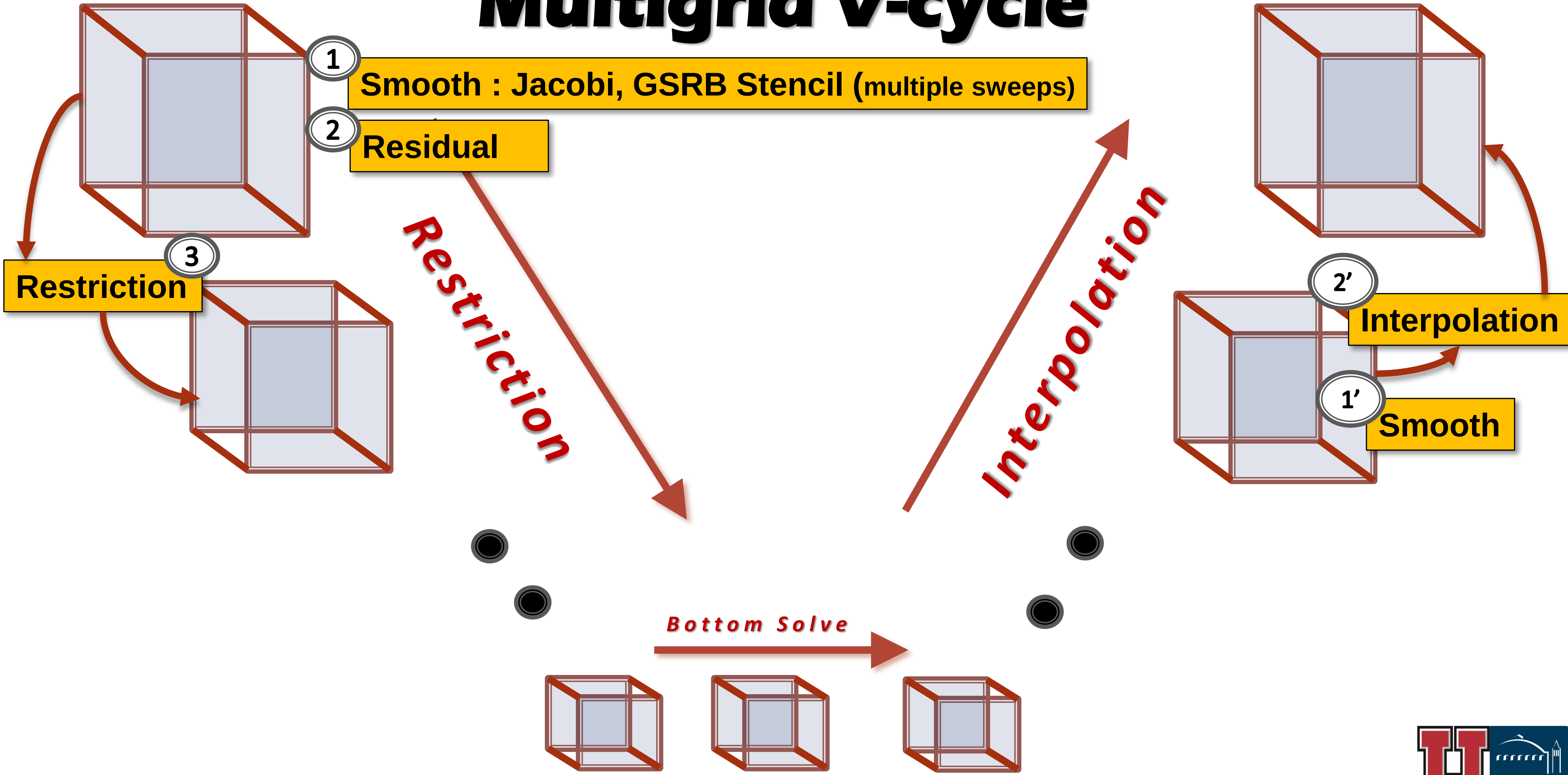
Compiler Generation and Autotuning of Communication-Avoiding Operators for Geometric Multigrid

Protonu Basu*, Samuel Williams†, Brian Van Straalen†, Anand Venkat*, Leonid Oliker†, Mary Hall*

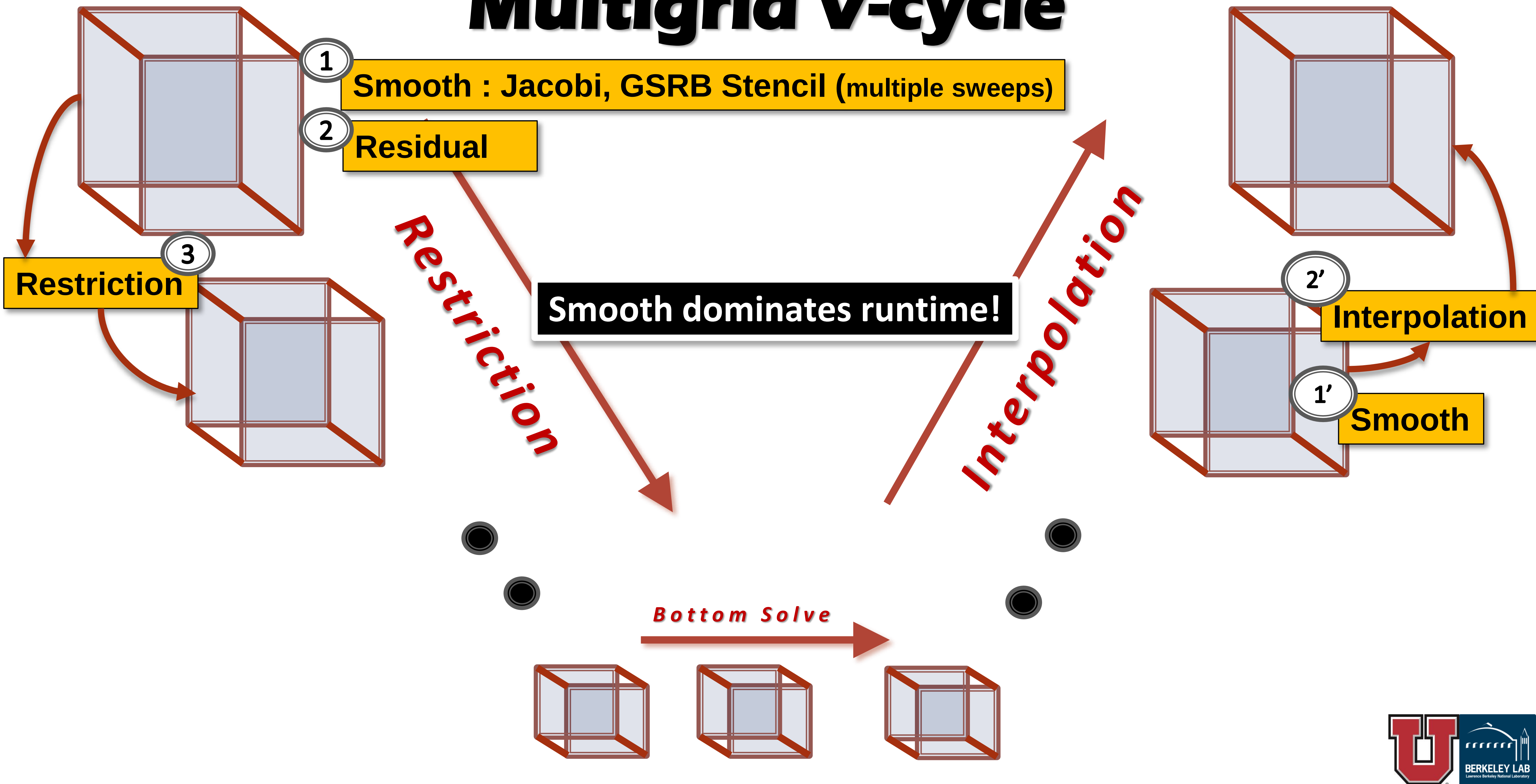
*University of Utah † Lawrence Berkeley National Laboratory
protonu@cs.utah.edu

http://ctop.cs.utah.edu/x-tune/

Multigrid V-cycle



Multigrid V-cycle



Compiler-Based Optimization of Smooth

Optimization Using Known Transformations

- Loop skew, permute and tiling

New Domain Specific Transformations

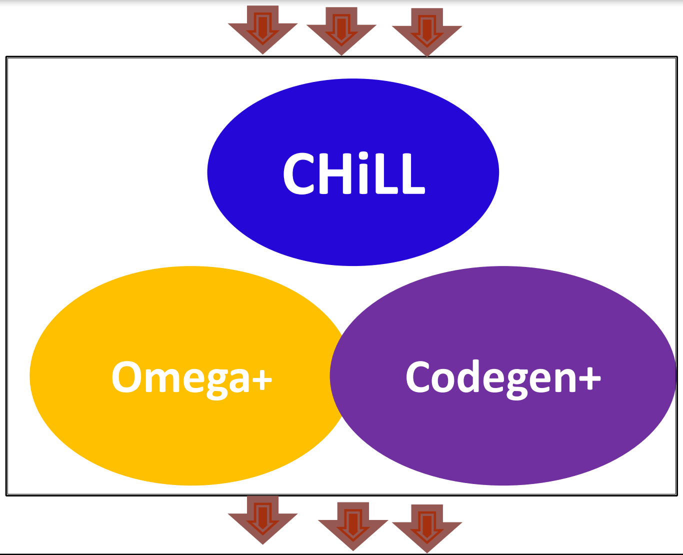
- Loop fusion in presence of fusion preventing dependences
- Adding ghost zones (communication avoiding) to Multigrid operators

High Performance OpenMP Code Generation

Optimizations Built into CHiLL

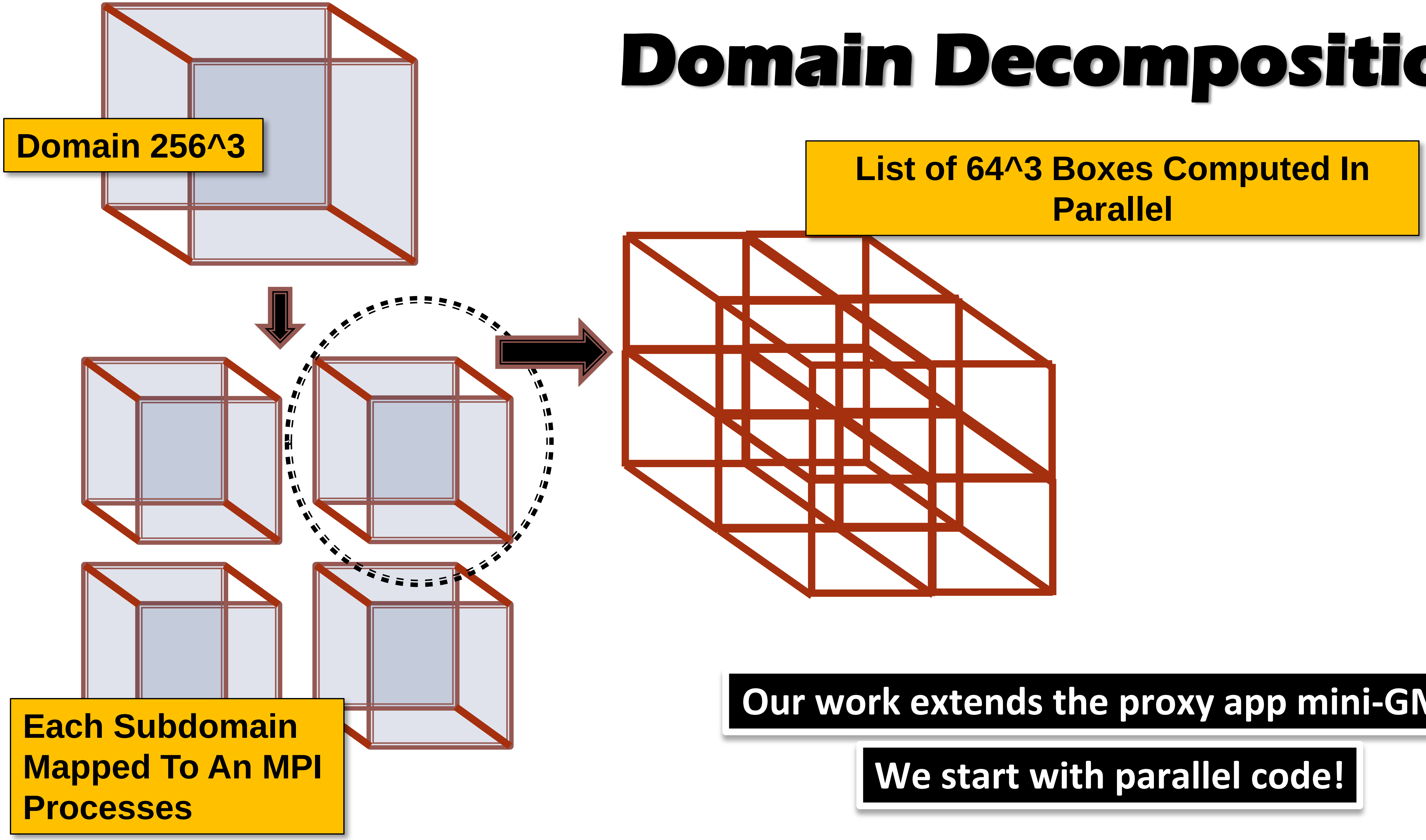
- CHiLL is loop transformation framework with a script interface

