][j][i] – phi[k][j][i])
                     -beta_k[k][j][i] * (phi[k][j][i] – phi[k-1][j][i]));

/* Helmholtz */
for (k=0; k<N; k++)
  for (j=0; j<N; j++)
    for (i=0; i<N; i++)
      temp[k][j][i] = a * alpha[k][j][i] * phi[k][j][i] – temp[k][j][i];

/* Gauss-Seidel Red Black Update */
for (k=0; k<N; k++)
  for (j=0; j<N; j++)
    for (i=0; i<N; i++)
      if (((i+j+k+color)%2 == 0)
           phi[k][j][i] = phi[k][j][i] – lambda[k][j][i] *
                  (temp[k][j][i] – rhs[k][j][i]));

Code A: miniGMG baseline smooth operator approximately 13 lines of code

Code B: miniGMG optimized smooth operator approximately 170 lines of code
**GPU code for GSRB Smooth**

**Code C:** miniGMG optimized smooth operator for GPU, 308 lines of code for just kernel
Background: Challenges

• Performance portability
  Across fundamentally different CPU and GPU architectures

• Programmer productivity
  High performance implementations will require low-level specification in
  standard MPI+OpenMP, CUDA

• Software maintainability and portability
  May require maintaining multiple implementation of same computation
Possible ways to address the challenges

- Follow MPI and OpenMP standards
  Same code unlikely to perform well across CPU and GPU
  Low level specification may be required for high-performing OpenMP
  Vendor C and Fortran compilers not optimized for HPC workloads

- Some domain-specific framework strategies
  Libraries, C++ template expansion, standalone DSL
  Not *composable* with other optimizations
Compiler Based Approach

- Exploit existing compiler transformations to accomplish optimization goals
- Develop new domain-specific transformations and required analysis and code generation support
- Supports autotuning

CHiLL is a source-to-source compiler framework with a script interface

Novel Domain-Specific optimization implemented in CHiLL
Compiler Based Approach

- Composable transformation and code generation
  
  Leverage rich set of existing transformations and code generation capability
  
  Mathematically represented using polyhedral framework

- Extensible to new domain-specific transformations and decision algorithms

  Compose with existing transformations
## Experience with CHiLL

<table>
<thead>
<tr>
<th>Experience with CHiLL</th>
<th>Input</th>
<th>Existing Transformations</th>
<th>Domain-specific transformations</th>
<th>Autotuning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometric Multigrid</td>
<td>Sequential C computation (w/ MPI and OpenMP harness)</td>
<td>Communication-avoiding: fusion, tile, wavefront (skew&amp;permute), OpenMP, CUDA</td>
<td>Ghost zones, Partial sums</td>
<td>Ghost zone depth, threading, strategy at each level of V-cycle</td>
</tr>
<tr>
<td>Tensor Contraction</td>
<td>Mathematical Formula</td>
<td>Tile, permute, scalar replacement, unroll, CUDA</td>
<td>Rewriting, Decision algorithm</td>
<td>Loop order, CUDA threading</td>
</tr>
<tr>
<td>Sparse Matrix Computation</td>
<td>Sequential C with CSR matrix</td>
<td>Tile, permute, skew, unroll, reduction, scalar expansion, OpenMP, CUDA</td>
<td>Generate inspectors, coalesce, make-dense, compact, split, level sets</td>
<td>Threading, matrix repr.</td>
</tr>
</tbody>
</table>
Performance Bottlenecks

<table>
<thead>
<tr>
<th>Stenc</th>
<th>Coefficient</th>
<th>Iteration</th>
<th>Flop</th>
<th>Byte</th>
<th>AI</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-point</td>
<td>Constant</td>
<td>Jacobi</td>
<td>8</td>
<td>24</td>
<td>0.33</td>
</tr>
</tbody>
</table>

Performance Limited by Memory Bandwidth!
<table>
<thead>
<tr>
<th>Stencil</th>
<th>Coefficient</th>
<th>Iteration</th>
<th>Flop</th>
<th>Byte</th>
<th>AI</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-point</td>
<td>Variable</td>
<td>GSRB</td>
<td>17</td>
<td>80</td>
<td>0.21</td>
</tr>
<tr>
<td>7-point</td>
<td>Constant</td>
<td>Jacobi</td>
<td>17</td>
<td>48</td>
<td>0.35</td>
</tr>
<tr>
<td>13-point</td>
<td>Constant</td>
<td>Jacobi</td>
<td>8</td>
<td>24</td>
<td>0.33</td>
</tr>
<tr>
<td>27-point</td>
<td>Constant</td>
<td>Jacobi</td>
<td>15</td>
<td>24</td>
<td>0.63</td>
</tr>
<tr>
<td>125-point</td>
<td>Constant</td>
<td>Jacobi</td>
<td>32</td>
<td>24</td>
<td>1.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>134</td>
<td>24</td>
<td>5.58</td>
</tr>
</tbody>
</table>
Geometric Multigrid (GMG)

