Experiences of Using the OpenMP Accelerator Model to Port DOE Stencil Applications

Pei-Hung Lin, Chunhua “Leo” Liao, Dan Quinlan, Stephen Guzik*

IWOMP’15 , Oct. 1st, 2015

Lawrence Livermore National Laboratory

* Colorado State University

LLNL-PRES-664299

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

Outline

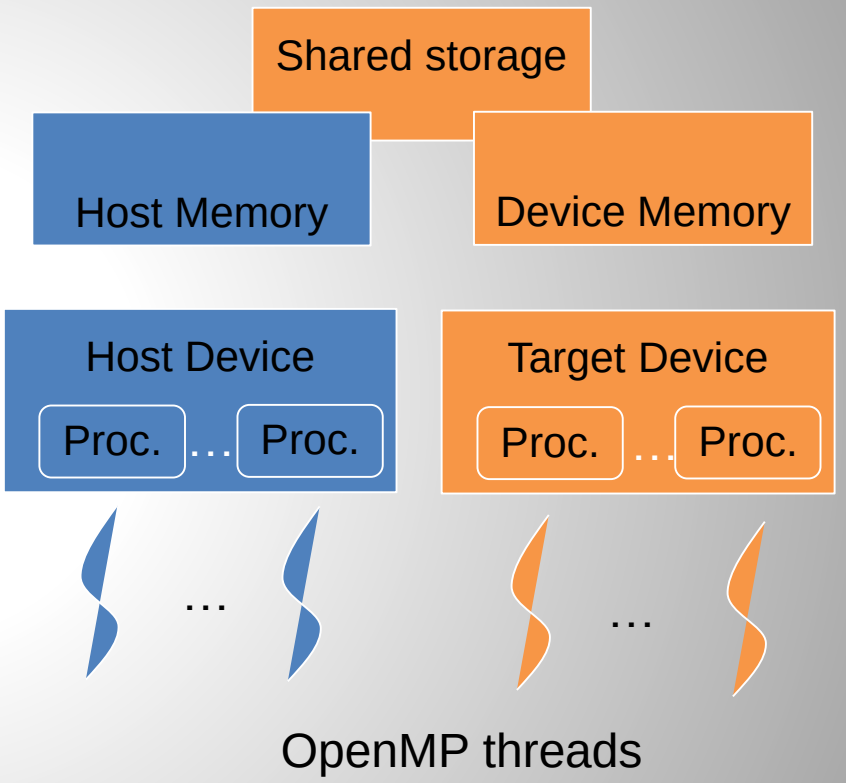
- Motivation
- Background
  - OpenMP 4.0 and HOMP
  - Stencil Mini-Apps
- Methodology of Porting
- Performance
- Discussions

Motivation

- Test improving productivity of programming accelerators through high-level programming model
- Evaluate the process to port non-trivial scientific stencil applications using the OpenMP accelerator model
- Discover challenges and suggest solutions or workarounds

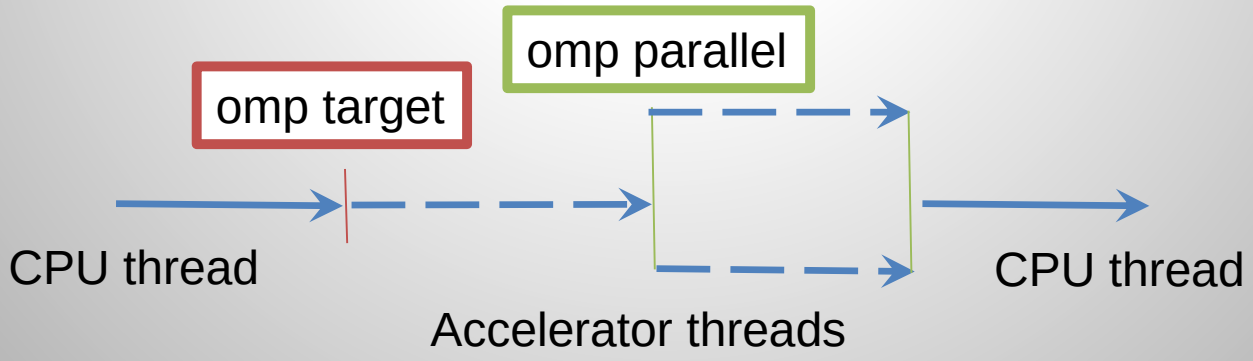
OpenMP 4.0's accelerator model

- Device: a logical execution engine
  - Host device: where OpenMP program begins, one only
  - Target devices: **1 or more** accelerators
- Memory model
  - Host data environment: one
  - Device data environment: one or more
  - Allow shared host and device memory
- Execution model: Host-centric
  - Host device : "offloads" code regions and data to accelerators/target devices
  - Target Devices: still fork-join model
  - Host waits until devices finish
  - Host executes device regions if no accelerators are available /supported



Computation and data offloading

- Directive: #pragma omp target *device(id) map() if()*
  - **target**: create a device data environment and offload computation to be run in sequential on the same device
  - **device (int_exp)**: specify a target device
  - **map(to|from|tofrom|alloc:var_list)** : data mapping between the current task's data environment and a device data environment
- Directive: #pragma omp target data *device(id) map() if()*
  - Create a device data environment



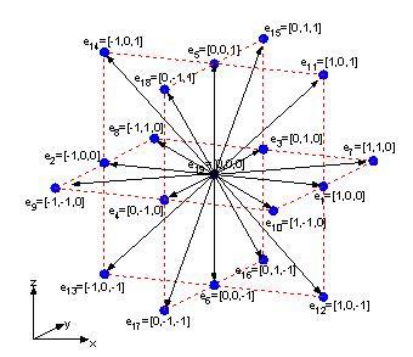
A prototype Implementation: HOMP

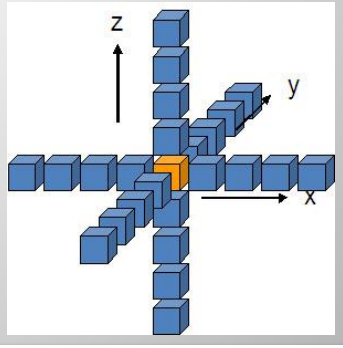
- HOMP: Heterogeneous OpenMP
  - A source-to-source approach
    - Input: OpenMP programs with accelerator directives
    - Output: CUDA
    - Initial OpenMP Accelerator Model support, Liao IWOMP' 13
    - Multiple accelerator extensions, Yan PMAM'15
    - Ongoing work for MPI code generation
  - Compiler
    - Built on top of ROSE source-source compiler framework at LLNL
    - Code: loop transformation, outliner for CUDA kernel generation
    - Variables: data environments, array linearization, reduction
  - Runtime
    - Device probe, kernel configuration & launch
    - Loop scheduling, data handling, reduction, …
    - C bindings: interoperate with C/C++ and Fortran

Stencil computation and mini-apps

Application	Lattice-Boltzmann (LB)	Compressible Navier-Stokes (CNS)
Math formula	Hydrodynamics using a discrete kinetic equation	Finite-difference methods
Programming languages	C++, MPI, OpenMP	Fortran, MPI, OpenMP
AMR Library	Chombo	BoxLib
Stencil Property	3D 1-point stencil	3D 25-point stencil
# line of code	4670 (12879 w/Chombo code)	1242 (25967 w/ BoxLib code)
Author/source	Stephen Guzik @Colorado State Univ.	miniApp in BoxLib @LBL

Stencil computation: common in simulation codes, e.g. for computational fluid dynamics, for solving partial differential equations, the Jacobi kernel, image processing etc.





