Early Experiences with the OpenMP Accelerator Model

Chunhua (Leo) Liao, Yonghong Yan*, Bronis R. de Supinski, Daniel J. Quinlan, and Barbara Chapman*

Canberra, Australia, IWOMP 2013, Sep. 17th

Lawrence Livermore National Laboratory

* University of Houston

LLNL-PRES- 642558

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
- OpenMP 4.0’s accelerator support
- Implementation
- Preliminary results
- Discussions

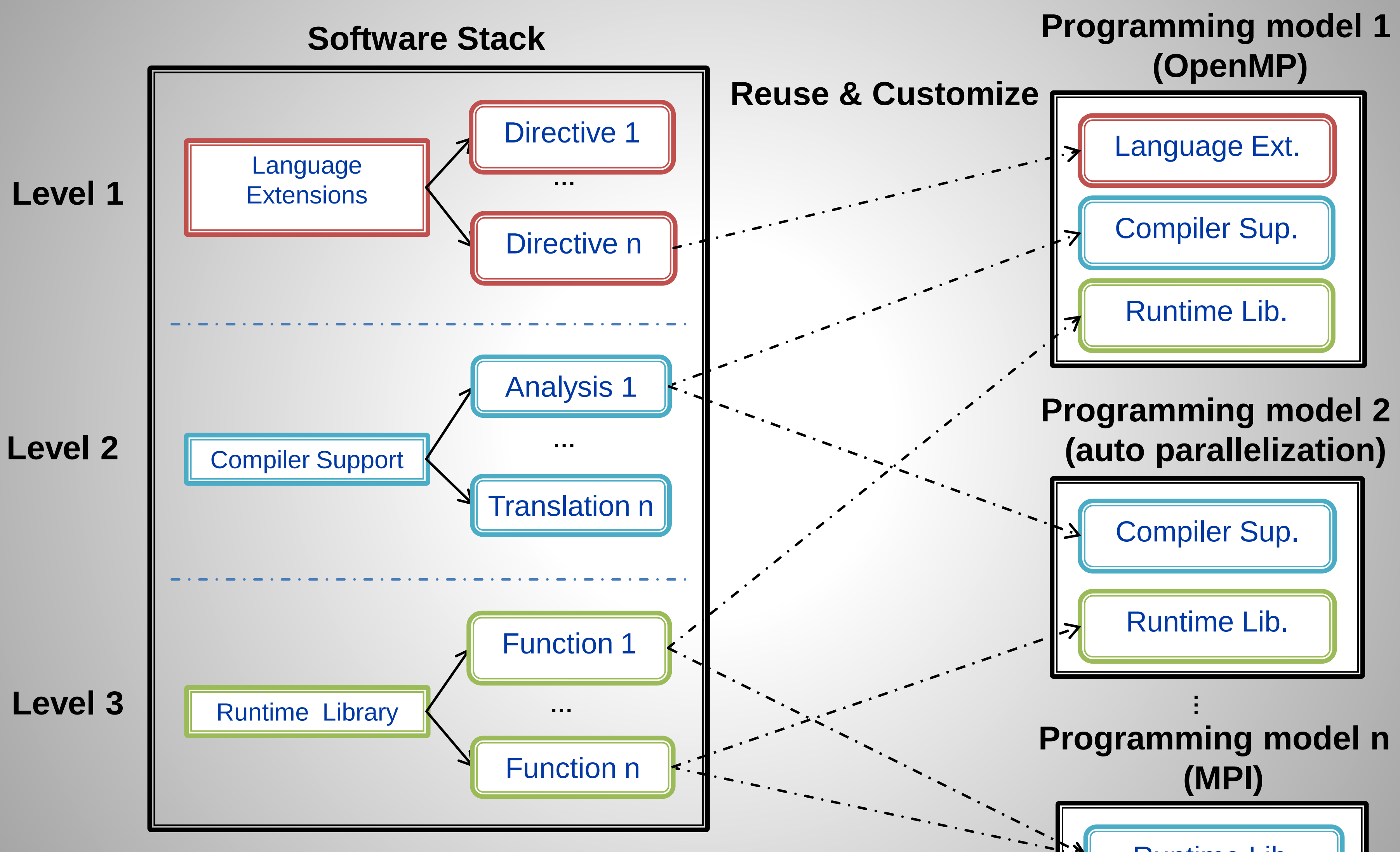
Motivation: a framework for creating node-level parallel programming models for exascale