Autotuner



Smooth Variants

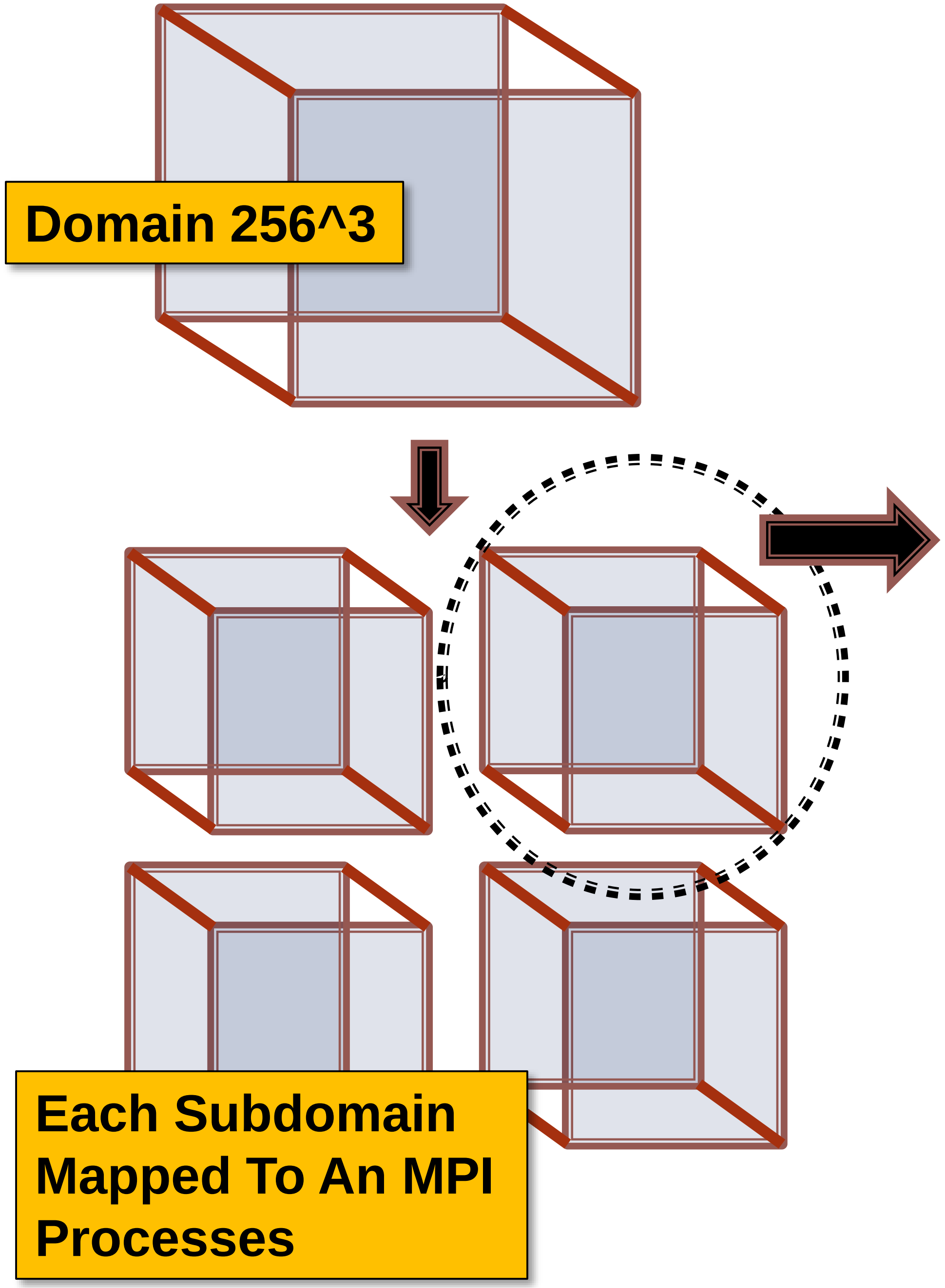
Domain Decomposition



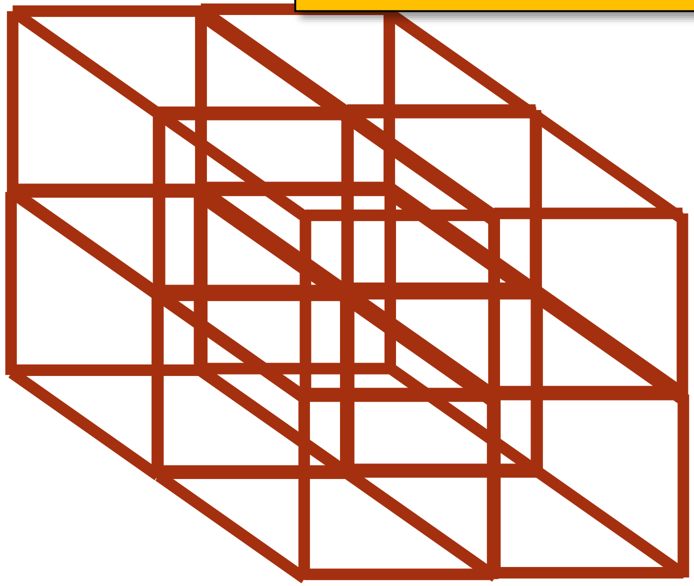
Our work extends the proxy app mini-GMG

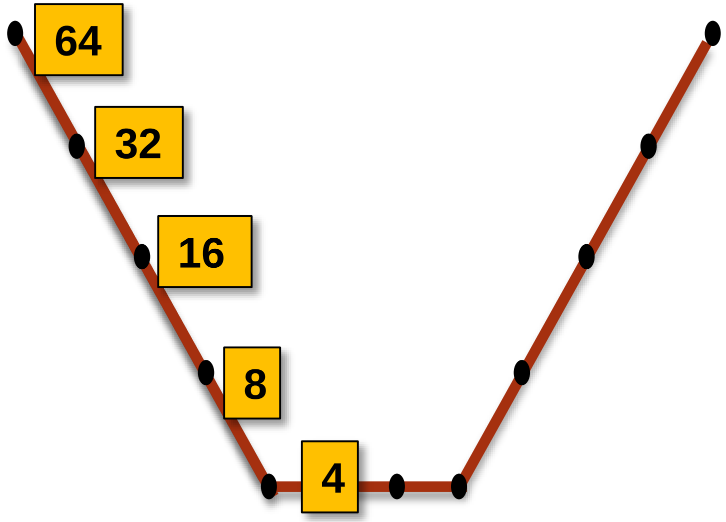
We start with parallel code!

Domain Decomposition



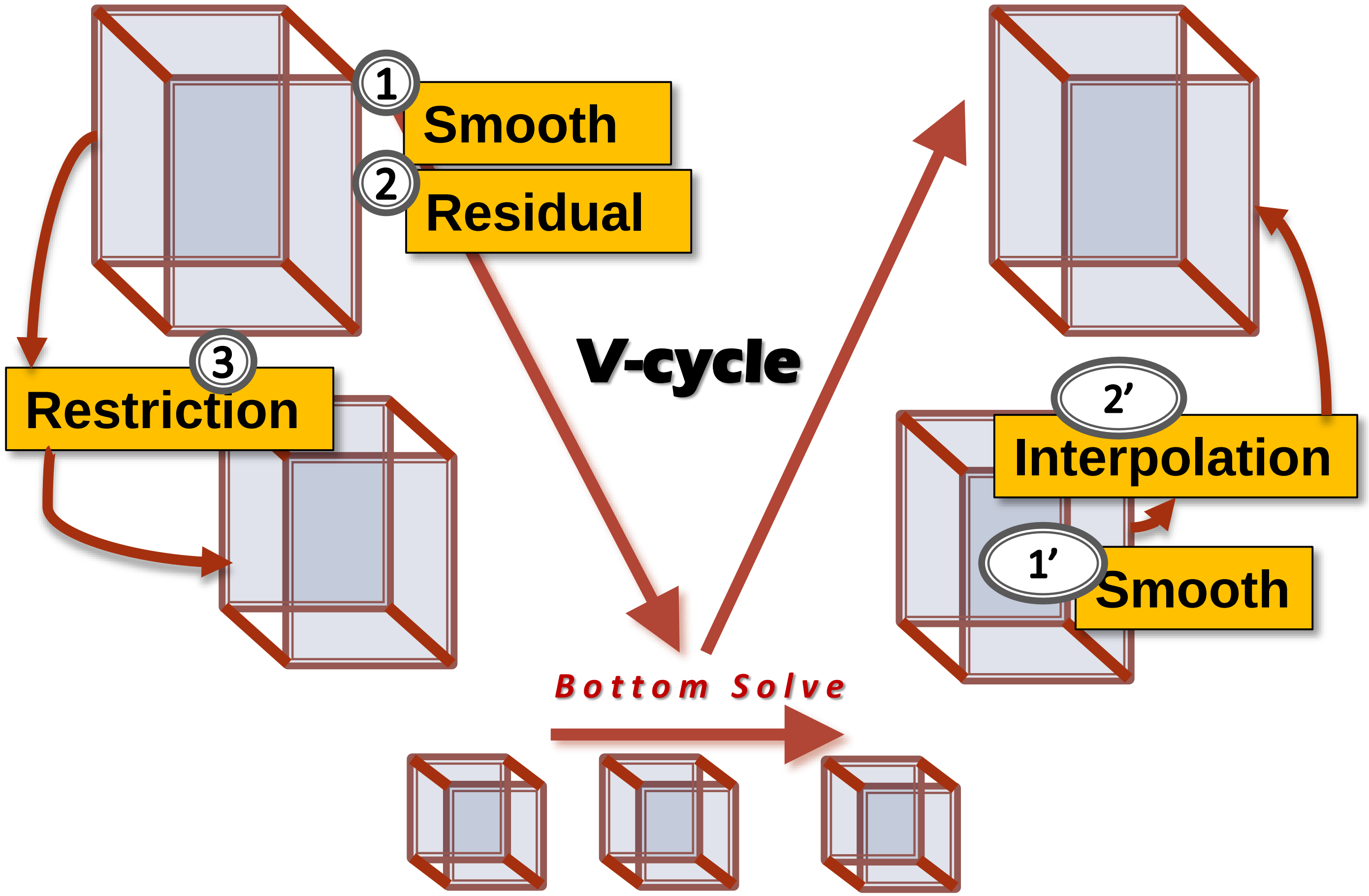
List of 64^3 Boxes Computed In Parallel





Our work extends the proxy app mini-GMG

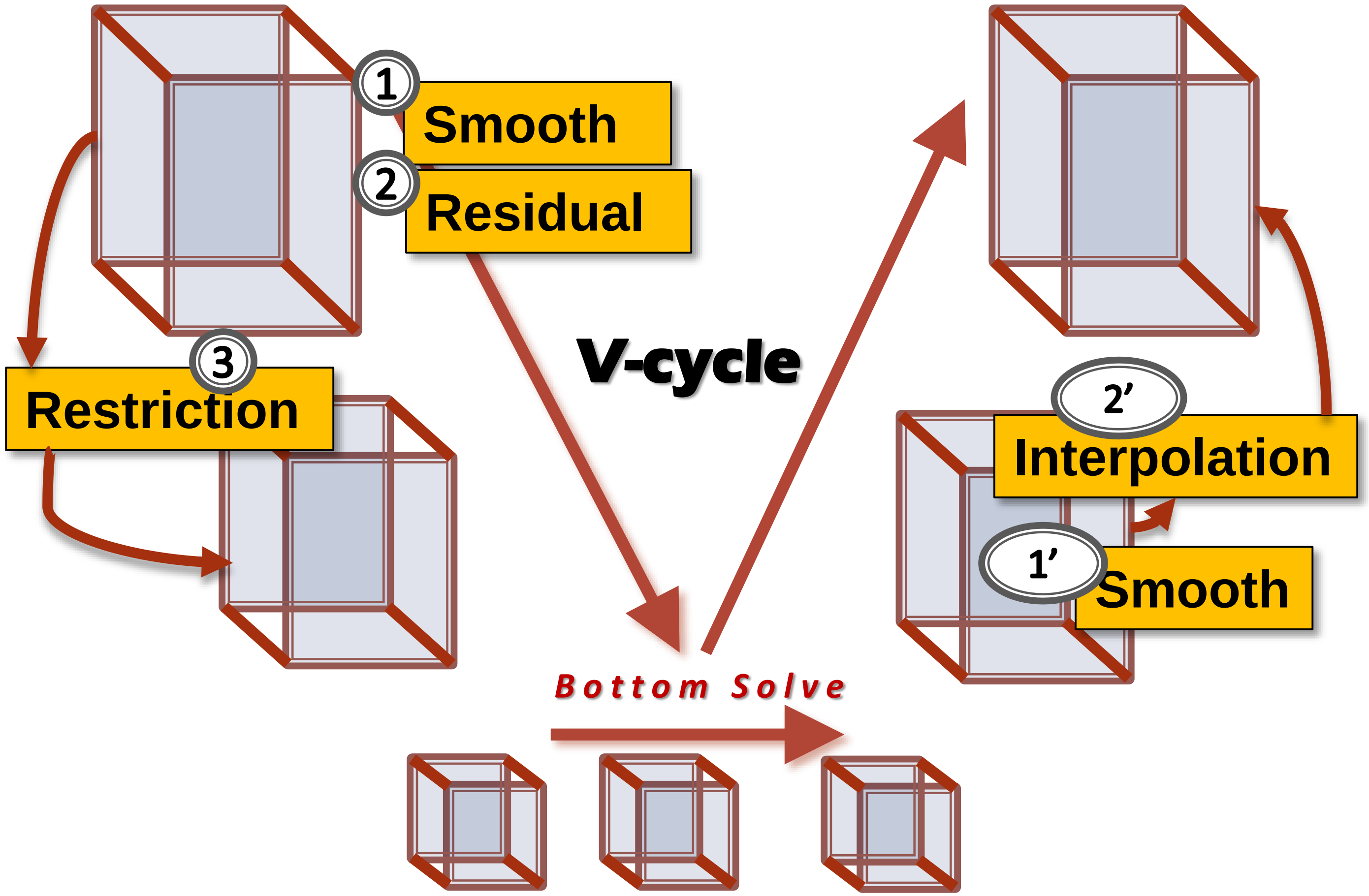
We start with parallel code!



Solve : Lu = f

Double precision, finite volume discretization of the variable-coefficient operator
$$L = a\vec{\alpha}I - b\nabla\vec{\beta}\nabla$$
with periodic boundary conditions

Right hand side (f) is:
$$\sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$$
on the [0,1] cubical domain



Solve : Lu = f

Double precision, finite volume discretization of the variable-coefficient operator
$L = a\vec{\alpha}I - b\nabla\cdot\vec{\beta}\nabla$
with periodic boundary conditions

Right hand side (f) is:
sin(2πx)sin(2πy)sin(2πz)
on the [0,1] cubical domain

8 GSRB sweeps per level of the V-cycle (4 going down and 4 coming back up)

48 GSRB sweeps at the bottom level

Smooth Operator

```
/* Laplacian(phi) = b div beta grad phi */
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][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(phi) = (a alpha I – Laplacian) * phi */
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];
```

```
/* GSRB relaxation  phi = phi – lambda(Helmholtz – rhs) */
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]);}
```

```
for (k...){ for (j...){ for (i...) { // LAPLACIAN OPERATOR }}}
for (k...){ for (j...){ for (i...) { // HELMHOLTZ OPERATOR}}}
for (k...){ for (j...){ for (i...)
{ if ((i+j+k+color+1)%2) // RED BLACK GAUSS SEIDEL OPERATOR}}}
```

Smooth Operator

```
for (k…){ for (j…){ for (i…) { // LAPLACIAN OPERATOR }}}
for (k…){ for (j…){ for (i…) { // HELMHOLTZ OPERATOR}}}
for (k…){ for (j…){ for (i…)
{ if ((i+j+k+color+1)%2) // RED BLACK GAUSS SEIDEL OPERATOR}}}
```

3D 7-Point, Variable Coefficient Stencil

```
/* Laplacian(phi) = b div beta grad phi */
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][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(phi) = (a alpha I – Laplacian) * phi */
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];
```

```
/* GSRB relaxation  phi = phi – lambda(Helmholtz – rhs) */
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]);}
```

Smooth Operator

```
/* Laplacian(phi) = b div beta grad phi */
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][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(phi) = (a alpha I – Laplacian) * phi */
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];
```

```
/* GSRB relaxation  phi = phi – lambda(Helmholtz – rhs) */
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]);}
```

```
for (k...){ for (j...){ for (i...) { // LAPLACIAN OPERATOR }}}
for (k...){ for (j...){ for (i...) { // HELMHOLTZ OPERATOR}}}
for (k...){ for (j...){ for (i...)
{ if ((i+j+k+color+1)%2) // RED BLACK GAUSS SEIDEL OPERATOR}}}
```

3D 7-Point, Variable Coefficient Stencil

Smooth Operator Optimization

Loop fusion across operators
Introduce ghost zones (halo regions)
Create a wavefront computation
OpenMP parallel code generation

Loop Fusion

```
/* Laplacian(phi) = b div beta grad phi */
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][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(phi) = (a alpha I – Laplacian) * phi */
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];
```

```
/* GSRB relaxation  phi = phi – lambda(Helmholtz – rhs) */
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]);}
```

Loop Fusion

7-point stencil

Fusion-preventing dependences

```
/* Laplacian(phi) = b div beta grad phi */
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][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(phi) = (a alpha I – Laplacian) * phi */
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];
```

```
/* GSRB relaxation  phi = phi – lambda(Helmholtz – rhs) */
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]);}
```

Loop Fusion

```
/* Laplacian(phi) = b div beta grad phi */
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][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] ) );
```

7-point stencil

Fusion-preventing dependences

```
/* Helmholtz(phi) = (a alpha I – Laplacian) * phi */
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];
```

```
/* GSRB relaxation  phi = phi – lambda(Helmholtz – rhs) */
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]);}
```

Solution: Array Data-Flow

Array data-flow analysis (temp) determines it is safe to contract iteration space for Laplacian and Helmholtz.