Kernels of two stencil Mini-Apps

```
1  fi(cells, 19, boxes) = initial data;
2  fiUpdate(cells, 19, boxes) = 0;
3  U(grid, 4, boxes);
4  Macroscopic(U, fi);
5  for (int iTS = 0; iTS != nTimeStep; ++iTS)
6  {
7    int iBox;
8    for (every box)
9    {
10     {  // Advance function
11       for (every cell)
12         Collision(fi, U);
13       Exchange(fi);
14       BC(fi);
15       Stream(fiUpdate, fi);
16       for (every cell)
17         Macroscopic(U, fiUpdate);
18       swap(fi, fiUpdate);
19     }
20   }
21 }
```

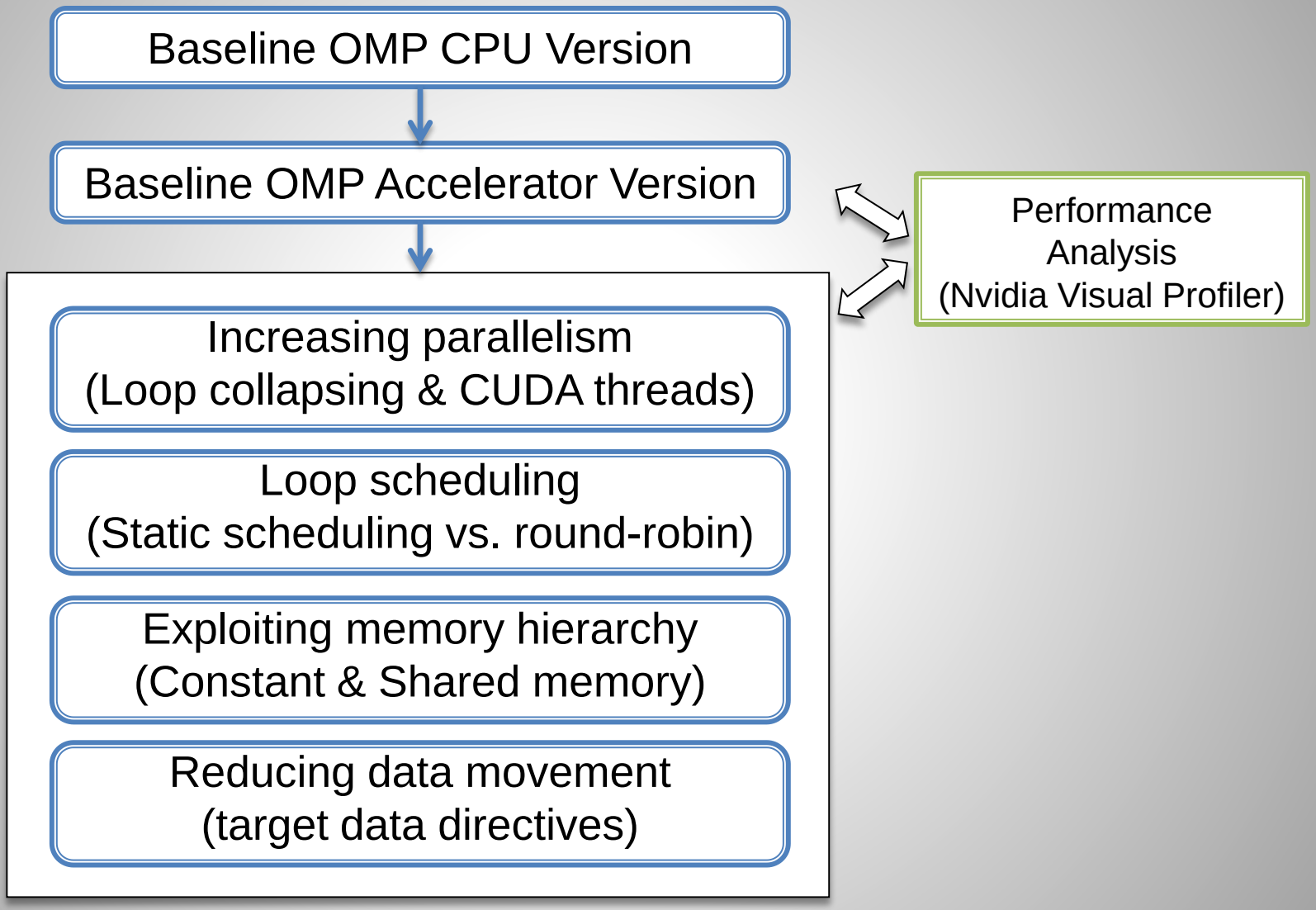
LB

```
1  init_data(U, dx, prob_lo, prob_hi)
2  for (int iTS = 0; iTS != nTimeStep; ++iTS)
3  {
4    int iFab;
5    for (every Fab)
6      {  // Advance function
7        for (1/3 timestep)
8          {  // Advance 1/3 of timestep in each iter.
9            for (every grid cell)
10             ctoprim(Unew, Q);
11           for (every grid cell)
12             diffterm(Q, D);
13           for (every grid cell)
14             hypterm(Q, F);
15         }
16     }
17 }
```

CNS

Similar algorithm: iterative, space/data decomposition, advance functions
Parallelism at multiple levels: e.g. every box/fab in a grid, or for every cell/points in a box, or for every dimension, etc.