- Problem:
  - Exascale machines: more challenges to programming models
  - Parallel programming models: important but increasingly lag behind node-level architectures

- Goal:
  - Speedup designing/evolving/adopting programming models for exascale

- Approach:
  - Identify and develop common **building blocks** in node-level programming models
  - Both **researchers** and **developers** can quickly create or customize their own models
  - Demonstrate the framework by providing **example programming models**

Building blocks and programming models



Programming model framework

Example programming model: Heterogeneous OpenMP (HOMP)

- Goal
  - Address heterogeneity challenge of exascale: computing using accelerators
  - Discover/develop building blocks for directive parsing, compiler translation, and runtime support
- Approach
  - Explore extensions to a general-purpose programming model
    - OpenMP accelerator model: OpenMP directives + accelerator directives
  - Build on top of existing OpenMP implementation in ROSE

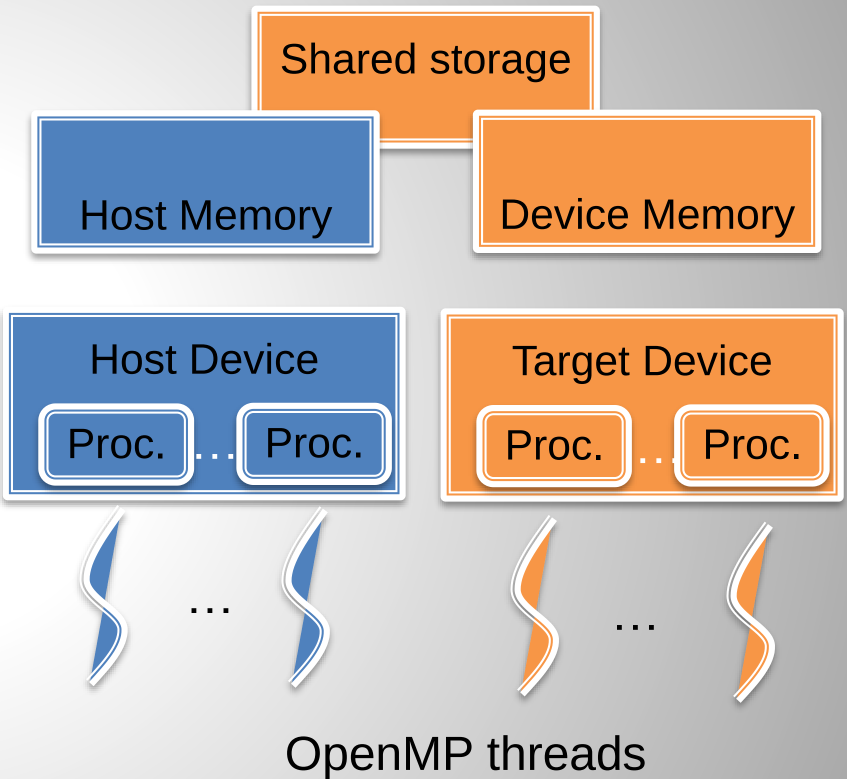
Language level:
target, device, map, num_devices, reduction, device_type, no-mid-sync, …

Compiler Level:
parser building blocks
outliner,
loop transformation,
data transformation, …