Loop Fusion

```
/* Laplacian(phi) = b div beta grad phi */
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][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] ) );
```

7-point stencil

Fusion-preventing dependences

```
/* Helmholtz(phi) = (a alpha I – Laplacian) * phi */
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];
```

```
/* GSRB relaxation  phi = phi – lambda(Helmholtz – rhs) */
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]);}
```

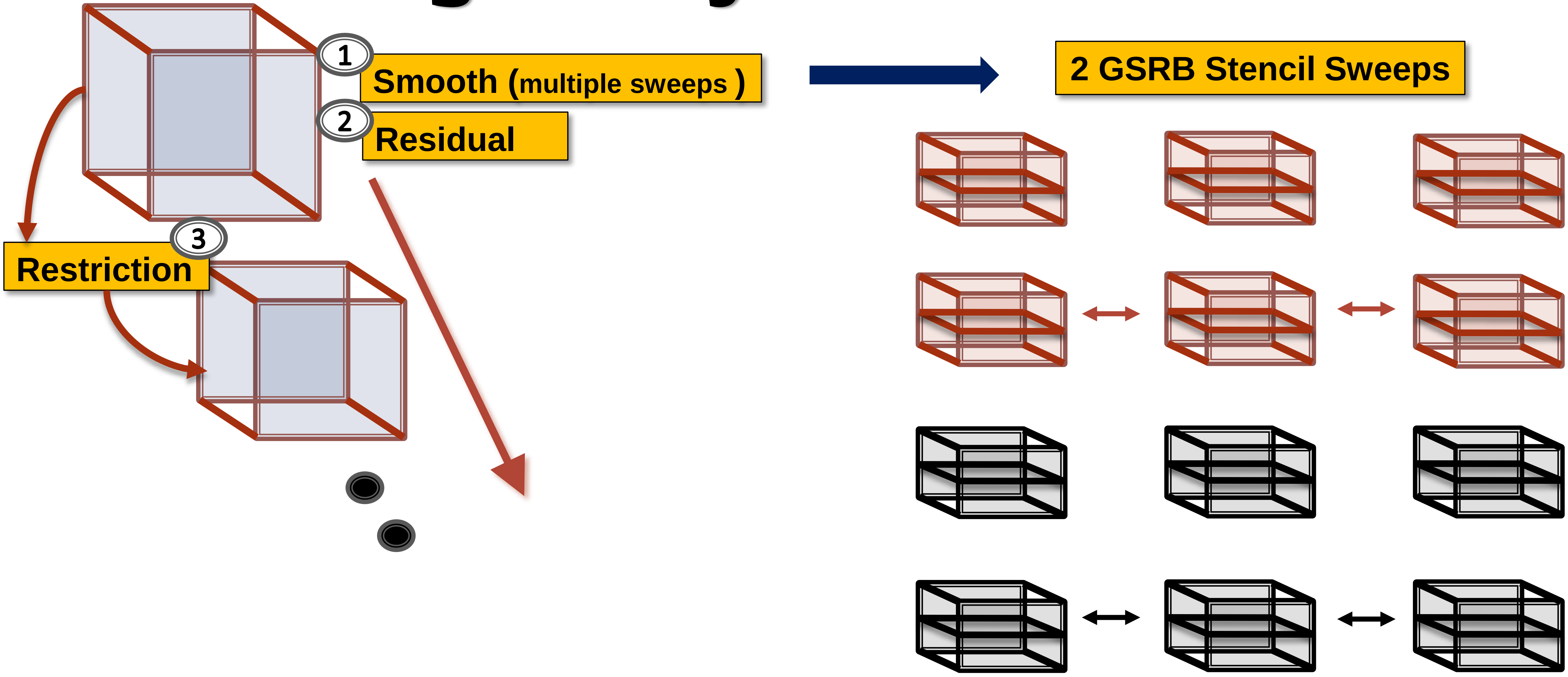
Solution: Array Data-Flow

Array data-flow analysis (temp) determines it is safe to contract iteration space for Laplacian and Helmholtz.

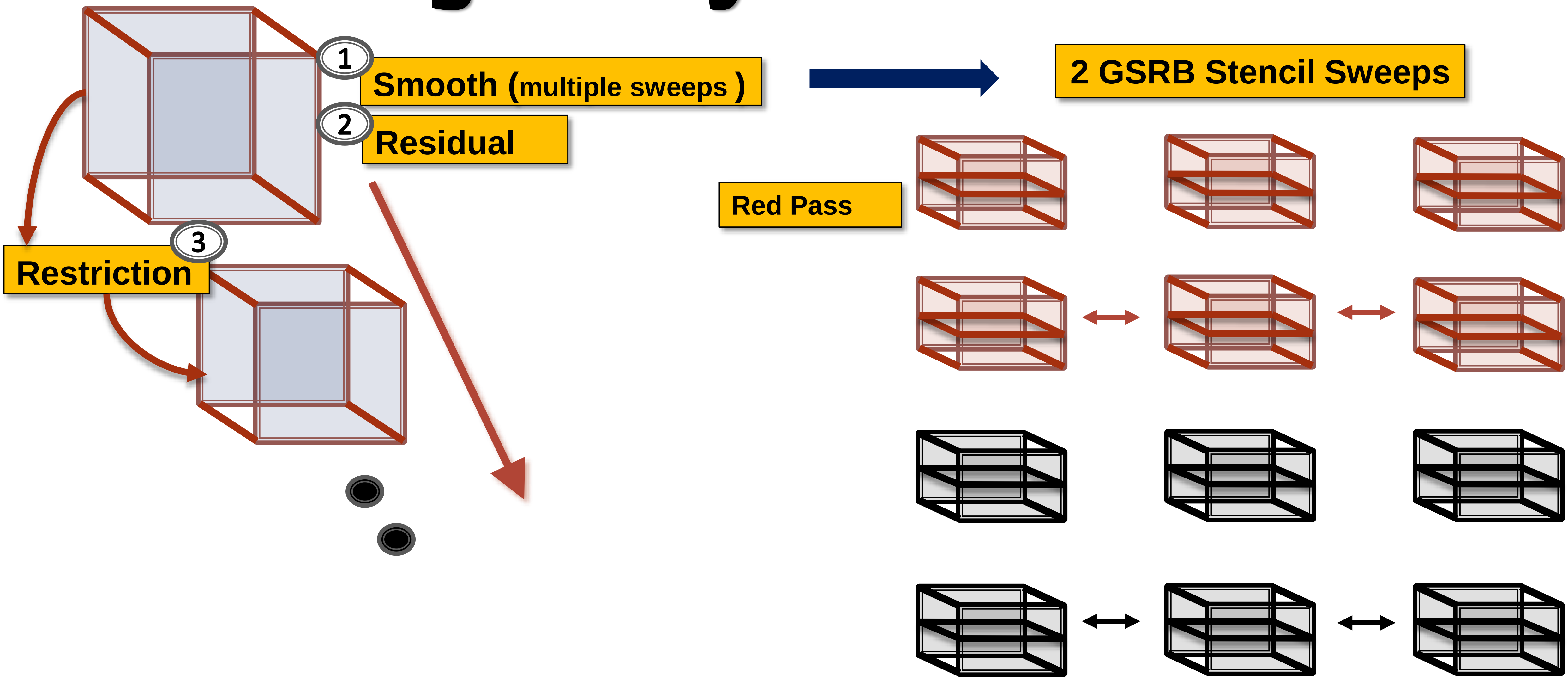
Fused

```
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) {
          S0(k,j,i); /* Laplacian  */
          S1(k,j,i); /* Helmholtz */
          S2(k,j,i); /* GSRB       */
      }
```