Methodology



Baseline OMP accelerator version

```
#pragma omp parallel for \
private(i,j,k,unp1,unp2,unp3,unp4,unm1,unm2,unm3,unm4)
  for (k = kFluxLowerBound; k <= kFluxUpperBound; k++) {
    for (j = jFluxLowerBound; j <= jFluxUpperBound; j++) {
      for (i = iFluxLowerBound; i <= iFluxUpperBound; i++) {
        unp1 = q[qu-1][k][j][i+1];
…}}}
```

```
#pragma omp target device (gpu0) map(in:qu, dxinv[0:3], irho, iFluxLowerBound,
jFluxLowerBound, kFluxLowerBound, iFluxUpperBound, jFluxUpperBound,
kFluxUpperBound, iSize,jSize,kSize,iFluxSize,jFluxSize, kFluxSize, q[0:q_ub],
cons[0:cons_ub], … ) map(inout:flux[0:flux_ub])
#pragma omp parallel for private(i,j,k,unp1,unp2,unp3,unp4,unm1,unm2,unm3,unm4)
  for (k = kFluxLowerBound; k <= kFluxUpperBound; k++) {
    for (j = jFluxLowerBound; j <= jFluxUpperBound; j++) {
      for (i = iFluxLowerBound; i <= iFluxUpperBound; i++) {
        unp1 = q[qu-1][k][j][i+1];
…}}}
```

From OMP CPU version
To OMP ACC version

- **e.g. hypterm() function**

in CNS

Performance:

- **Less than 2% GPU occupancy**
- **Small loop iteration count**

Increasing parallelism (collapsing)

//Input C code:

```
#pragma omp target device (gpu0) map(in:qu, dxinv[0:3], irho, imx, imy, imz, qpres,iene,  ALP, BET, GAM,
DEL,iLowerBound,jLowerBound,kLowerBound,iFluxLowerBound,jFluxLowerBound,kFluxLowerBound,iFluxUpperBound,jFluxUpperBound,kFluxUpperBound,i
Size,jSize,kSize,iFluxSize,jFluxSize, kFluxSize, q[0:q_ub], cons[0:cons_ub] ) map(inout:flux[0:flux_ub])
#pragma omp parallel for private(i,j,k,unp1,unp2,unp3,unp4,unm1,unm2,unm3,unm4) collapse(3)
  for (k = kFluxLowerBound; k <= kFluxUpperBound; k++) {
    for (j = jFluxLowerBound; j <= jFluxUpperBound; j++) {
      for (i = iFluxLowerBound; i <= iFluxUpperBound; i++) {
        unp1 = q[qu-1][k][j][i+1];
…}}}
```

Loop collapse

// Output CUDA code:

```
__global__ void OUT__8__4081__(int irho,…,int iFluxLowerBound,int jFluxLowerBound,int kFluxLowerBound,int iFluxSize,int jFluxSize,int kFluxSize,int
iLowerBound,int jLowerBound,int kLowerBound,int iSize,int jSize,int kSize,int final_total_iters__expression_0x33906b0,int
k_interval__expression_0x33908e0,int j_interval__expression_0x3390a30,double *_dev_cons,double *_dev_q,double *_dev_flux,double *_dev_dxinv)
{…
XOMP_static_sched_init(0,final_total_iters__expression_0x33906b0 -
1,1,1,_dev_thread_num,_dev_thread_id,&_dev_loop_chunk_size,&_dev_loop_sched_index,&_dev_loop_stride
  while(XOMP_static_sched_next(&_dev_loop_sched_index,final_total_iters__expression_0x33906b0 -
1,1,_dev_loop_stride,_dev_loop_chunk_size,_dev_thread_num,_dev_thread_id,&_dev_lower,&_dev_upper
    for (_dev_i = _dev_lower; _dev_i <= _dev_upper; _dev_i += 1) {
    _p_k = _dev_i / k_interval__expression_0x33908e0 * 1 + kFluxLowerBound;
        int k_remainder = _dev_i % k_interval__expression_0x33908e0;
    _p_j = k_remainder / j_interval__expression_0x3390a30 * 1 + jFluxLowerBound;
    _p_i = k_remainder % j_interval__expression_0x3390a30 * 1 + iFluxLowerBound;
    _p_unp1 = _dev_q[_p_i - iLowerBound + iSize * (_p_j - jLowerBound + jSize * (_p_k + 1 - kLowerBound + kSize * (qw - 1)))];
…}}}
```

Extra index computation

Increasing parallelism (Multidimensional thread structure)

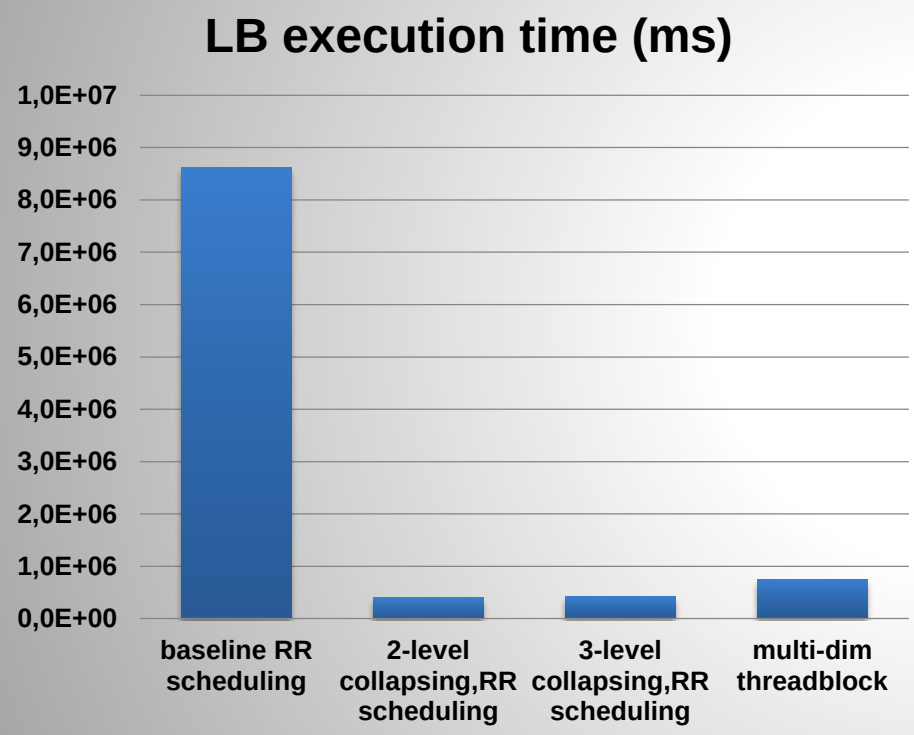
- Continue with same loop in the previous example
- Map to different grid, thread block configurations:
  - One example: two outer loops: mapped to a 2-D grid of 1-D thread blocks, innermost loop: mapped to the 1-D thread block
  - Autotuning opportunity