Runtime Level:
device probing,
data management, loop scheduling, reduction, …

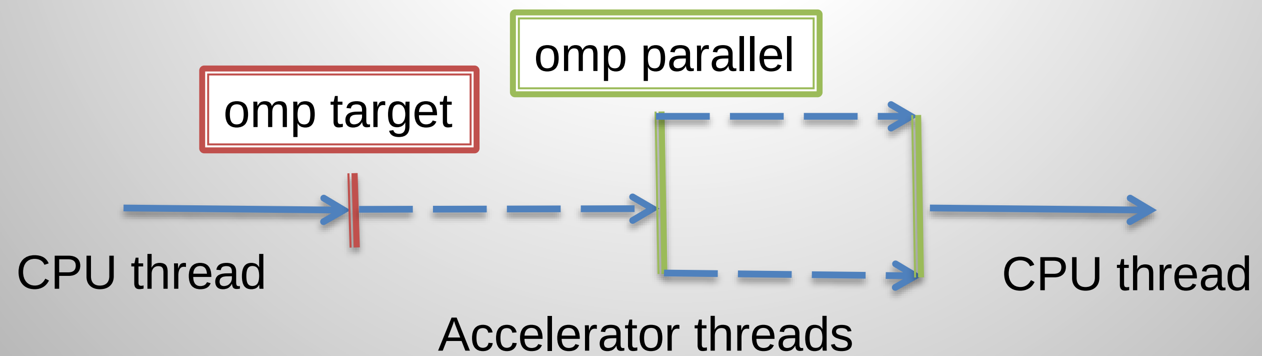
OpenMP 4.0's accelerator model

- 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



Example code: Jacobi

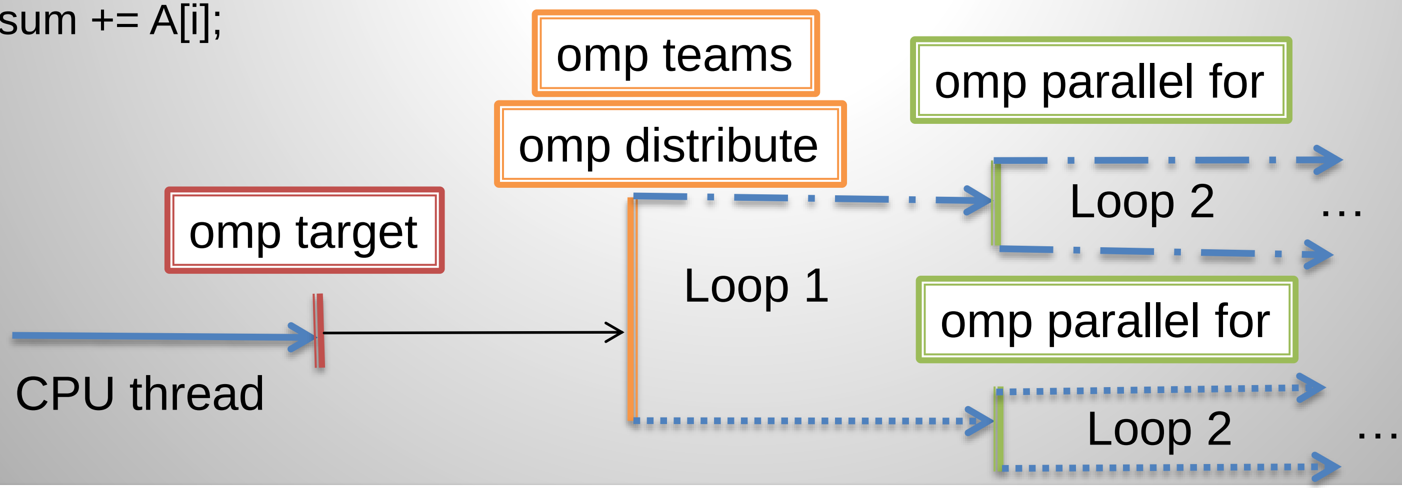
```
#pragma omp target  data device (gpu0) map(to:n, m, omega, ax, ay, b, \
   f[0:n][0:m]) map(tofrom:u[0:n][0:m]) map(alloc:uold[0:n][0:m])
while ((k<=mits)&&(error>tol))
{
// a loop copying u[][] to uold[][] is omitted here
 …
#pragma omp target  device(gpu0) map(to:n, m, omega, ax, ay, b, f[0:n][0:m], \
    uold[0:n][0:m])  map(tofrom:u[0:n][0:m])
#pragma omp parallel for private(resid,j,i) reduction(+:error)
for (i=1;i<(n-1);i++)
  for (j=1;j<(m-1);j++)
  {
    resid = (ax*(uold[i-1][j] + uold[i+1][j])\
        + ay*(uold[i][j-1] + uold[i][j+1])+ b * uold[i][j] - f[i][j])/b;
    u[i][j] = uold[i][j] - omega * resid;
    error = error + resid*resid ;
  } // the rest code omitted  ...
}
```