MG is a hierarchical approach to solving the linear system $Ax=B$

GMG solves the linear system $Ax=B$, where $A$ is a stencil applied on a grid

- Smooth $L u^h = f^h$
- $r^h = f^h - L u^h$ (residual)
- $f^{2h} = \text{restrict}(r^h)$
- Smooth $L u^{2h} = f^{2h}$
- $r^{2h} = f^{2h} - L u^{2h}$
- $f^{4h} = \text{restrict}(r^{2h})$
- Smooth $L u^{4h} = f^{4h}$
- $r^{4h} = f^{4h} - L u^{4h}$
- $f^{8h} = \text{restrict}(r^{4h})$
- Smooth $L u^{8h} = f^{8h}$
- $u^h += \text{interpolate}(u^{2h})$
- $u^{2h} += \text{interpolate}(u^{4h})$
- $u^{4h} += \text{interpolate}(u^{8h})$

Multiple smooth's on $L u^{8h} = f^{8h}$ (or Iterative Solver like BiCGStab)

Progress within V-cycle

GMG V-cycle

Bottom Solve

Smooth

Residual

Restriction

Interpolation

Smooth
miniGMG

Domain $256^3$

List of $64^3$ Boxes Computed In Parallel (OMP)

Domain decomposed to MPI processes (2)

Smooth Dominates Runtime

48 iterations of Smooth

GMG V-cycle

4 iterations of smooth

4
8
16
32
64
Memory Bandwidth Bound

Stencil/Smooth

Compiler Autotuning Matches Manual Tuning!

Compute Bound by High FLOP intensity and Poor Register Reuse

Compiler Autotuning Beats Roofline*!

GSRB Smooth on 64, 64³ boxes

CUDA-CHiLL

Compiler Autotuning Beats Roofline*!
for (k=0; k<N; k++)
  for (j=0; j<N; j++)
    for (i=0; i<N; i++)
      /* statement S0 */
      // temp[k][j][i] = b * h2inv * (beta_i[k][j][i+1] * (phi[k][j][i+1] – phi[k][j][i])
      // - beta_i[k][j][i] * (phi[k][j][i] – phi[k][j][i-1])
      // + beta_j[k][j+1][i] * (phi[k][j+1][i] – phi[k][j][i])
      // - beta_j[k][j][i] * (phi[k][j][i] – phi[k-1][j][i])
      // + beta_k[k+1][j][i] * (phi[k+1][j][i] – phi[k][j][i])
      // - beta_k[k][j][i] * (phi[k][j][i] – phi[k-1][j][i]));

for (k=0; k<N; k++)
  for (j=0; j<N; j++)
    for (i=0; i<N; i++)
      /* statement S1 */
      // temp[k][j][i] = a * alpha[k][j][i] * phi[k][j][i] – temp[k][j][i];

for (k=0; k<N; k++)
  for (j=0; j<N; j++)
    for (i=0; i<N; i++)
      if (((i+j+k+color)%2 == 0))
        /* statement S2 */
        // phi[k][j][i] = phi[k][j][i] – lambda[k][j][i] *(temp[k][j][i] – rhs[k][j][i]);
Wavefront: Reducing Vertical Communication

Wavefront fuses multiple grid sweeps reducing DRAM traffic
Wavefront: Reducing Vertical Communication

Wavefront = Loop Skew + Loop Permute

We tune to find the ghost zone depth and wavefront depth!
OpenMP Code Generation: Nested Parallelism

Wavefront has a larger working set

Thread blocking to manage working set

Thread 0, Thread 1, Thread 2, Thread 3
For each level of the V-cycle, our search space is:

- Ghost zone depth ‘d’ \{1,2,4,5\}
- Fused Code or Wavefront variant
- Thread decomposition <x, y>, where x*y = #cores/chip
Performance of GSRB Smooth

Manual tuning spent considerable effort on finer $64^3$ boxes but did not specialize for smaller boxes.

Autotuning picked nested-parallelism for finer boxes; manually tuned code used intra-box threading.