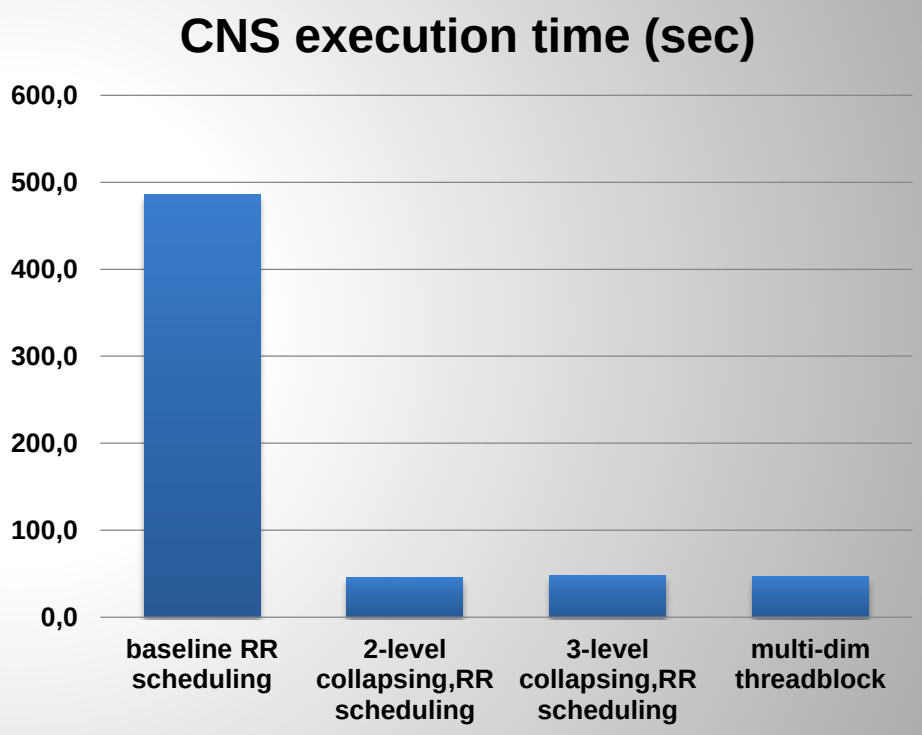
```
dim3 _num_blocks_((hi[1] - lo[1]+ 2 * ng + 1),(hi[2] - lo[2]+ 2 * ng + 1));  // 2-D grid of blocks to map outer 2 loops

dim3 _threads_per_block_((hi[0] - lo[0] +  2 * ng + 1));  // innermost loop mapped to a thread block

OUT__11__4611__<<<_num_blocks_,_threads_per_block_>>>(CVinv,GAMMA,ng,iLowerBound,jLowerBound,kLowerBound,iSize,jSize,kSize,_dev_lo,_dev_hi,_dev_u,_dev_q);
```

Compared to using the default 1-D thread block structure

```
int _num_blocks_ = xomp_get_max1DBlock(hi[2] + 2 * ng - lo[2] + 1);
int _threads_per_block_ = xomp_get_maxThreadsPerBlock();
OUT__11__4611__<<<_num_blocks_,_threads_per_block_>>>(CVinv,GAMMA,ng,iLowerBound,jLowerBound,kLowerBound,iSize,jSize,kSize,_dev_lo,_dev_hi,_dev_u,_dev_q);
```

Effects of increasing parallelism





- RR: loop scheduling policy is Round-Robin scheduling
- More collapsing, more computation overhead for index calculation

Exploiting memory hierarchy :constant memory in LB

Example from LBLevel function in LB

```
LBLevel::LBLevel(…)
  :  … m_bodyForce(a_bodyForce),  m_tau(a_tau),  …
{
m_fi[0].define(m_boxes, LBParameters::g_numVelDir, LBParameters::g_numGhost);
m_fi[1].define(m_boxes, LBParameters::g_numVelDir, LBParameters::g_numGhost);
m_U.define(m_boxes, LBParameters::g_numState, 0);
Const int nBox = m_boxes.localSize();
#pragma omp target data cache (const m_fi[0](0:nBox), const m_f1[1](-:nBox ), const m_U(0:nBox ),
const m_tau, const m_bodyForce)
  {
    // loop body omitted  …
  }
}
```

LB uses many read-only arrays for storing co-efficiencies, stride distances, weights, etc.

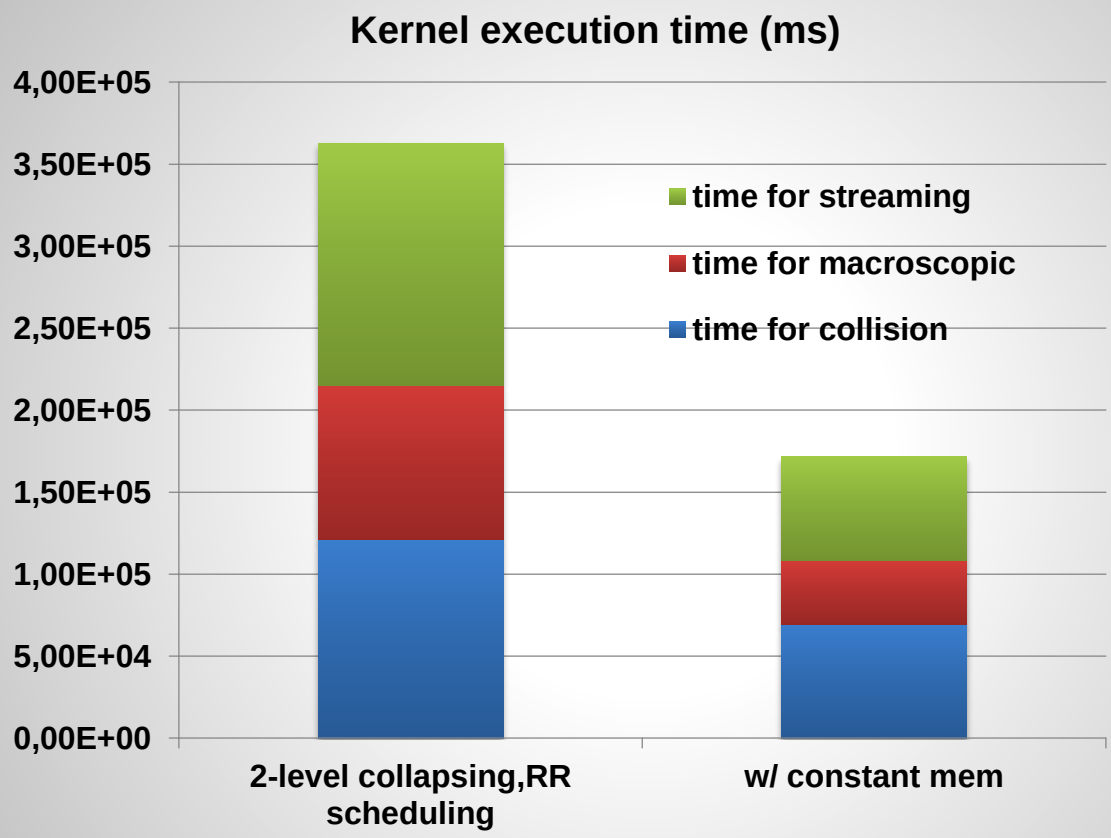
Exploiting memory hierarchy (constant memory in LB)

```
// Generated host code for the LBLevel function in LB
__constant__ Real** c_fiFabPtrs[2]
 __constant__ Real** c_UFabPtrs;
__constant__ Real c_tau;
__constant__ Real c_bodyForce;
…
cudaMemcpyToSymbol(c_fiFabPtrs, &(a_fi0DeviceFabPtrs.device), sizeof(AccelSymbol));
cudaMemcpyToSymbol(c_fiFabPtrs, &(a_fi1DeviceFabPtrs.device), sizeof(AccelSymbol), sizeof(AccelSymbol));
cudaMemcpyToSymbol(c_UFabPtrs, &(a_UDeviceFabPtrs.device), sizeof(AccelSymbol));
cudaMemcpyToSymbol(c_tau, &a_tau, sizeof(Real));
cudaMemcpyToSymbol(c_bodyForce, &a_bodyForce, sizeof(Real));
….
```