Thread Hierarchy

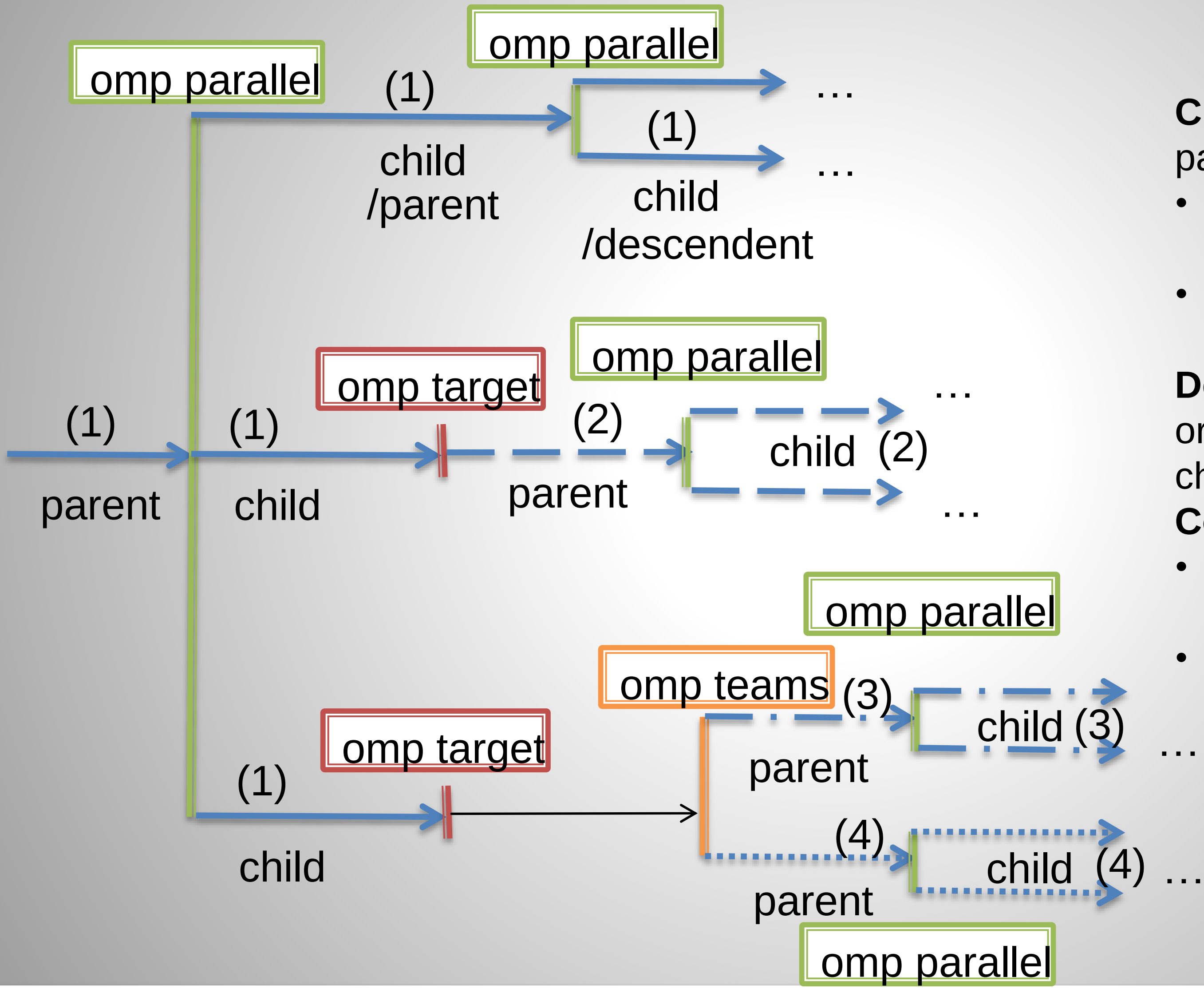
- Accelerators: massively parallel with hundreds or even thousands of threads with different synchronization/memory access properties
  - E.g. CUDA: threads → thread blocks → grid
- OpenMP Accelerator model:
  - Thread hierarchy: threads -> teams -> league
  - **#pragma omp teams [num_teams() thread_limit() …]**
    - Create a league of thread teams and the master thread of each team executes the region
    - **num_teams(int-exp)**: upper limit of the number of teams
    - **thread_limit(int-exp)**: upper limit of threads of each team
  - **#pragma omp distribute [collapse() dist_schedule()]**
    - Worksharing among master threads of teams
    - **dist_schedule(kind[,chunk_size])** : static chunk based round-robin scheduling