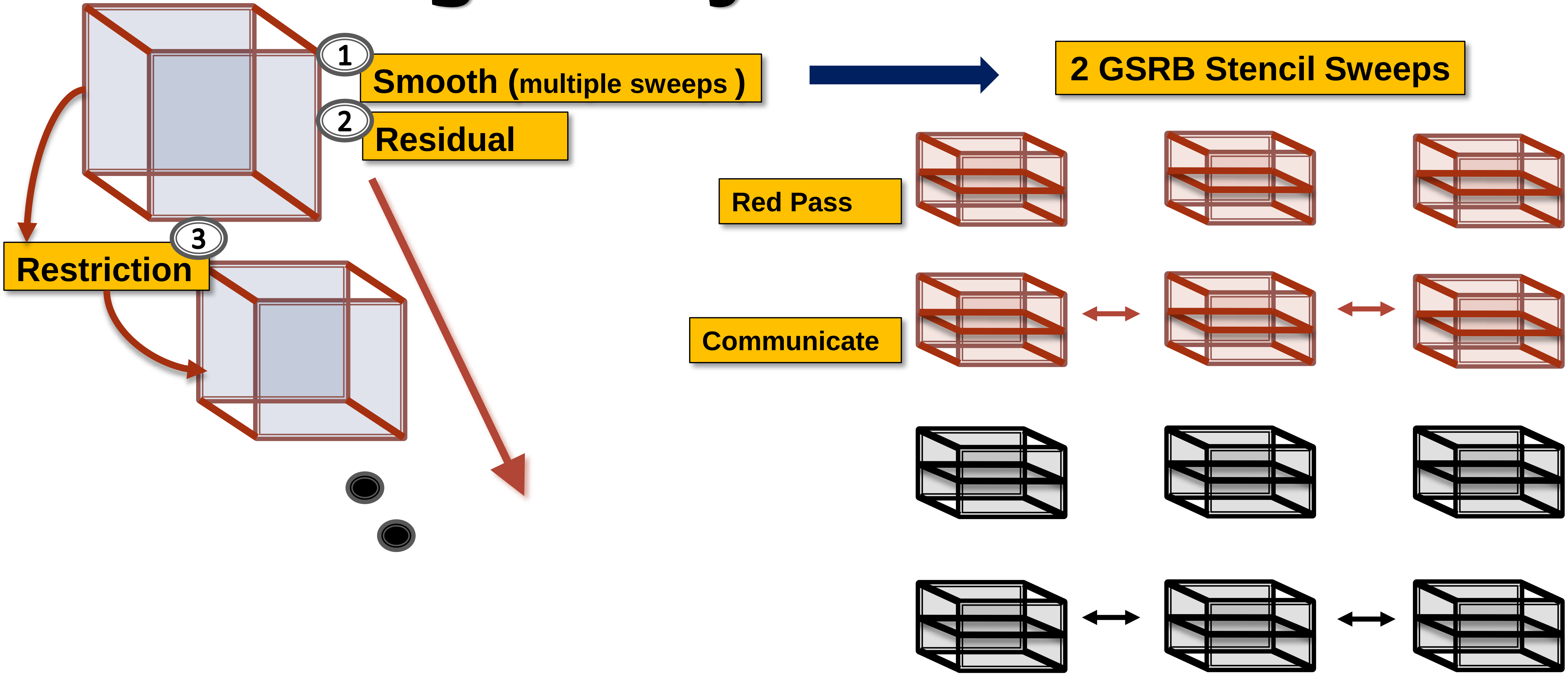
Multigrid V-cycle : Closer Look



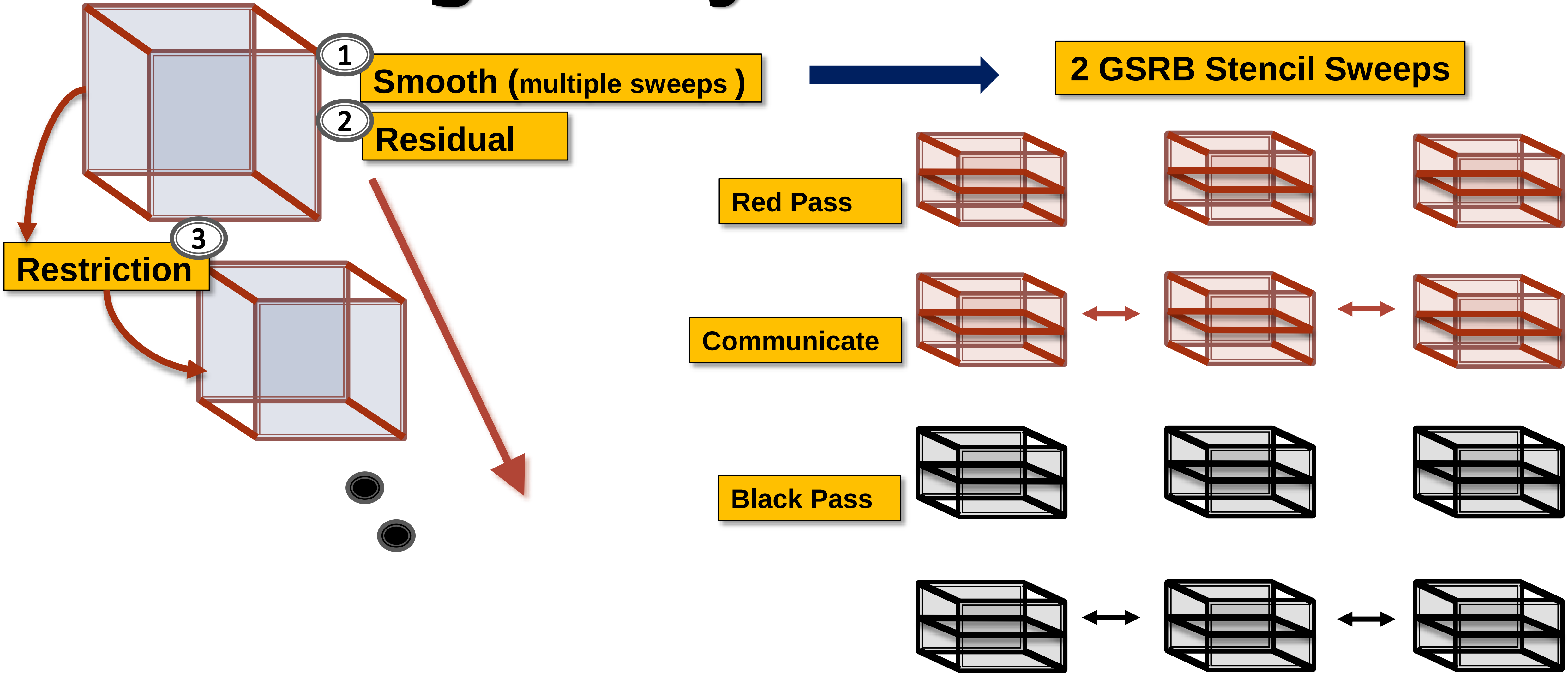
Multigrid V-cycle : Closer Look



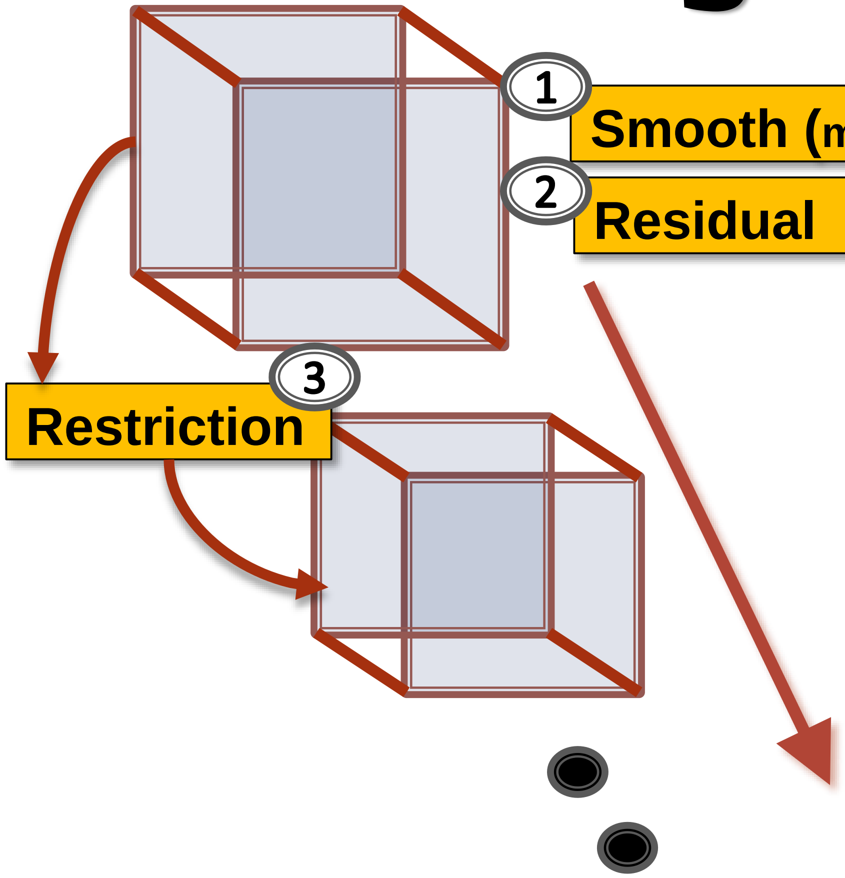
Multigrid V-cycle : Closer Look

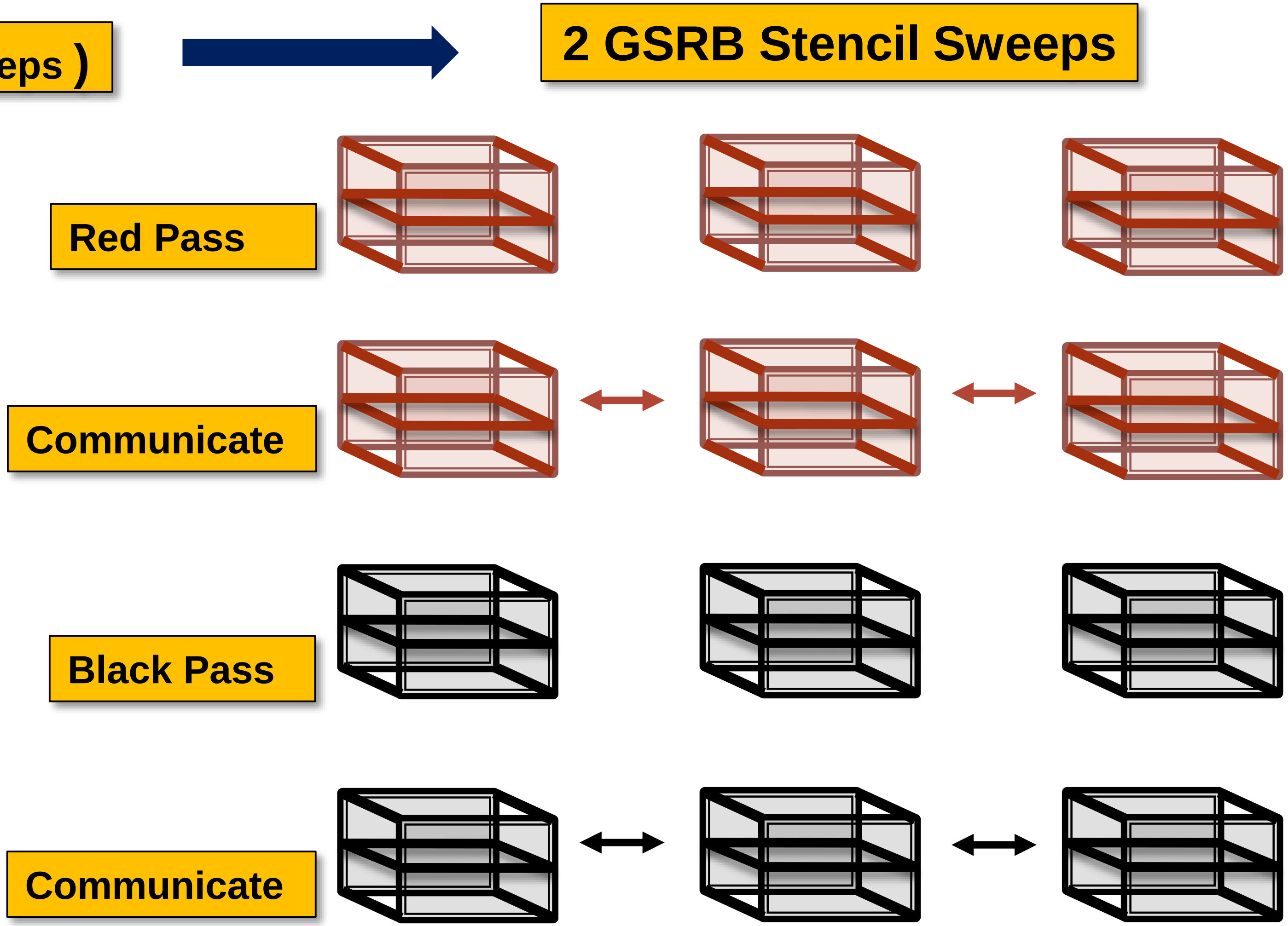


Multigrid V-cycle : Closer Look



Multigrid V-cycle : Closer Look

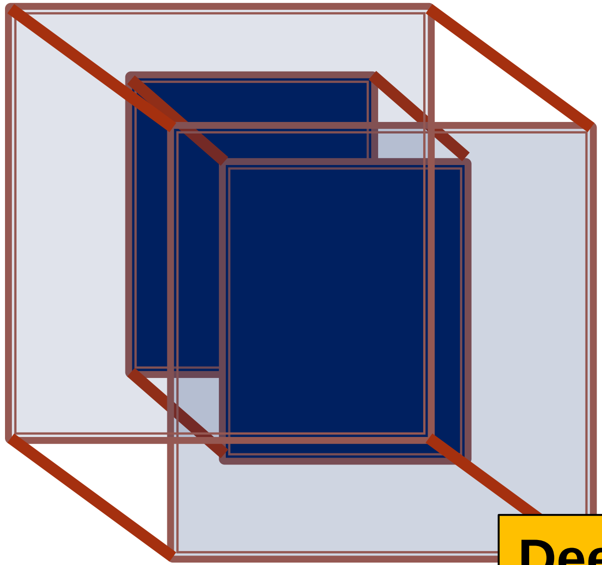




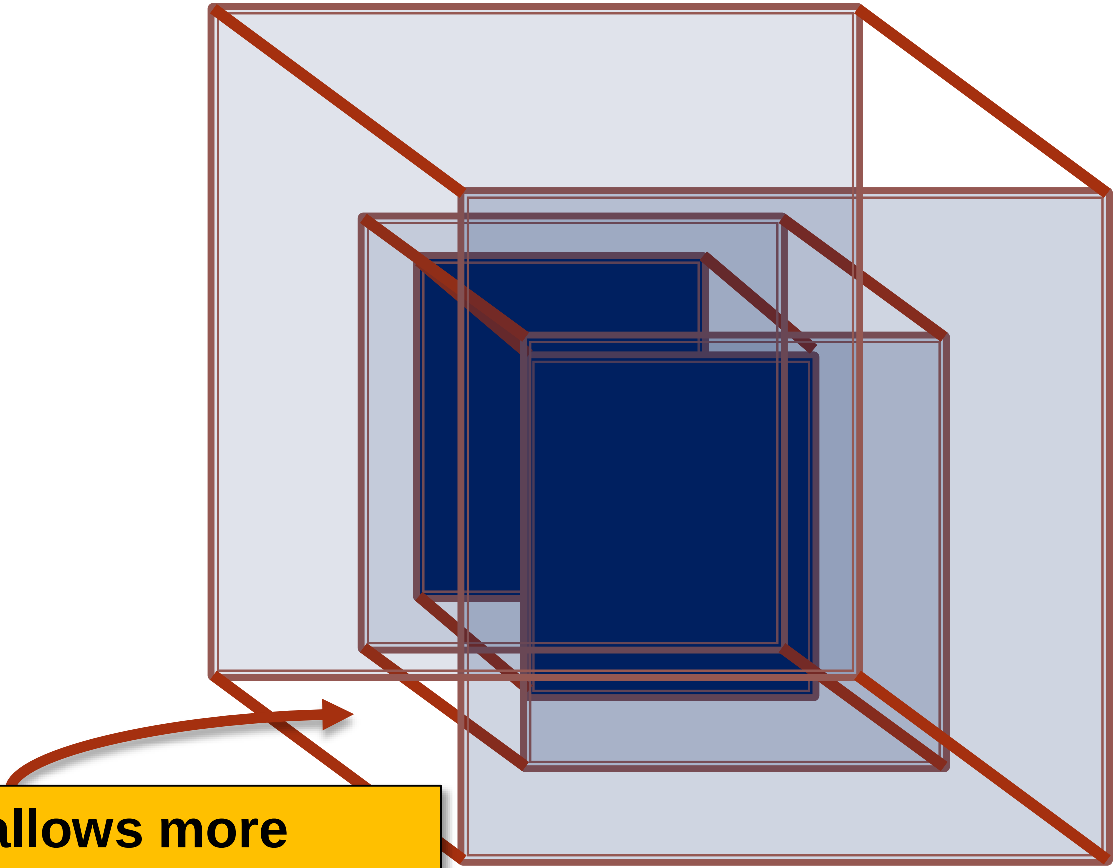
Adding Ghost Zones : Compute More, Communicate Less

Each grid sweep consumes one ghost layer

One communication step per sweep



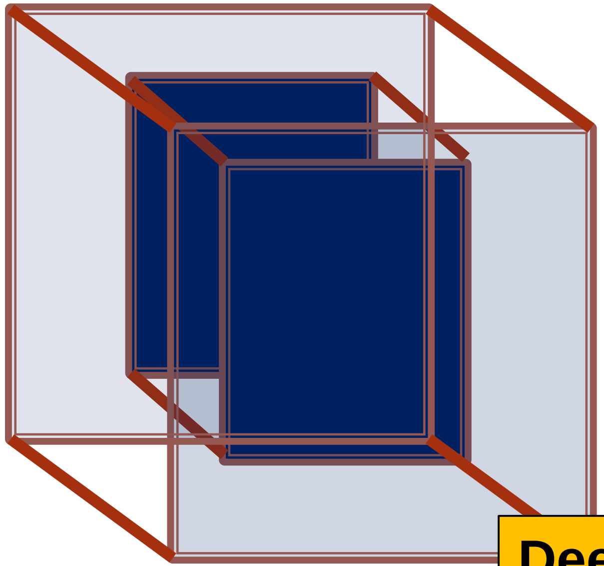
Deeper ghost zone allows more sweeps per communication step

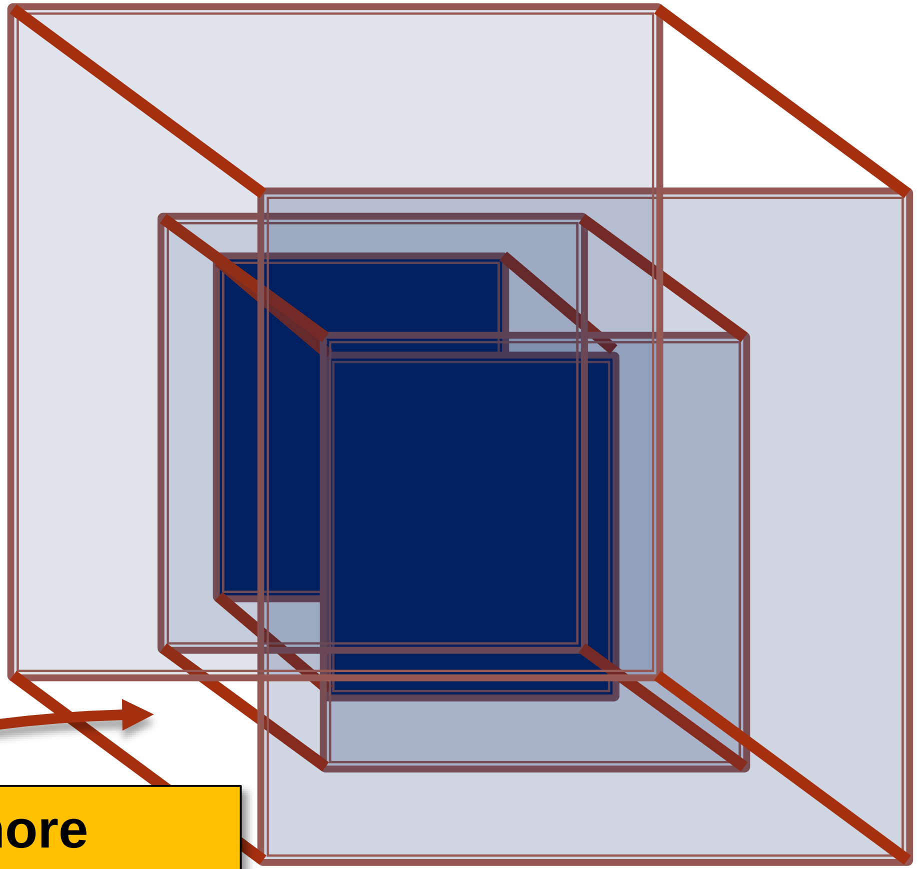


Adding Ghost Zones : Compute More, Communicate Less

Each grid sweep consumes one ghost layer

One communication step per sweep





Deeper ghost zone allows more sweeps per communication step

Ghost zone depth depends on box size and machine!

Adding Ghost Zone

Fused

```
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) {
            S0(k,j,i); /* Laplacian  */
            S1(k,j,i); /* Helmholtz */
            S2(k,j,i); /* GSRB       */
         }
```

Ghost Zones

```
/* d = ghost zone depth */
for (t=0; t<d; t++)
   for (k=t-(d-1); k<N+(d-1)-t; k++)
      for (j=t-(d-1); j<N+(d-1)-t; j++)
         for (i=t-(d-1); i<N+(d-1)-t; i++) {
            if ((i+j+k+color+t)%2 == 0) {
               S0(k,j,i); /* Laplacian  */
               S1(k,j,i); /* Helmholtz */
               S2(k,j,i); /* GSRB       */
            }
```

Add Time Step Loop

Vary Iteration Space with Time

Update Conditional

Adding Ghost Zone

Fused

```
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) {
            S0(k,j,i); /* Laplacian  */
            S1(k,j,i); /* Helmholtz */
            S2(k,j,i); /* GSRB       */
         }
```

Solution

Expand iteration space (compiler abstraction) to include ghost zone. Code generation involves scanning polyhedra described by iteration space

Ghost Zones

```
/* d = ghost zone depth */
for (t=0; t<d; t++)
   for (k=t-(d-1); k<N+(d-1)-t; k++)
      for (j=t-(d-1); j<N+(d-1)-t; j++)
         for (i=t-(d-1); i<N+(d-1)-t; i++) {
            if ((i+j+k+color+t)%2 == 0) {
               S0(k,j,i); /* Laplacian  */
               S1(k,j,i); /* Helmholtz */
               S2(k,j,i); /* GSRB       */
            }
```

Add Time Step Loop

Vary Iteration Space with Time

Update Conditional

Adding Ghost Zone

Fused

```
for (k=0; k<N; k++)
   for (j=0; j<N; j++)
      for (i=0; i<N; i++) {
         if ((i+j+k+color)%
            S0(k,j,i); /* Laplacian  */
            S1(k,j,i); /* Helmholtz */
            S2(k,j,i); /* GSRB       */
         }
```

Original Iteration Space

S0 := { [k,j,i] : 0 ≤ k < N && 0 ≤ j < N && 0 ≤ i < N && k+j+I + 2α + color = 0};

Relations to Map to New Iteration Space

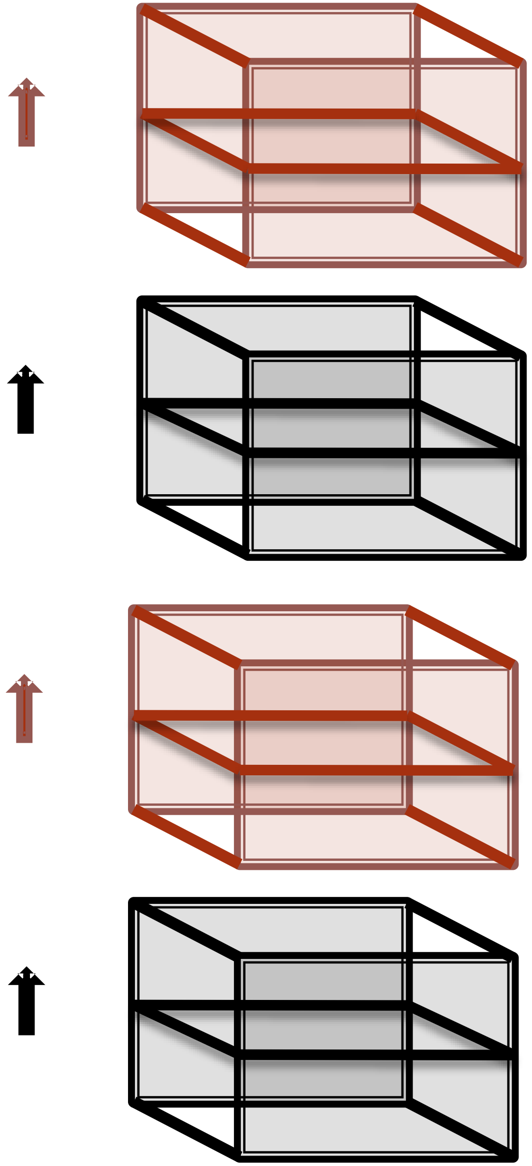
Ghost Zones

```
/* d = ghost zone depth */
for (t=0; t<d; t++)
   for (k=t-(d-1); k<N+(d-1)-t; k++)
      for (j=t-(d-1); j<N+(d-1)-t; j++)
         for (i=t-(d-1); i<N+(d-1)-t; i++) {
            if ((i+j+k+color+t)%2 == 0) {
               S0(k,j,i); /* Laplacian  */
               S1(k,j,i); /* Helmholtz */
               S2(k,j,i); /* GSRB       */
            }
```