Copy read only array into constant memory

Copy constant coefficients into constant memory

Performance w/ constant memory (LB)



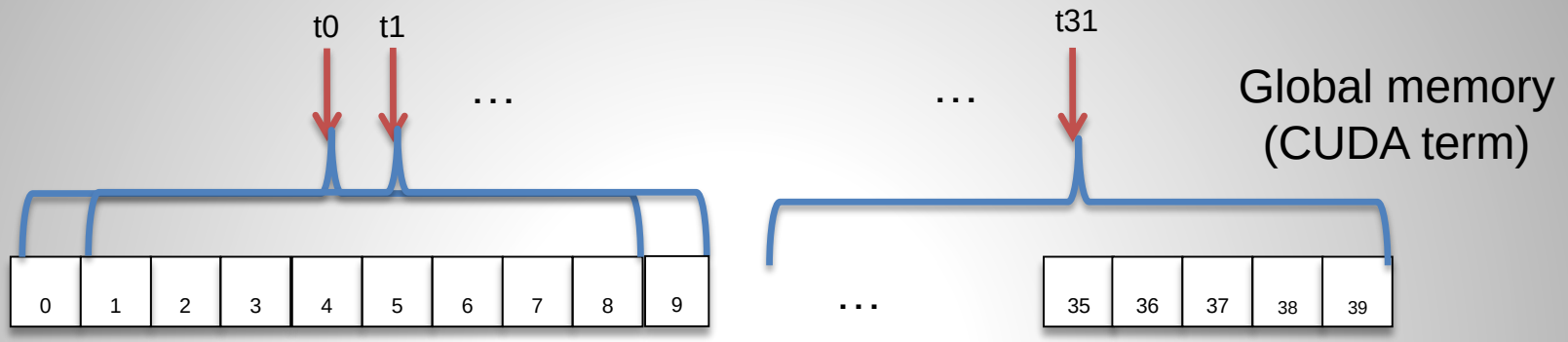
Speedup of three kernels: 1.32x, 1,74x, and 2.44x

Memory hierarchy optimization: data reuse in CNS

Example from hypterm function in CNS

```
#pragma omp target device (gpu0) map(in:qu, dxinv[0:3], ALP, BET, GAM, \
iFluxLowerBound, jFluxLowerBound, kFluxLowerBound, iFluxUpperBound, \
jFluxUpperBound, kFluxUpperBound,…) map(inout:flux[0:flux_ub]) \
cache(cons[0:cons_ub] )
#pragma omp parallel for private(i,j,k,unp1,…)
  for (k = kFluxLowerBound; k <= kFluxUpperBound; k++) {
    for (j = jFluxLowerBound; j <= jFluxUpperBound; j++) {
      for (i = iFluxLowerBound; i <= iFluxUpperBound; i++) {
   …
  // 8+1 points stencil on each split dimension, 8x3+1= 25-point stencil for 3-D
      Flux[rho][k][j][i]= -((  ALP * (cons[imx][k][j][i-1]- cons[imx][k][j][i+1]) +
                               BET * (cons[imx][k][j][i-2] - cons[imx][k][j][i+2]) +
                               GAM * (cons[imx][k][j][i-3] - cons[imx][k][j][i+3]) +
                               DEL * (cons[imx][k][j][i+3] - cons[imx][k][j][i-3] ) + …
                             ) * dxinv[0]);
…  }}}
```

Exploiting shared memory for CNS

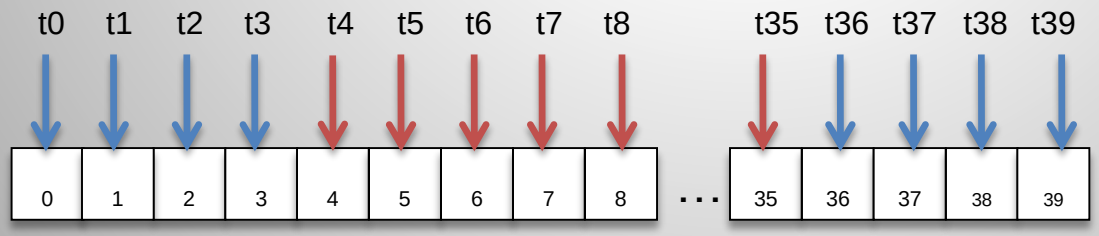


9-point stencil in 1D: every GPU thread loads 9 contiguous elements from global memory for computation.

All contiguous elements can be stored in shared memory and shared by all threads in a thread block.

Two choices:

1. First and last thread load more elements: thread divergence
2. Use more GPU threads to load data into shared memory: partial warp usage.



Exploiting memory hierarchy (shared memory in CNS)

// Generated Host code:

```
dim3 _threads_per_block_((hi[0] + ng - (lo[0] - ng) + 1));
dim3 _num_blocks_((hi[1] - lo[1] + 1),(hi[2] - lo[2] + 1));
OUT__10__4611__<<<_num_blocks_,_threads_per_block_,11*iSize*sizeof(double)>>>(irho,imx,imy,imz,iene,qu,qpres,ALP,BET,GAM,DEL,iFluxLowerBound,jFluxLowerBound,kFluxLowerBound,iFluxUpperBound,jFluxUpperBound,kFluxUpperBound,iFluxSize,jFluxSize,kFluxSize,iLowerBound,jLowerBound,kLowerBound,iSize,jSize,kSize,_dev_cons,_dev_q,_dev_flux,_dev_dxinv);
```