<table>
<thead>
<tr>
<th>Box</th>
<th>Edi/Hop</th>
<th>Thread Decomposition</th>
<th>Code Variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>4</td>
<td>&lt;4,3&gt;</td>
<td>Edi/Hop</td>
</tr>
<tr>
<td>32</td>
<td>4</td>
<td>&lt;4,3&gt;</td>
<td>Wave</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>&lt;12,1&gt;</td>
<td>Wave</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>&lt;12,1&gt;</td>
<td>Wave</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>&lt;12,1&gt;</td>
<td>Fused</td>
</tr>
</tbody>
</table>

Box Size

Speedup Over Baseline
CUDA-CHiLL is a thin layer built on top of CHiLL to generate CUDA code. It deconstructs (tiles) a loop nest, and assigns loops to threads and blocks.
Parallelization via Loop Tiling

Input GSRB smooth
for(box = 0; box <= 63; box++) {
    for(k = 1; k <= 64; k++) {
        for(j = 1; j <= 64; j++) {
            for(i = 0; i <= 15; i++) {
                for(ii = 0; ii <= 1; ii++) {
                    for(i = intMod(-j-k-color-1,2); i <= 31; i += 2) {
                        S0();
                        S1();
                        S2();
                    }}}}}

BZ is fixed to 64 (number of boxes)
Tune to find best value of TX, TY (dimensions of 2D block)

BX=64(box size)/TX, BY=64(box size)/TY

Tiled loop nest with loops marked for blocks/threads
for(box = 0; box <= 63; box++) {
    for(k = 1; k <= 64; k++) {
        for(jj = 0; jj <= 3; jj++) {
            for(j = 0; j <= 15; j++) {
                for(ii = 0; ii <= 1; ii++) {
                    for(i = intMod(-j-k-color-1,2); i <= 31; i += 2) {
                        S0();
                        S1();
                        S2();
                    }}}}}}

mark as block dim z (BZ=64)
mark as block dim y (BY=4)
mark as thread dim y (TY=16)
mark as block dim x (BX=2)
mark as thread dim x (TX=32)
/* gsrblua, variable coefficient GSRB, 64^3 box size */
init("gsrb_mod.cu", "gsrb",0,0)
dofile("cudaize.lua") # custom commands in lua

# set up parallel decomposition, adjust via autotuning
TI=32
TJ=4
TK=64
TZ=64
tile_by_index(0, {"box","k","j","i"},{TZ,TK,TJ, TI},
{l1_control="bb", l2_control="kk", l3_control="jj", l4_control="ii"},{"bb","box","kk","k","jj","j","ii","i"})
cudaize(0, "kernel_GPU",
{_temp=N*N*N*N,_beta_i=N*N*N*N,_phi=N*N*N*N},
{block={"ii","jj","box"}, thread={"i","j"}})
Performance on K20c

% DRAM Bandwidth achieved by GSRB
Smooth on 64, 64³ boxes

2D Thread Blocks <TX,TY>
Higher-order stencil promise huge reduction in data movement, but maybe bottlenecked by floating-point pressure and poor register reuse.
Higher-Order Stencils

<table>
<thead>
<tr>
<th>Stencil</th>
<th>Coefficient</th>
<th>Iteration</th>
<th>Flop</th>
<th>Byte</th>
<th>AI</th>
</tr>
</thead>
<tbody>
<tr>
<td>7-point</td>
<td>Constant</td>
<td>Jacobi</td>
<td>8</td>
<td>24</td>
<td>0.33</td>
</tr>
<tr>
<td>13-point</td>
<td>Constant</td>
<td>Jacobi</td>
<td>15</td>
<td>24</td>
<td>0.63</td>
</tr>
<tr>
<td>27-point</td>
<td>Constant</td>
<td>Jacobi</td>
<td>32</td>
<td>24</td>
<td>1.33</td>
</tr>
<tr>
<td>125-point</td>
<td>Constant</td>
<td>Jacobi</td>
<td>134</td>
<td>24</td>
<td>5.58</td>
</tr>
</tbody>
</table>

Higher-order stencil promise huge reduction in data movement, but maybe bottlenecked by floating-point pressure and poor register reuse.
for (j=0; j<N; j++)
for (i=0; i<N; i++){
    out[k][j][i] = w1* (in[j-1][i ] + in[j+1][i] +in[j ][i-1] + in[j ][i+1] ) + w2 *(in[j-1][i-1] + in[j+1][i-1] +in[j-1][i+1] + in[j+1][i+1] ) + w3* in[j ][i ];
}

Right (leading) edge of points from the input grid is reused in the next two iterations of the inner-loop

The right edge acts as the center and left edge for the next iterations
Partial Sums

Exploiting symmetry reduces flops significantly for 27, 125-pt stencils.

For 3D stencils we pick the leading plane.