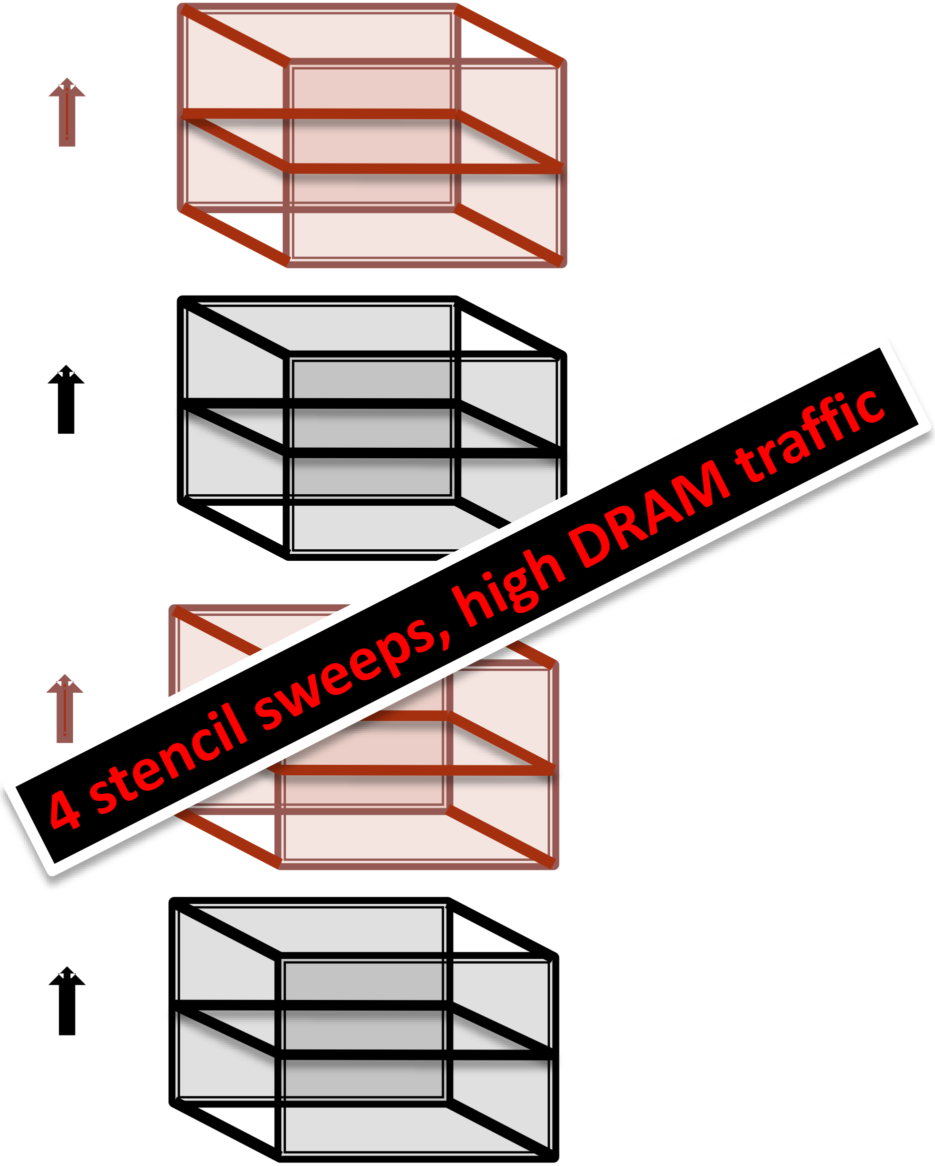
map1 := { [k,j,i] ➔ [t,k',j',i'] :
0 ≤ t < d &&
k-d+1+t ≤ k'< k+d-t &&
j-d+1+t ≤ j' < j+d-t &&
i-d+1+t ≤ I' < i+d-t};

map2 := {[color] ➔ [color+t]};