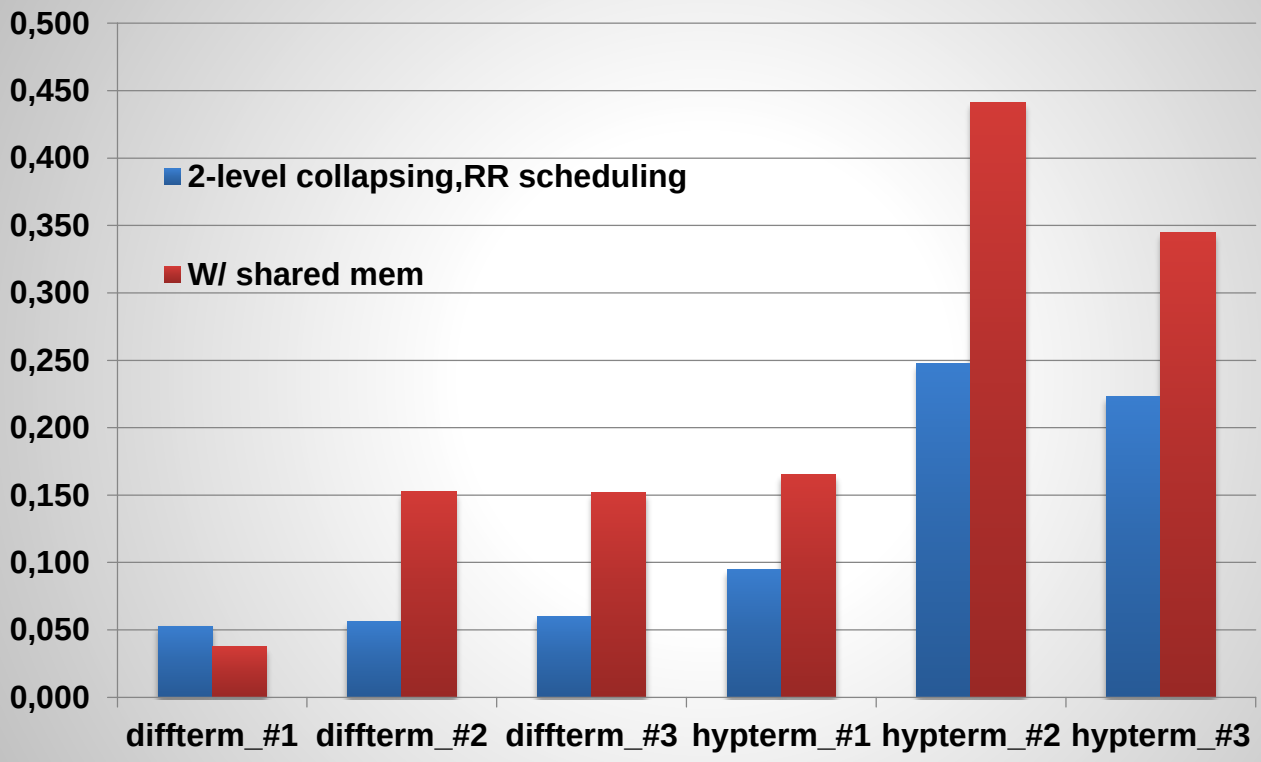
Dynamic Shared Memory

// Generated device code:

```
__global__ void OUT__10__4611__(…)
{…
  extern __shared__ double s[];
  double* s_cons = (double*)s;
  …
for(idx = 0; idx < 5; idx++)
    s_cons[_p_k+idx * kSize] = _dev_q[blockIdx.x + iSize * (blockIdx.y + jSize * (_p_k + kSize * (idx)))];
if(_p_k >= ng && _p_k <= kSize – ng)  {
_dev_flux[_p_i - iFluxLowerBound + iFluxSize * (_p_j - jFluxLowerBound + jFluxSize * (_p_k - kFluxLowerBound + kFluxSize * (irho - 1)))] = _dev_flux[_p_i - iFluxLowerBound + iFluxSize * (_p_j - jFluxLowerBound + jFluxSize * (_p_k - kFluxLowerBound + kFluxSize * (irho - 1)))] - (ALP * (s_cons[ (_p_k + 1  + kSize * (imz - 1))] - s_cons[ (_p_k - 1  + kSize * (imz - 1))]) + BET * (s_cons[ (_p_k + 2  + kSize * (imz - 1))] - s_cons[ (_p_k - 2  + kSize * (imz - 1))]) + GAM * (s_cons[ (_p_k + 3  + kSize * (imz - 1))] - s_cons[ (_p_k - 3  + kSize * (imz - 1))]) + DEL * (s_cons[ (_p_k + 4  + kSize * (imz - 1))] - s_cons[ (_p_k - 4  + kSize * (imz - 1))])) * _dev_dxinv[2];
}
```

Load data elements into shared memory from global memory

Performance w/ shared memory (CNS)



6 loops using shared memory.
Partial warp utilization hurts the GPU occupancy

Reduce data transfers between CPU and GPUs

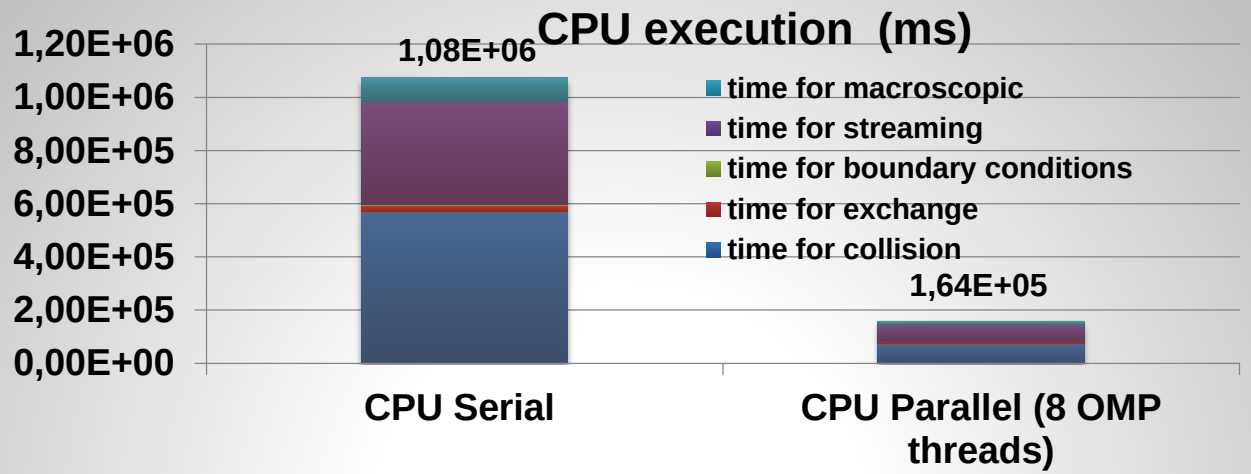
- OpenMP 4.0 restrictions on map variables
  - Must be a complete type
  - Partial variables not allowed unless parts of arrays
  - Cannot contain static data or virtual members
  - Memory blocks pointed by pointers are not mapped
- LB and CNS:
  - Outer loop iterates over multiple Boxes/Fabs.
  - High level data types are exposed, with static data and pointers
  - Significant code rewriting is needed