For 125-pt stencil, 124 adds went down to 38 adds (over 3x reduction).
Smooth Performance

Partial sums takes the performance of the smoother to near the roofline bound.

Naively one may conclude reaching the bound is the upper end of performance.

Achieving memory bound implies we can now apply communication-avoiding optimizations!
Smooth Performance

Smother Performance (Fine Grid)

- All Optimizations
- +Fusion & Wavefront
- +Fusion & Partial Sums
- +Fusion
- Baseline
- Roofline Memory Bound

Transformation must work with other CA optimizations!
/* jacobi_box_4_64.py, 27-pt stencil, 64³ box size */
from chill import *

#select which computation to optimize
source('jacobi_box_4_64.c')
procedure('smooth_box_4_64')

loop(0)
original() # fuse wherever possible

# create a parallel wavefront
skew([[0,1,2,3,4,5],2],[2,1])
permute([2,1,3,4])

# partial sum for high order stencils and fuse result
distribute([[0,1,2,3,4,5],2])
stencil_temp(0)
stencil_temp(5)
fuse([2,3,4,5,6,7,8,9],1)
fuse([2,3,4,5,6,7,8,9],2)
fuse([2,3,4,5,6,7,8,9],3)
fuse([2,3,4,5,6,7,8,9],4)
Summary and Conclusions

• Compiler technology can be leveraged for automated architecture-specific optimization from high-level specification for several motifs

• Compiler technology allows composing a sequence of transformations, and mixing known and novel domain-specific optimizations

• Performance rivaling manually-tuned code and sometimes better
Extra Slides
Arithmetic Intensity (AI) of Stencil Computation

```c
for (k=0; k<N; k++)
for (j=0; j<N; j++)
for (i=0; i<N; i++){
    phi_out[k][j][i] = w1 * phi_in[k][j][i]
    + w2 * (phi_in[k+1][j][i] + phi_in[k-1][j][i])
    + phi_in[k][j+1][i] + phi_in[k][j-1][i]
    + phi_in[k][j+1][i] + phi_in[k][j-1][i]);
}
```

6 adds+2 muls = 8 flops

- Read $N^3$ Grid (phi_in)
- Write Allocate $N^3$ Grid (phi_out)
- Write $N^3$ Grid (phi_out)

Ideal cache behavior, compulsory (cold) misses only

\[
\text{Floating Point Ops (flops)} = \frac{8 \times N^3}{3 \times N^3 \times 8} = 0.33
\]
Machine Balance (Edison)

Floating Point Ops per second = 460.8
DRAM Memory Bandwidth = 88

Machine Balance = 5.2

To achieve peak performance, code needs to execute 5.2 flops per byte!
for (j=0; j<N; j++)
  for (i=0; i<N; i++){
    out[k][j][i] = w1*
    (in[j-1][i ] + in[j+1][i]
    +in[j ][i-1] + in[j ][i+1] )
    + w2 *(in[j-1][i-1] + in[j+1][i-1]
    +in[j-1][i+1] + in[j+1][i+1] )
    + w3* in[j ][i ];
  }
Buffering Partial Sums: Exploiting Reuse

\[
\text{out } [j][i] = R[i] + C[i] + L[i]
\]
Exploiting Symmetry to Reduce Computation

Due to symmetry of coefficients the loaded edge (plane) can be factored into sums.

2D 9-point stencil loads the right edge of points.

Factors are reused to compute partial sums saving flops (5 adds instead of 8).

loads the right edge of points

factor the points multiplied by the same coefficient

for (j=0; j<N; j++)
  for (i=0; i<N; i++){
out[k][j][i] = w1*(in[j-1][i ] + in[j+1][i] + in[j ][i-1] + in[j ][i+1]) + w2 *(in[j-1][i-1] + in[j+1][i-1] + in[j-1][i+1] + in[j+1][i+1]) + w3* in[j ][i ];
  }

r1 = in[j][i+1];
r2 = in[j+1][i+1] + in[j-1][i +1];
R[i] = w1 * r1 + w2 * r2;
C[i+1] = w3 * r1 + w1 * r2;
L[i+2] = R[i];
out[j][i] = L[i] + C[i]+ R[i];
Exploiting Symmetry to Reduce Computation

Exploiting symmetry reduces flops significantly for 27, 125-pt stencils

For 3D stencils we pick the leading plane

For 125-pt stencil, 124 adds went down to 38 adds (over 3x reduction)
miniGMG

Domain $256^3$

List of $64^3$ Boxes Computed In Parallel (OMP)

Domain decomposed to MPI processes (2)

Smooth Dominates Runtime

4 iterations of smooth

GMG V-cycle

48 iterations of Smooth