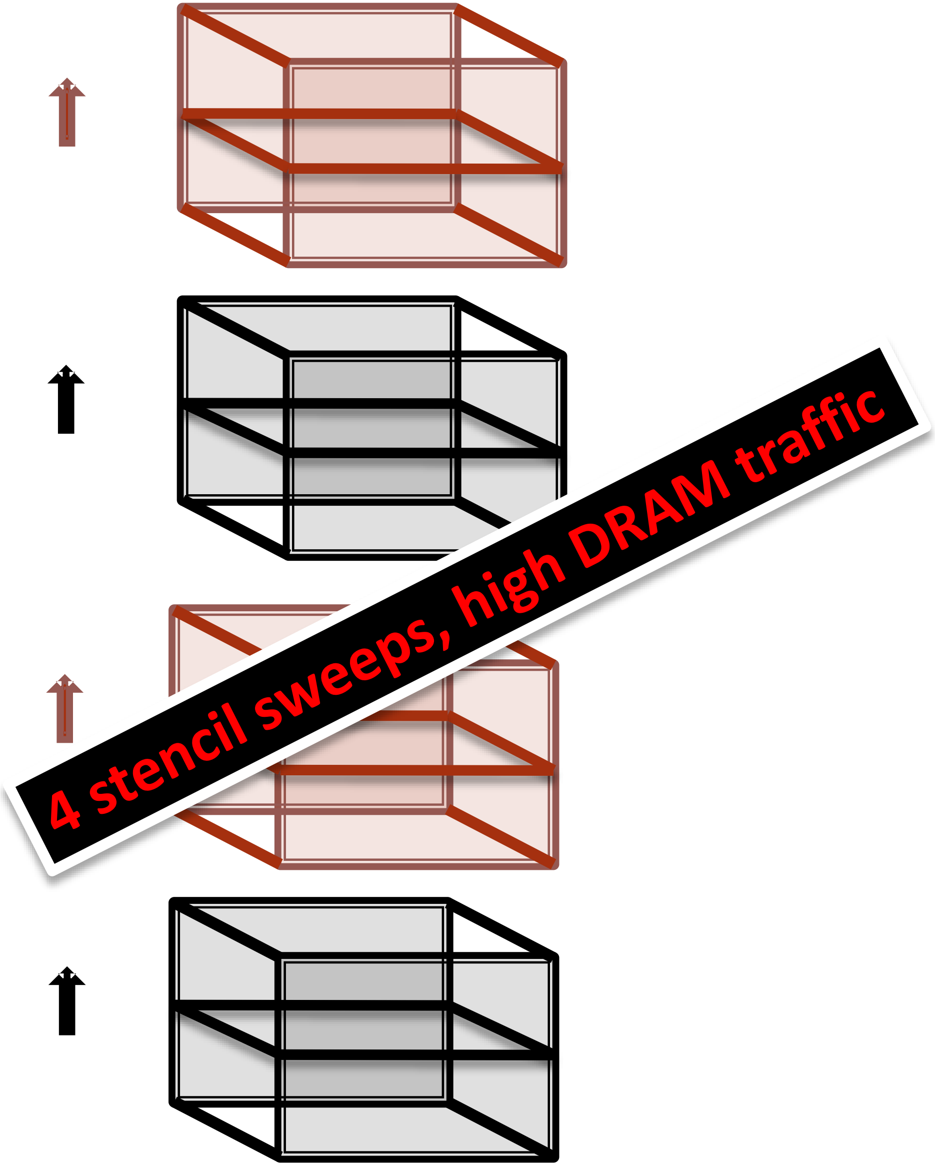
Wavefront

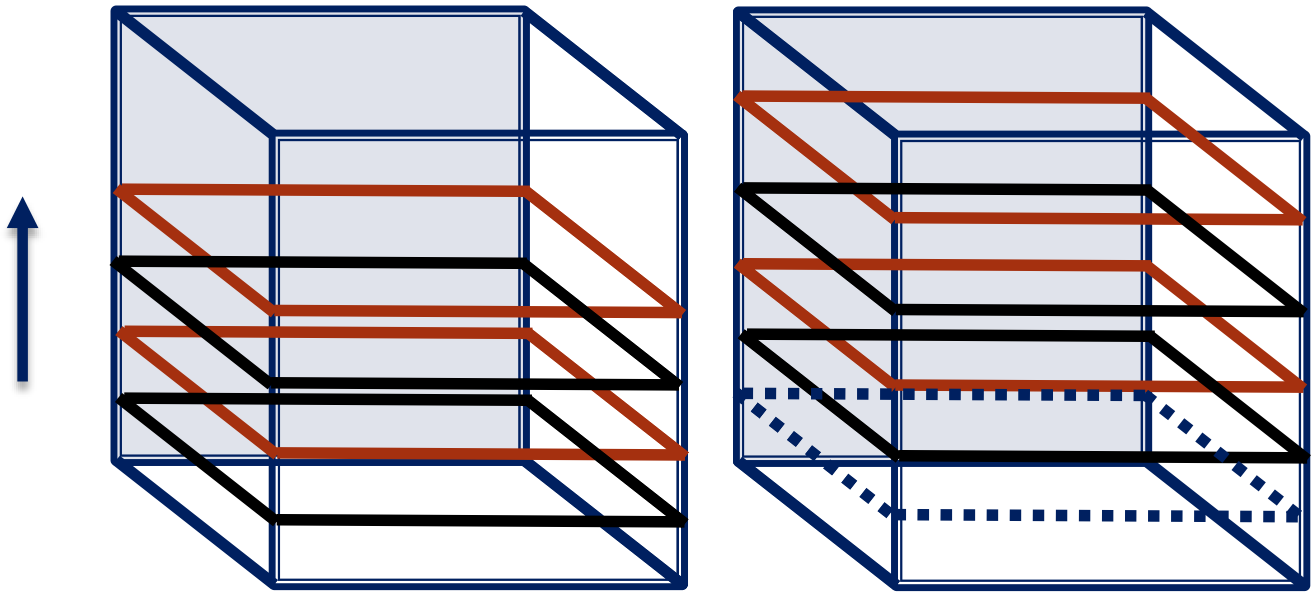


Wavefront

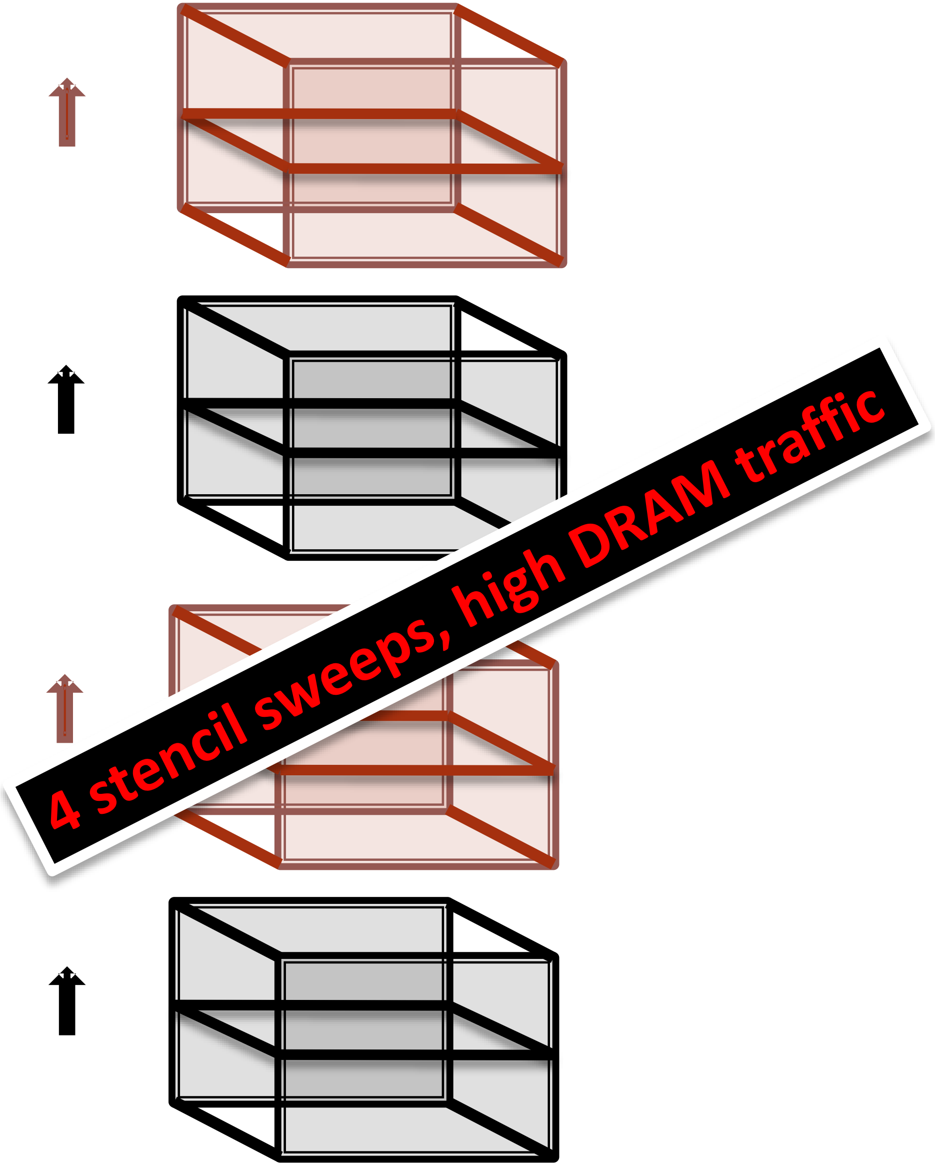


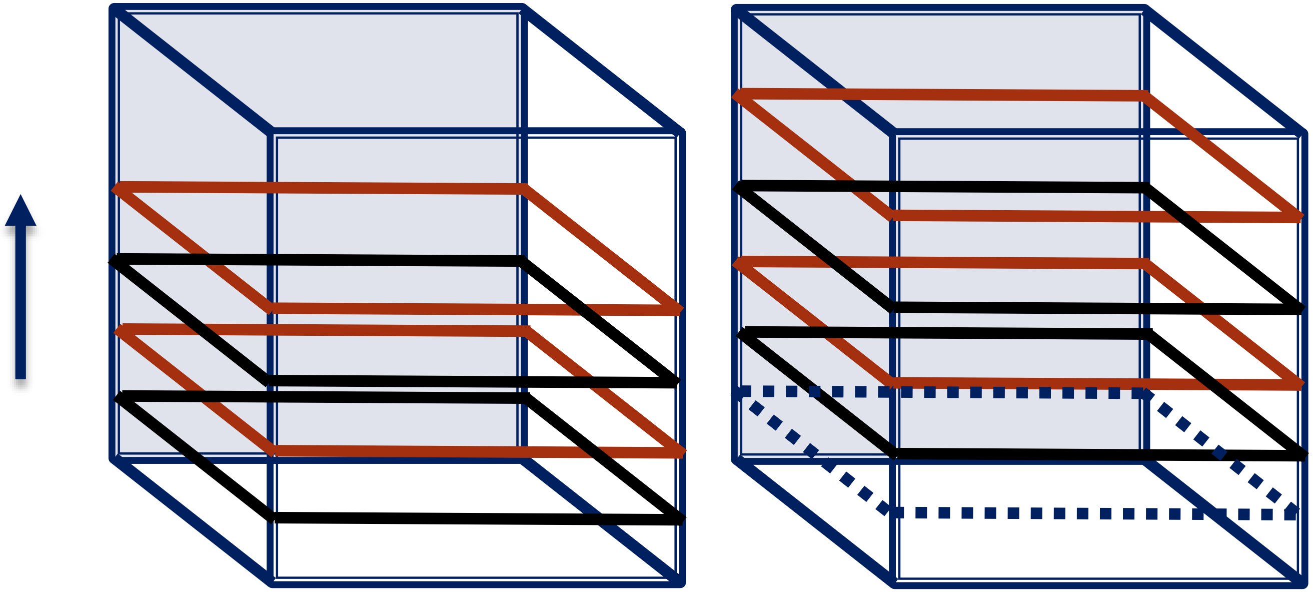
Wavefront





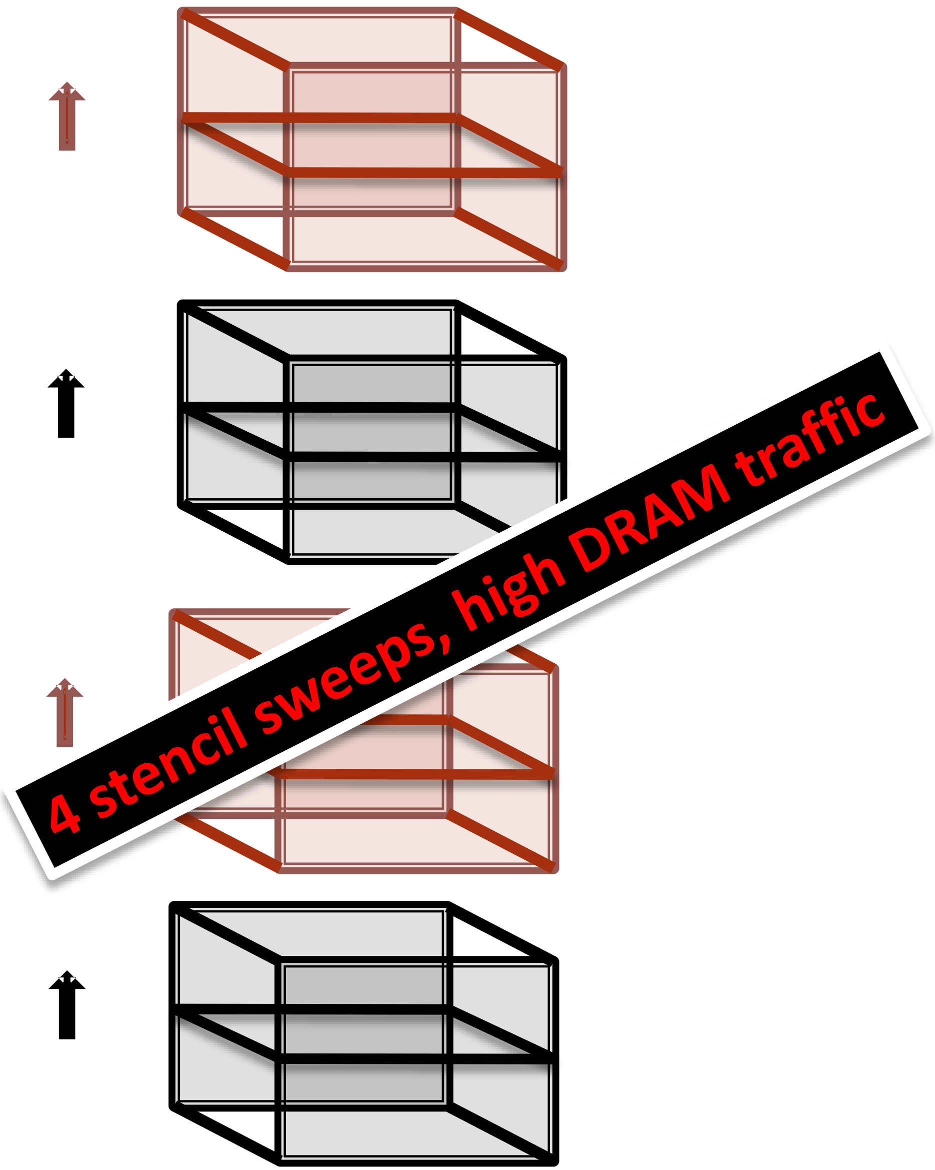
Wavefront



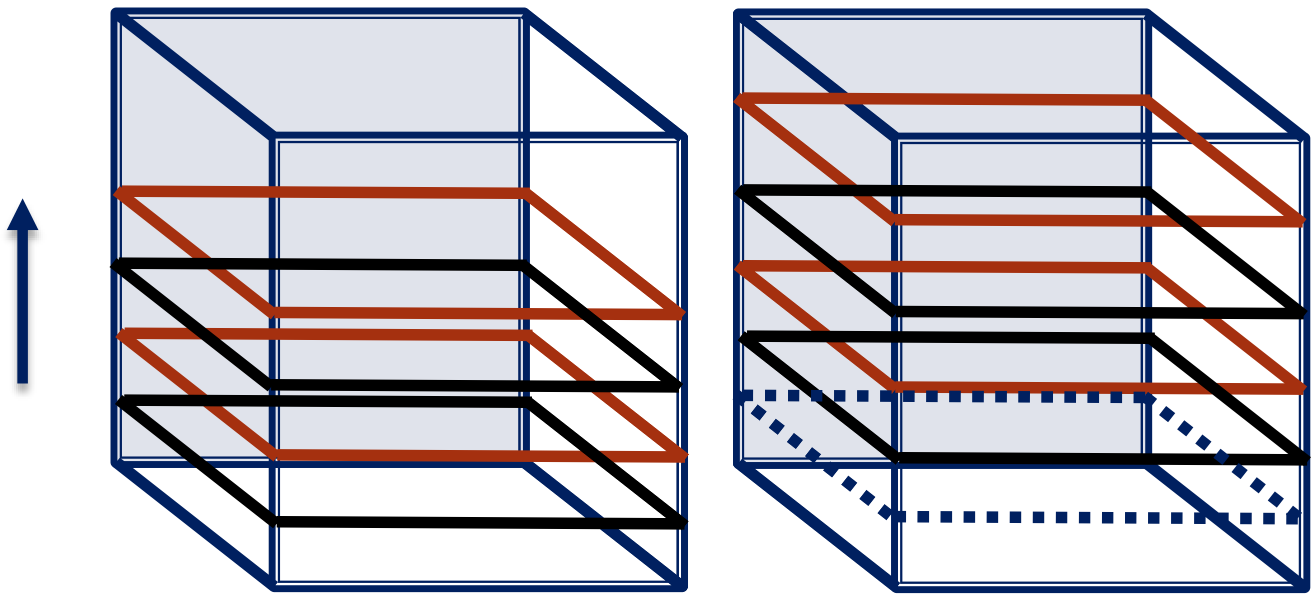


- Single stencil sweep
- Larger working set
- Thread blocking needed

Wavefront



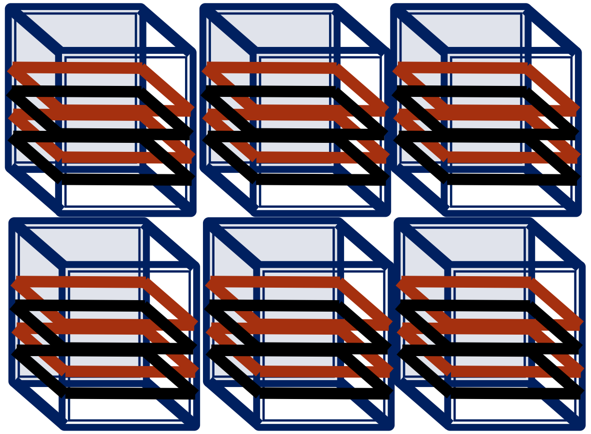
4 stencil sweeps, high DRAM traffic



- Single stencil sweep
- Larger working set
- Thread blocking needed

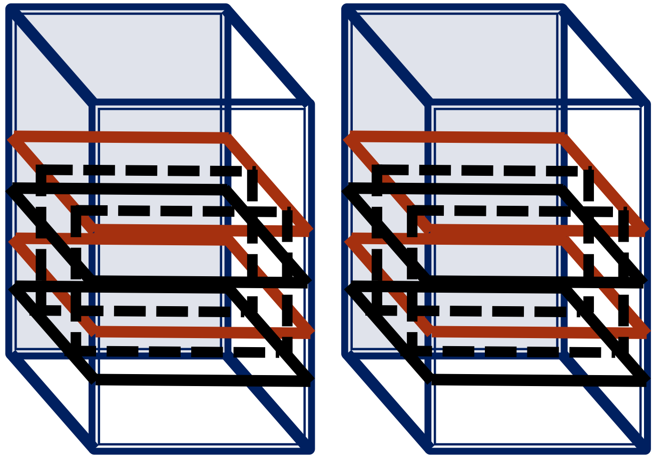
Wavefront vs. Fusion depends on box size and machine!

Inter-Box Parallelism
Thread Configuration <6,1>

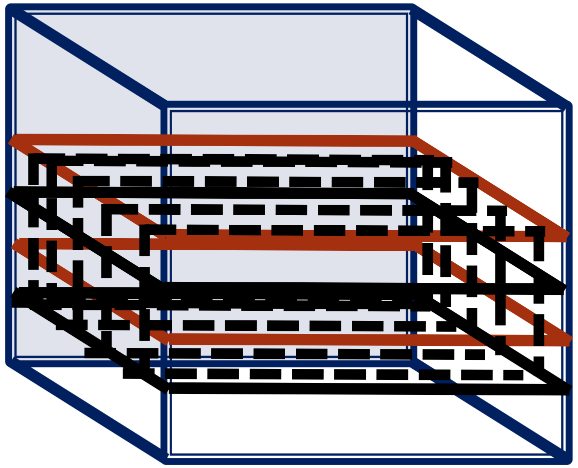


Parallel Decomposition

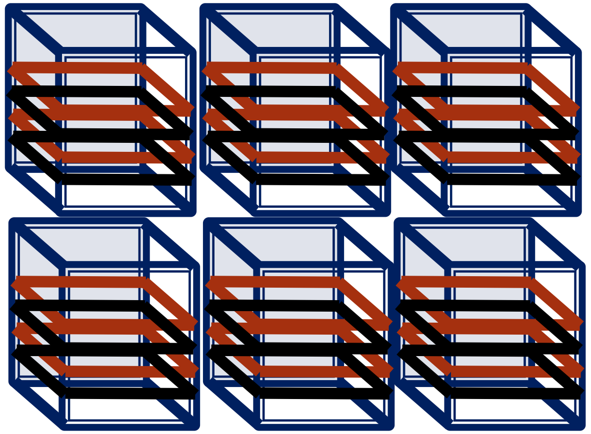
Nested Parallelism
Thread Configuration <2,3>



Intra-Box Parallelism
Thread Configuration <1,6>



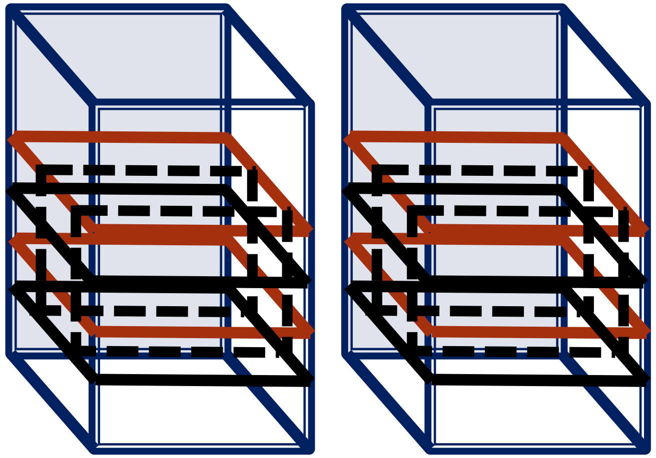
Inter-Box Parallelism
Thread Configuration <6,1>



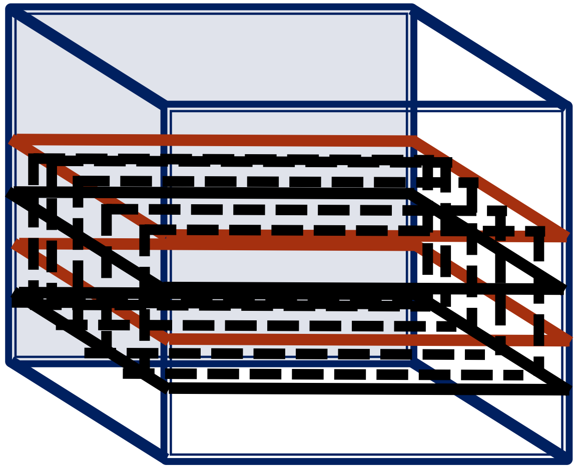
Best parallel code generation strategy depends on box size and machine!

Parallel Decomposition

Nested Parallelism
Thread Configuration <2,3>



Intra-Box Parallelism
Thread Configuration <1,6>



Wavefront and Parallel Code Generation

```
#pragma omp parallel private (...) num_threads(y)
{
   tid=omp_get_thread_num();
   for (k = -3; k <= 66; k++) {
       for (t = 0; t <= min(3,intFloor(t+3,2)); t++) {
          for (j = 6*tid-3; j <= min(6*tid+2,66); j++) {
              for (i= t-3+intMod(-k-color-j-(t-3),2); i<=-t+66; i+=2) {
                  S0(t,k-t,j,i); /* Laplacian */
                  S1(t,k-t,j,i); /* Helhmoltz */
                  S2(t,k-t,j,i); /* GSRB      */
               }
           }
      }
      #pragma omp barrier (or explicit locks)
   }
}
```

Wavefront = skew + permute

OpenMP parallel code generation

- The j-loop is tiled
- Each tile is assigned a thread
- Use spin locks or OMP BARRIER

Wavefront and Parallel Code Generation

```
#pragma omp parallel private (...) num_threads(y)
{
  tid=omp_get_thread_num();
  for (k = -3; k <= 66; k++) {
     for (t = 0; t <= min(3,intFloor(t+3,2)); t++) {
        for (j = 6*tid-3; j <= min(6*tid+2,66); j++) {
           for (i= t-3+intMod(-k-color-j-(t-3),2); i<=-t+66; i+=2) {
              S0(t,k-t,j,i); /* Laplacian */
              S1(t,k-t,j,i); /* Helhmoltz */
              S2(t,k-t,j,i); /* GSRB      */
           }
        }
     }
     #pragma omp barrier (or explicit locks)
  }
}
```

Wavefront = skew + permute

OpenMP parallel code generation

- The j-loop is tiled
- Each tile is assigned a thread
- Use spin locks or OMP BARRIER

Wavefront and Parallel Code Generation

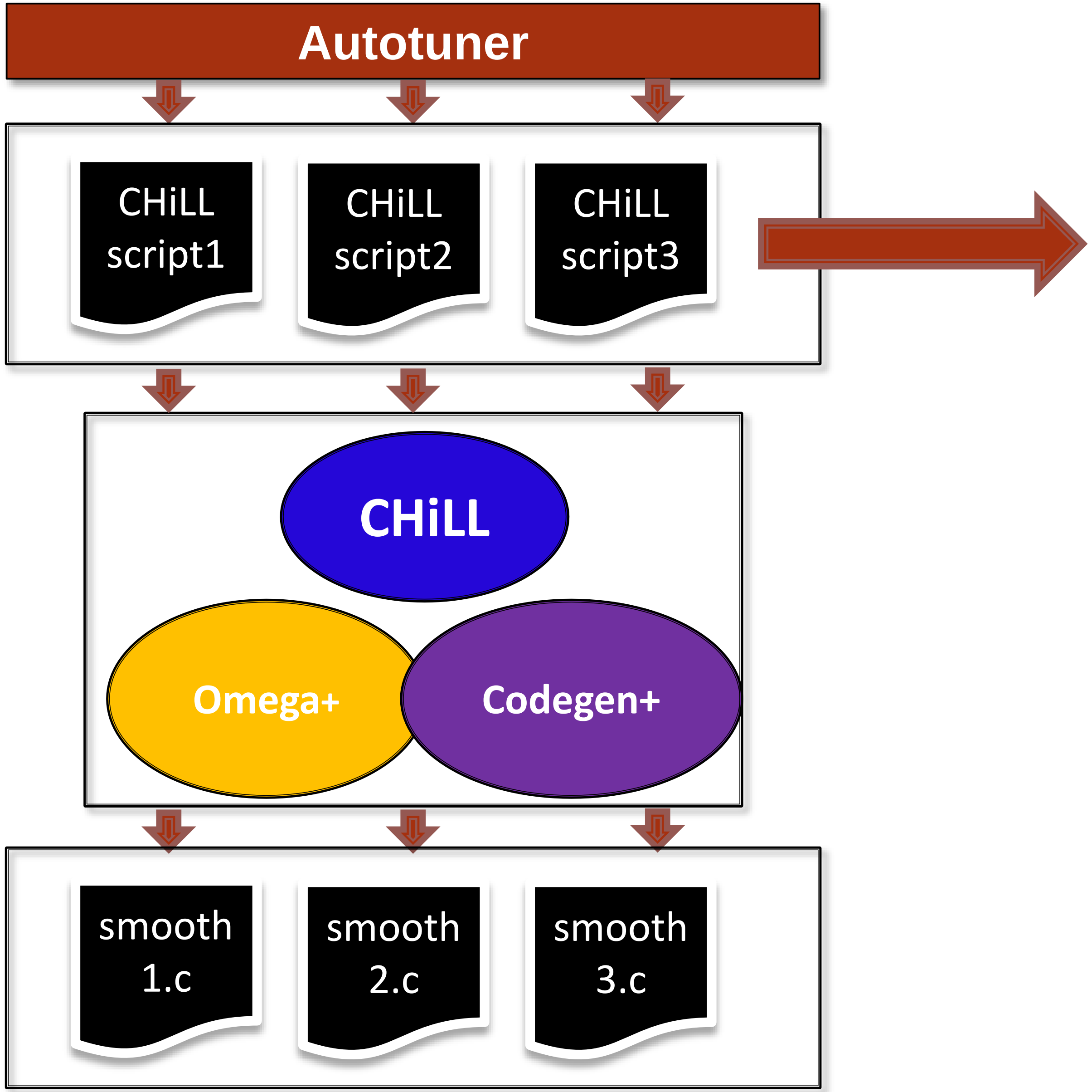
```
#pragma omp parallel private (...) num_threads(y)
{
  tid=omp_get_thread_num();
  for (k = -3; k <= 66; k++) {
     for (t = 0; t <= min(3,intFloor(t+3,2)); t++) {
        for (j = 6*tid-3; j <= min(6*tid+2,66); j++) {
           for (i= t-3+intMod(-k-color-j-(t-3),2); i<=-t+66; i+=2) {
              S0(t,k-t,j,i); /* Laplacian */
              S1(t,k-t,j,i); /* Helhmoltz */
              S2(t,k-t,j,i); /* GSRB      */
           }
        }
     }
     #pragma omp barrier (or explicit locks)
  }
}
```