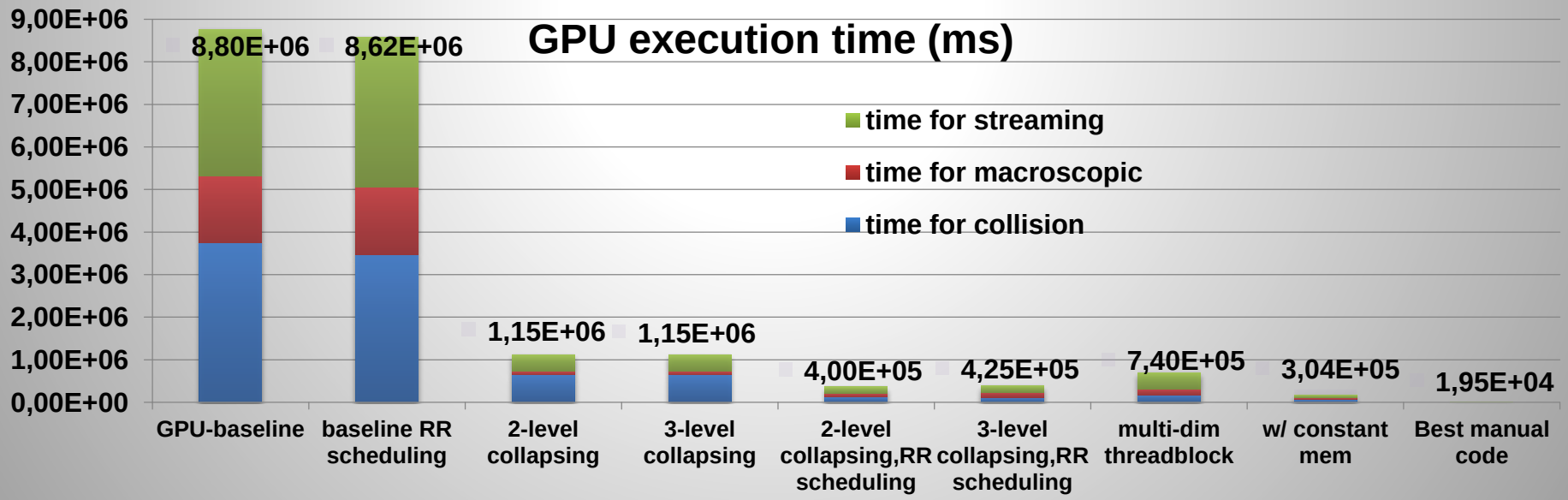
```
// High level kernel of LB.

// potential location for  #pragma omp target data
// complete, complex data types are exposed
 for (iBox = 0; iBox < nBox; ++iBox)
    {
      bidx = *dit;
      ++dit;
// Collision () contains a OMP target parallel for loop
// operating on extracted simpler data
      LBPatch::collision(m_boxes[bidx],
                         fi()[bidx],
                         U()[bidx],
                         m_tau,
                         m_bodyForce);
    }
```

A reference manual version (LB)

- Fab structure is simplified to store only the essential data members.
  - Easier to be offloaded
  - Enable later consolidation of all memory copying to avoid memory traffic between host and device
  - Expose more constant memory usage
- Some Chombo library functions: converted into CUDA version
  - Enable more code regions to be offloaded

Performance (LB)



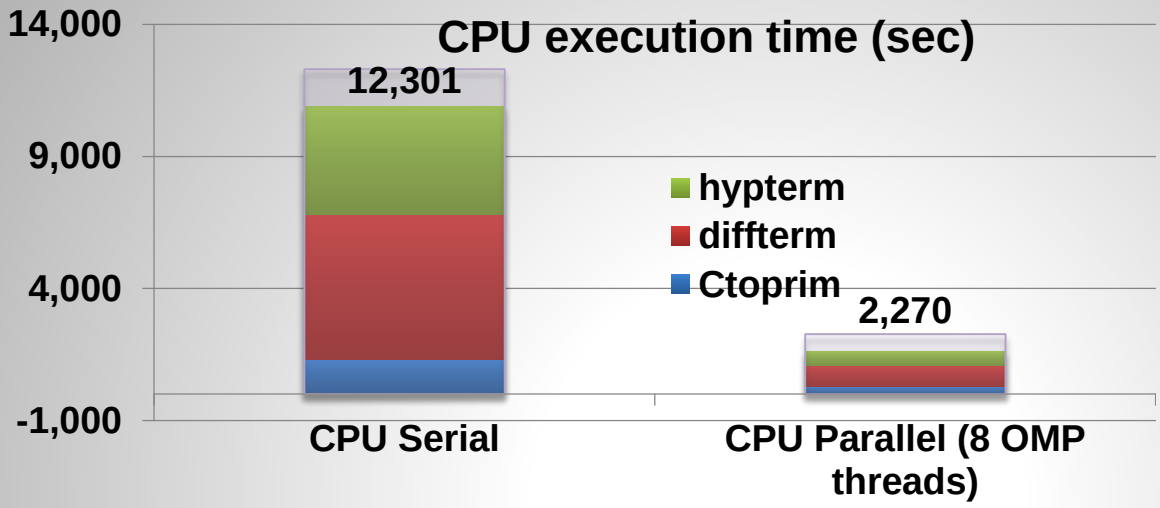


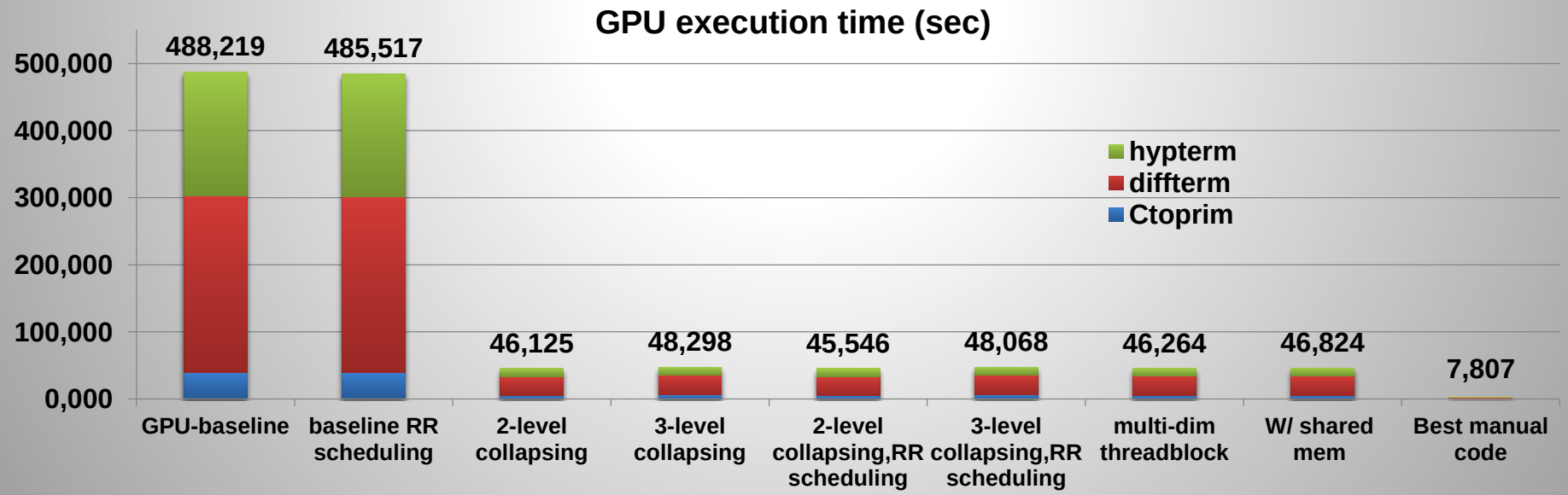
A reference manual version (CNS)

- Merge 3 directional split loops into one.
- Use 3-D 4^3 thread block size
  - Maximize usage of shared memory
  - Avoid partial warp (1 warp = 32 GPU threads) utilization

```
//Example host code:
 dim3 _threads_per_block_(4,4,4);
 dim3 _num_blocks_((iFluxUpperBound-iFluxLowerBound+1)/4, (jFluxUpperBound-jFluxLowerBound +1)/4,
(kFluxUpperBound – kFluxLowerBound + 1) / 4);
    hypterm_kernel<<<_num_blocks_,_threads_per_block_>>>(…);

// Example device code in CUDA:
__shared__ double s_q[3][2][12];
int _p_i = threadIdx.x + blockIdx.x * 4 + iFluxLowerBound;
int _p_j = threadIdx.y + blockIdx.y * 4 + jFluxLowerBound;
int _p_k = threadIdx.z + blockIdx.z * 4 + kFluxLowerBound;
s_q[0][0][threadIdx.x + 4] = _dev_q[qu][_p_k][_p_j][_p_i];
s_q[0][0][threadIdx.x ]      = _dev_q[qu][_p_k][_p_j][_p_i+4];
s_q[0][0][threadIdx.x - 4]  = _dev_q[qu][_p_k][_p_j][_p_i-4];
```

Performance (CNS)





Productivity study

Productivity report (lines of code)								
Functions	Source #	Directives	Generated Host code #		Generated Device code #		ratio (generated # / directives)	
				w/ count in runtime		w/ count in runtime		w/ count in runtime
LB collision	45	2	57	464	48	58	52.5	261.0
LB macroscopic	46	2	52	421	45	55	48.5	238.0
LB stream	21	2	53	460	35	45	44.0	252.5
CNS ctoprim	14	2	27	205	30	40	28.5	122.5
CNS hypterm	57	6	81	793	123	153	34.0	157.7
CNS diffterm	82	14	335	2647	206	276	38.6	208.8

Discussion

- OpenMP accelerator model is straightforward in the porting process
  - Prepare a baseline OpenMP CPU version
  - Incrementally apply accelerator directives and clauses to improve performance
- Challenges:
  - Map clause does not allow complex data types
    - Impact #omp target device and #omp target data directives
  - 3rd party libraries complicates porting to GPUs: CUDA version vs. use CPUs
  - Non-perfectly nested loops prohibit collapsing
  - The current specification does not exploit accelerator memory hierarchy

Discussion (Cont.)

- Attempted workaround and extensions
  - Unsupported data types for map
    - Extract simpler portions of a complex data type for data offloading
  - Significant manual code simplification and reorganizing to expose more data reuse opportunities
    - CNS: three directional split loops merged into one 25-point loop
  - 3rd party libraries
    - Prepare CUDA versions for key lib functions
  - Non-perfectly nested loops
    - Force loop collapsing with developer's consent
  - cache (var_list) to exploit software managed memories
    - var_list contain reusable variables, some of which are constants indicated as "const var"
    - Extra memory management code may hurt performance

Thank You!

- Questions?

Experimenting share memory with tiling

- Shared memory is shared within a thread block.
- The maximum thread blocks per multiprocessor in GPU is limited by the shared memory per multiprocessor.
- The occupancy of each multiprocessor is affected by the shared memory usage.

Shared memory report			
Kernel	size/block (byte)	threads/ block	Occupancy
Hypterm original	1920	40	50%
Tiled 2 iterations	3840	80	56%
Tiled 3 iterations	5760	120	50%
Tiled 4 iterations	7680	160	47%
Diffterm original	3520	40	41%
Tiled 2 iterations	7040	80	28%
Tiled 3 iterations	10620	120	25%
Tiled 4 iterations	14080	160	23%

Max 48KB for shared memory on GPU model ???
Tiling increases size of shared memory, but hurts GPU occupancy

LB Pseudo code:

```
fi(cells, 19, boxes) = initial data;
fiUpdate(cells, 19, boxes) = 0;
U(grid, 4, boxes);
Macroscopic(U, fi);
for (int iTS = 0; iTS != nTimeStep; ++iTS)
{
  int iBox;
  for (every box)
  {
    {  // Advance function
      for (every cell)
        Collision(fi, U);
      Exchange(fi);
      BC(fi);
      Stream(fiUpdate, fi);
      for (every cell)
        Macroscopic(U, fiUpdate);
      swap(fi, fiUpdate);
    }
  }
}
```

CNS Pseudo code:

```
init_data(U, dx, prob_lo, prob_hi)
for (int iTS = 0; iTS != nTimeStep; ++iTS)
{
  int iFab;
  for (every Fab)
    {
     // Advance function
      for (1/3 timestep)
        {
        // Advance 1/3 of timestep in each iter.
          for (every grid cell)
            ctoprim(Unew, Q);
          for (every grid cell)
            diffterm(Q, D);
          for (every grid cell)
            hypterm(Q, F);
        }
    }
}
```

Reduce data movement: difficulties in using #target data

- CNS:
  - Additional outer loop iterates over multiple Fabs.
  - Data pointers are extracted only for each Fab.
  - Not trivial to map Fab data structure

Outer code for CNS.
ctoprim function contains OMP parallel loops that can be offloaded to GPU.

```
do n=1,nfabs(Q)

      up => dataptr(U,n)
      qp => dataptr(Q,n)

      lo = lwb(get_box(Q,n))
      hi = upb(get_box(Q,n))
      call ctoprim(lo,hi,ng,up,qp,dx,courno_proc)
   end do
```