Wavefront = skew + permute

OpenMP parallel code generation

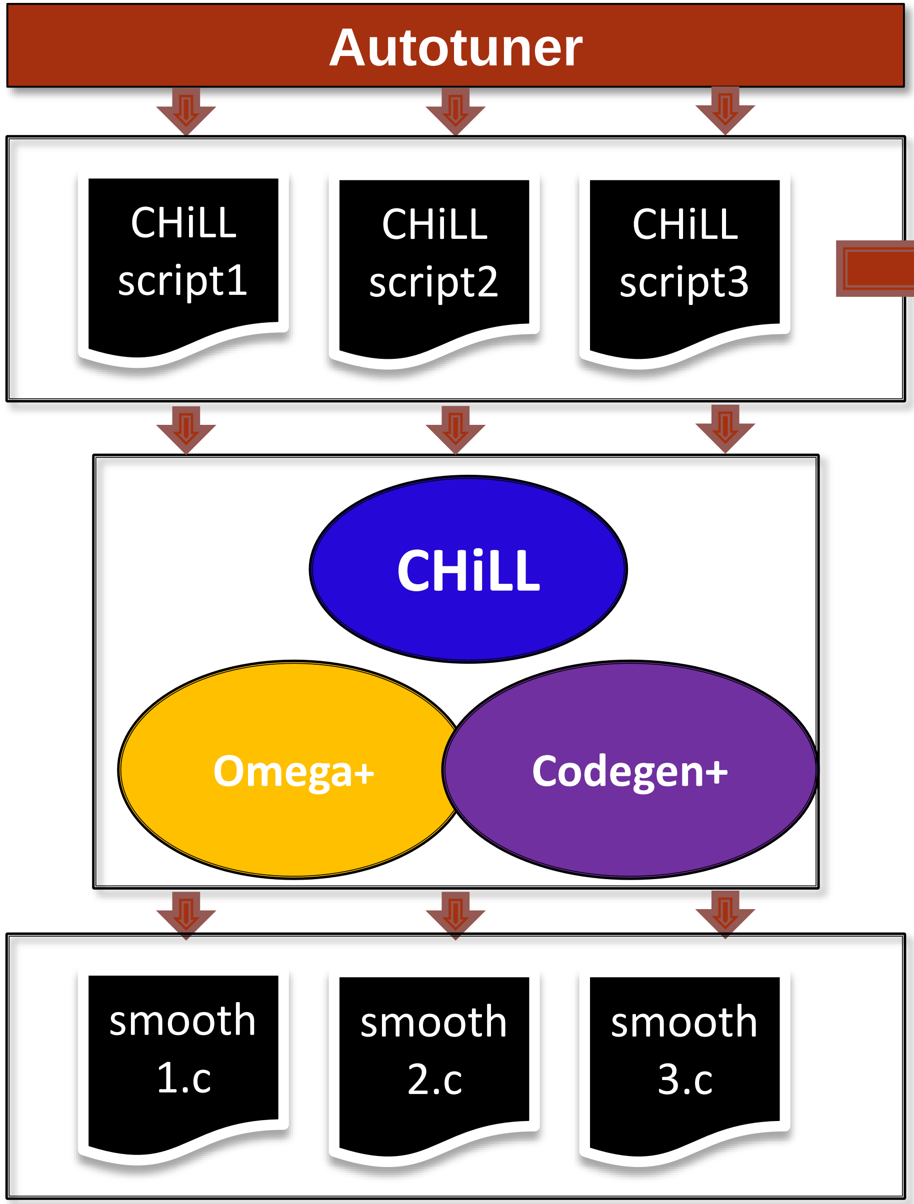
- The j-loop is tiled
- Each tile is assigned a thread
- Use spin locks or OMP BARRIER

Experimental Methodology




```
known (d = 4) #d sets ghost zone
original()
skew ([0,1,2], 2, [2,1])
permute ([2,1,3,4])
tile(s0,3,TJ,2,counted)
gen_omp_parallel_region (locks, y)
```

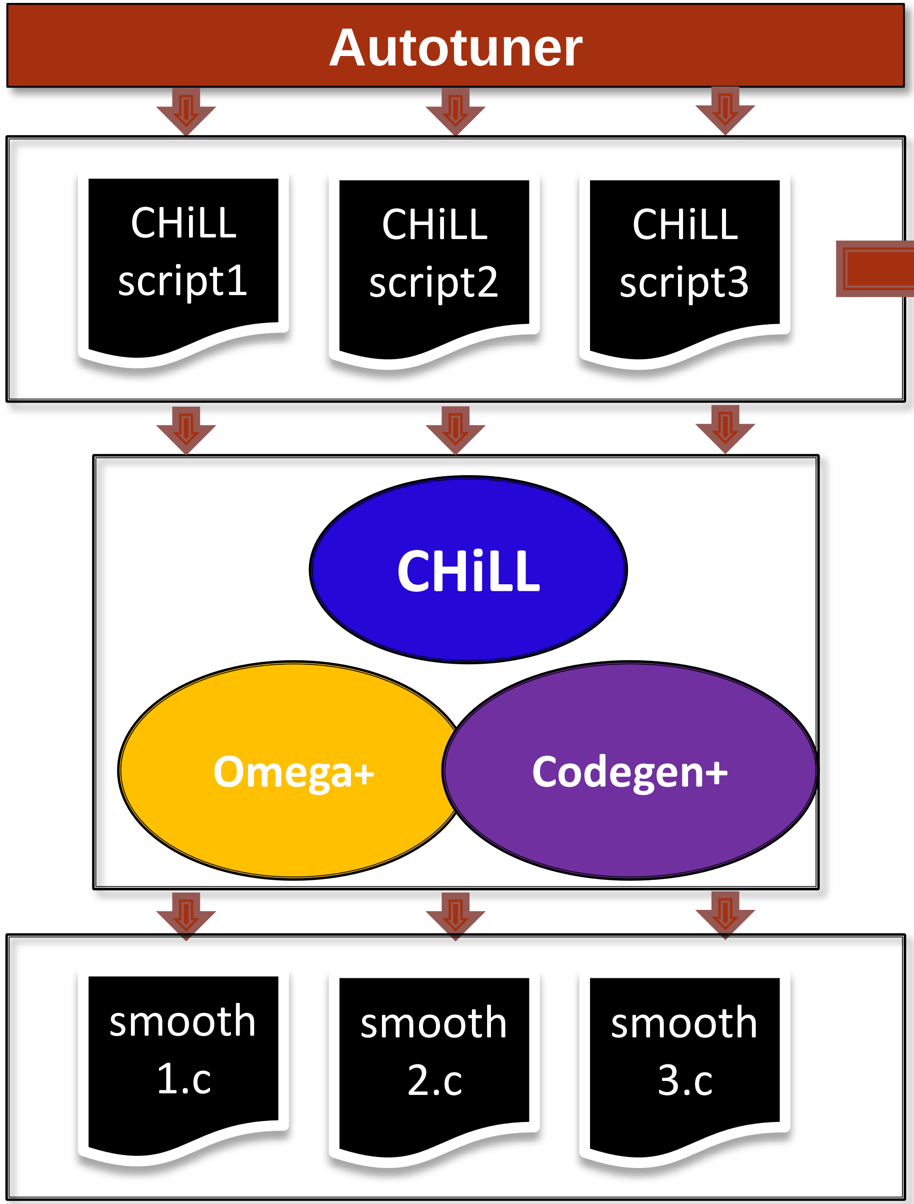
Experimental Methodology




```
known (d = 4) #d sets ghost zone
original()
skew ([0,1,2], 2, [2,1])
permute ([2,1,3,4])
tile(s0,3,TJ,2,counted)
gen_omp_parallel_region (locks, y)
```

Sets ghost zone depth

Experimental Methodology

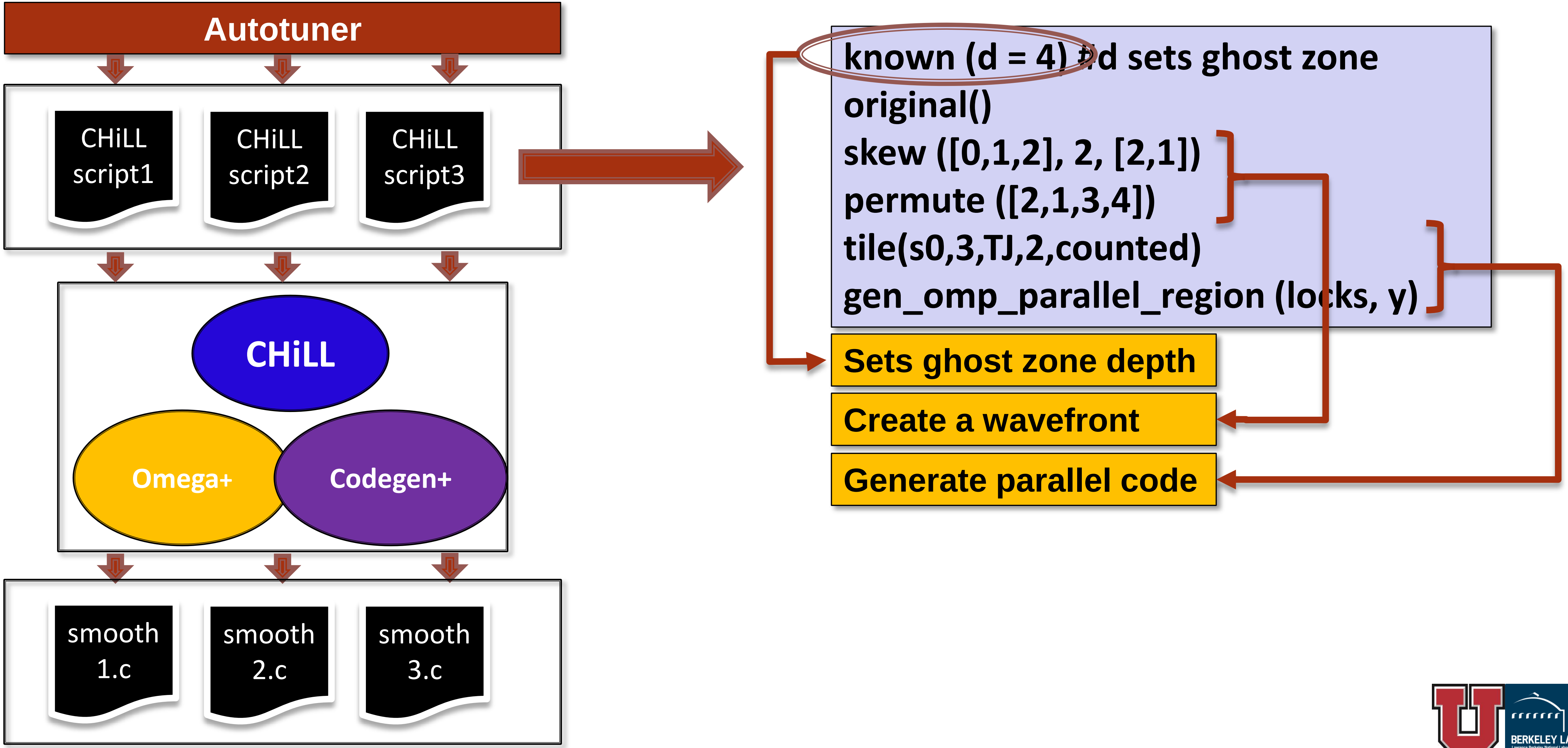



```
known (d = 4) #d sets ghost zone
original()
skew ([0,1,2], 2, [2,1])
permute ([2,1,3,4])
tile(s0,3,TJ,2,counted)
gen_omp_parallel_region (locks, y)
```

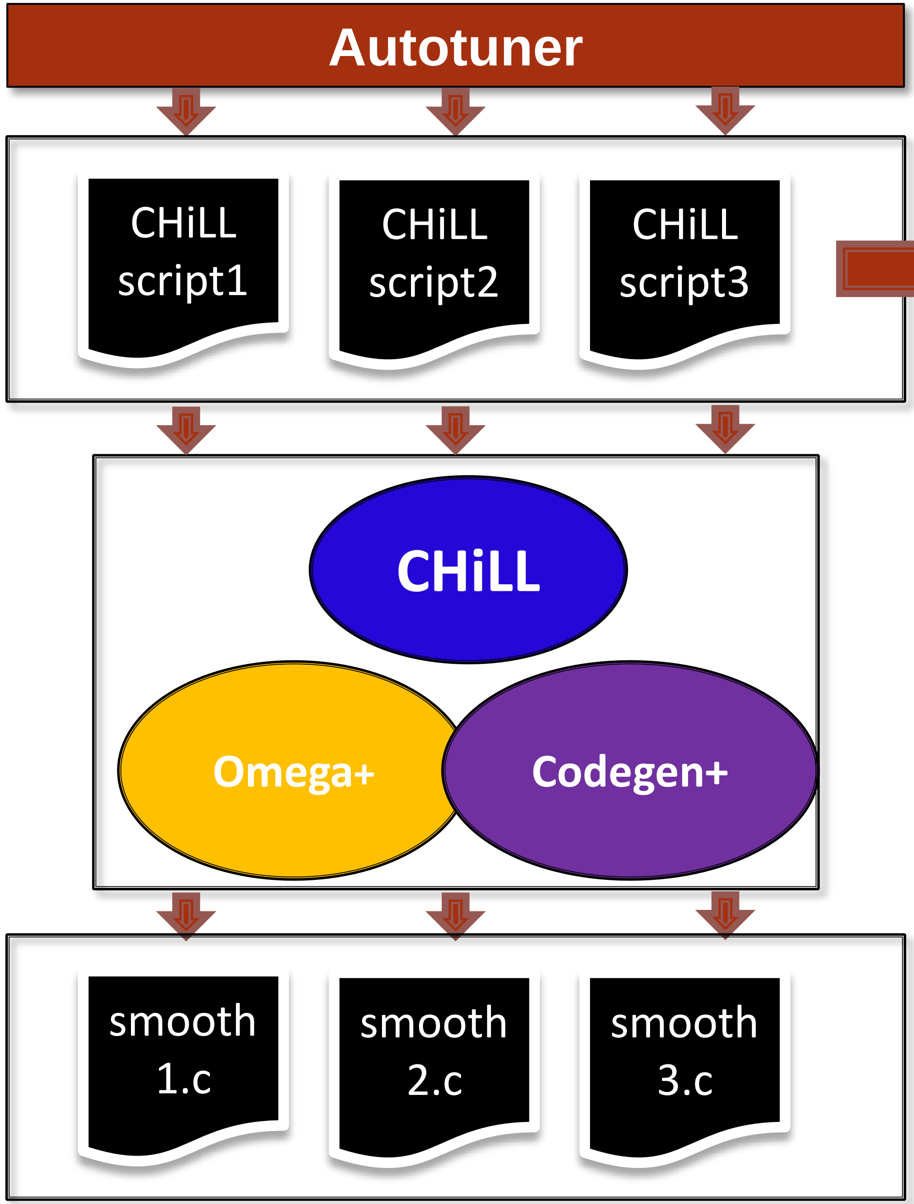
Sets ghost zone depth

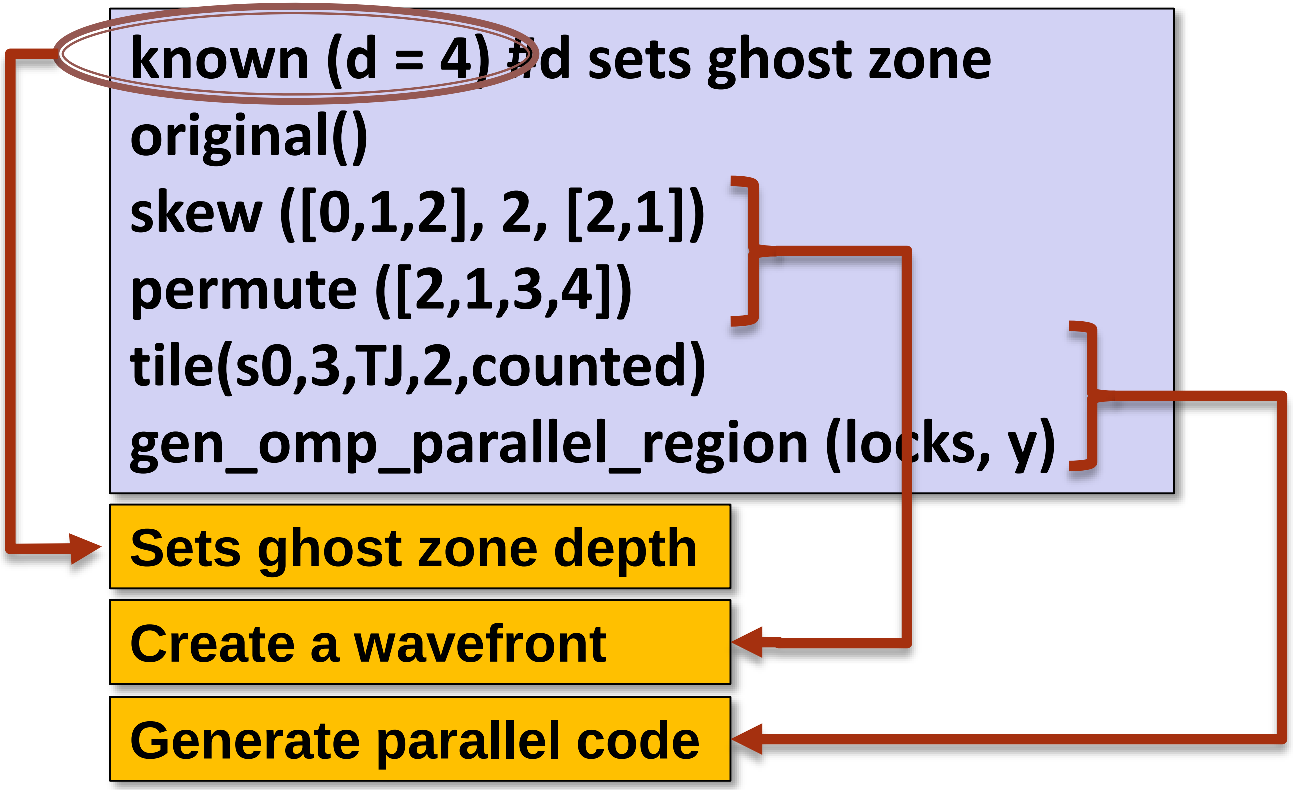
Create a wavefront

Experimental Methodology



Experimental Methodology





For each level of the V-cycle, our search space is:

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

Experimental Methodology

Problem configuration

- 256^3 problem size (domain), decomposed to 64^3 boxes
- Variable coefficient 3D 7-point Gauss-Seidel red-black smooth operator
- Periodic boundary conditions
- V-cycle run 10 times

Target Architectures

Code was run on a single node on two NERSC machines. Hopper, a Cray XE6 and the new Cray XC30 Edison Phase II.

Hopper	Edison (Phase II)
AMD Opteron Cores	Intel Sandy Bridge Cores
4 chips per node, 6 cores per chip	2 chips per node, 12 cores per chip

Experimental Methodology

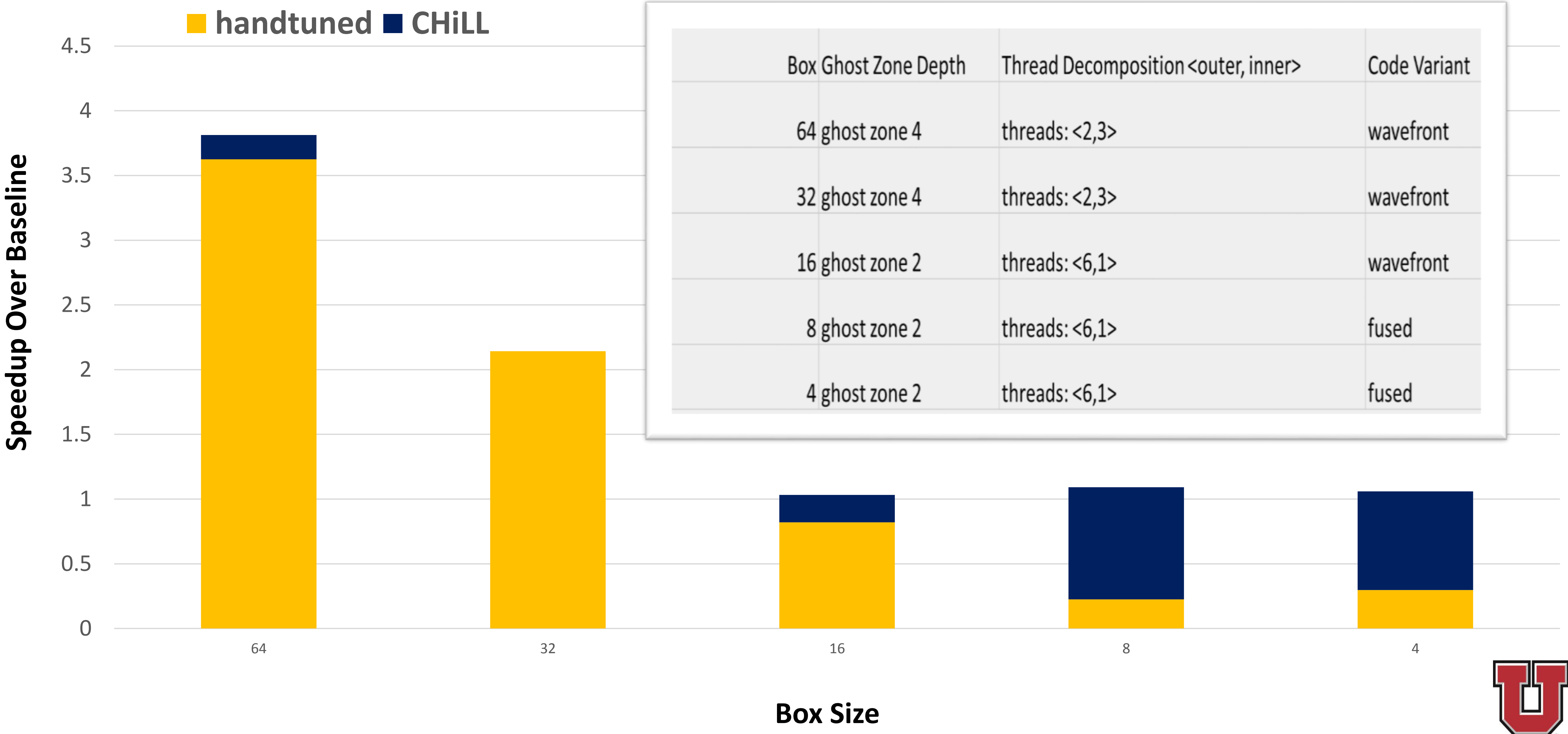
Baseline Code

- Operators (Laplacian, Helmholtz, GSRB) not fused
- Ghost zone depth is one
- Inter-box thread decomposition

Hand tuned Code : miniGMG

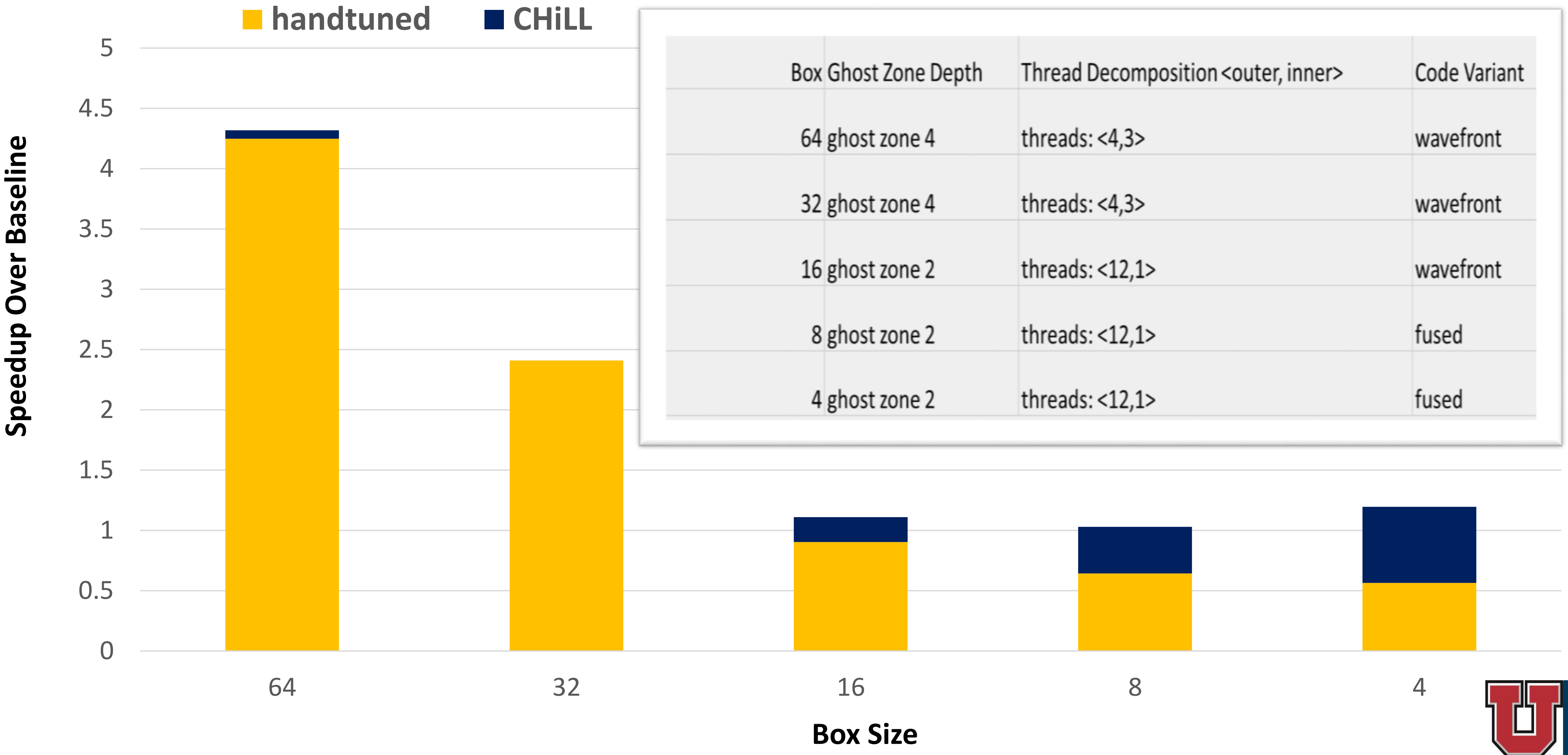
High performance code from Samuel Williams et al. Supercomputing’12 paper : “Optimization of Geometric Multigrid for Emerging Multi- and Many core Processors”

Performance of Smooth on Hopper



Box	Ghost Zone Depth	Thread Decomposition <outer, inner>	Code Variant
64	ghost zone 4	threads: <2,3>	wavefront
32	ghost zone 4	threads: <2,3>	wavefront
16	ghost zone 2	threads: <6,1>	wavefront
8	ghost zone 2	threads: <6,1>	fused
4	ghost zone 2	threads: <6,1>	fused

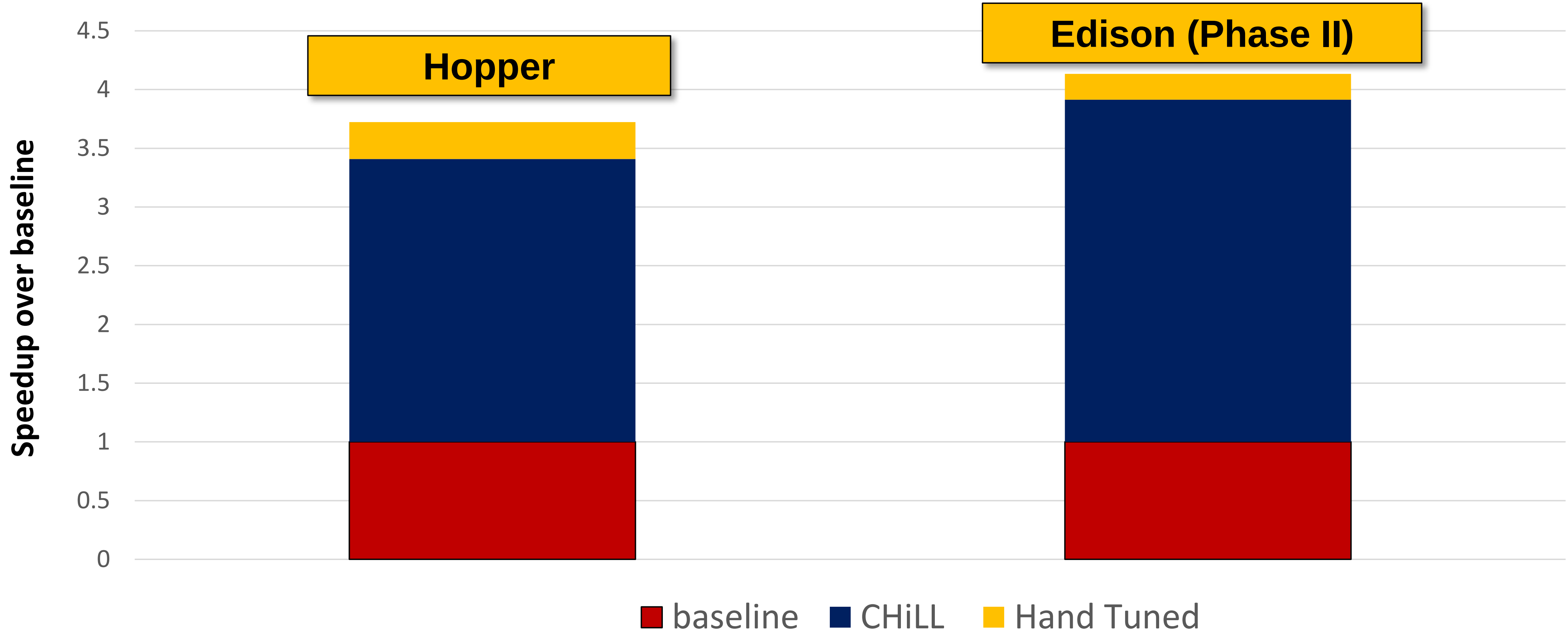
Performance of Smooth on Edison (Phase II)



Box	Ghost Zone Depth	Thread Decomposition <outer, inner>	Code Variant
64	ghost zone 4	threads: <4,3>	wavefront
32	ghost zone 4	threads: <4,3>	wavefront
16	ghost zone 2	threads: <12,1>	wavefront
8	ghost zone 2	threads: <12,1>	fused
4	ghost zone 2	threads: <12,1>	fused

Overall Speedup for Box Size 64

Total Time = Smooth + Residual + Restriction



Summary and Conclusion

Need For Autotuning And Compiler Support

At each level of the V-cycle we *need* to autotune for :

- Ghost zone depth
- Create a wavefront computation or use simple fused loops
- Thread decomposition

Higher Performance From Searching a Rich Space of Variants

Generated code betters hand tuned code without

- Software prefetching
- SSE/AVX code

Compiler-generated code variant has different threading configuration than manually tuned code!

Questions?

Extra

We use a double-precision, finite volume discretization of the variable-coefficient operator $L = a\vec{\alpha}I - b\nabla\vec{\beta}\nabla$ with periodic boundary conditions as the linear operator within our test problem. Variable-coefficient is an essential (yet particularly challenging) facet as most real-world applications demand it. The right-hand side (f) is $sin(2\pi x)sin(2\pi y)sin(2\pi z)$ on the [0,1] cubical domain. The u, f, and $\vec{\alpha}$ are cell-centered data, while the $\vec{\beta}$'s are face-centered.