Example code using teams and distribute

```
int sum = 0;
int A[1000];
...
#pragma omp target device(gpu0) map(to:A[0:1000]) map(tofrom:sum)
#pragma omp teams num_teams(2) thread_limit(100) reduction(+:sum)
#pragma omp distribute
  for (i_0 = 0; i_0 < 1000; i_0 += 500) // Loop 1
{
#pragma omp parallel for reduction(+:sum)
 for (i = i_0; i < i_0 + 500; i++)  // Loop 2
    sum += A[i];
}
```



Contention groups



Child thread: single level parent-child relationship

- Generated by a **parallel** construct:
- Not for a **target** or a **team** construct

Descendent thread: one or multiple levels parent-child relation

Contention group:

- An initial thread and its descendent threads
- May contain multiple thread teams (nested parallelism)

Other language features

- target declare: a declarative directive specifies that variables, functions and subroutines to be mapped to a device
- target update: make specified items in the device data environment consistent with their original list items
- Combined constructs (4.0)
  - #pragma target teams distribute
    - #pragma omp target teams
    - #pragma omp teams distribute
  - #pragma target teams distributed parallel for
    - #pragma omp teams distribute parallel for
- Environment variables
  - OMP_DEFAULT_DEVICE: the default device number
- Runtime library routines
  - void omp_set_default_device(int device_num);
  - int omp_get_num_devices(void);
  - int omp_get_num_teams(void);
  - int omp_is_initial_device(void);

A prototype Implementation: HOMP

- HOMP: Heterogeneous OpenMP
  - Built upon ROSE's OpenMP implementation*
    - Focus on CUDA code generation
    - 1st version has been released with ROSE
  - Extensions to OpenMP lowering
    - Code: target + parallel regions, loops, CUDA kernels
    - Variables: data environments, arrays, reduction
  - Extensions to the XOMP runtime library
    - Support compiler translation: loop scheduling, data handling, kernel launch configuration, reduction, …
    - C bindings: interoperate with C/C++ and Fortran

*C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski, "A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries," in Beyond loop level parallelism in OpenMP: accelerators, tasking and more (IWOMP'10), Springer, 2010, pp. 15-28.

Target + parallel regions

- A CUDA kernel launch for a target region
- A parallel region within a target region
  - May contain barriers in the middle of a parallel region
  - But only threads within the same thread block can (easily) synchronize in CUDA
- Explicit use of omp teams
  - Spawn CUDA threads with right number of thread blocks and threads per block
- Without omp teams
  - Choice 1: conservatively spawn threads within one block only
  - Choice 2: spawn threads across multiple thread blocks after using an analysis to ensure no synchronization points inside a parallel region
  - Choice 3: a new clause to indicate no synchronization points

Loops

- Map to a two-level thread hierarchy
  - Master threads of teams vs. threads of a same team
- Naïve mapping: 1 iteration to 1 CUDA thread
  - Limited number of total CUDA threads or thread blocks
- Scheduling
  - Static: static even vs. round robin with a chunk size
  - Dynamic and guided?
- Multi-level parallel loops: collapse
  - Choice 1: linearization
  - Choice 2 (only for 2 or 3 levels): map to 2-D or 3-D thread blocks

Data handling

- Translation of device variables
  - Basic operations
    - Data declaration, allocation, copying, deallocation
  - Multi-dimensional arrays
    - Flattened to be 1-D arrays to be operated on GPU
  - Data environments
    - A stack: each layer stores pointers to data allocated within a data environment
    - Track and reuse data within nested data regions: search the stack
  - Reduction: contiguous reduction pattern (folding)
    - Two levels: GPU device and CPU Host
  - Implemented through both compiler translations and runtime support

Example code: Jacobi using OpenMP accelerator directives

```
#pragma omp target  data device (gpu0) map(to:n, m, omega, ax, ay, b, \
   f[0:n][0:m])  map(tofrom:u[0:n][0:m]) map(alloc:uold[0:n][0:m])

while ((k<=mits)&&(error>tol))
{
// a loop copying u[][] to uold[][] is omitted here
 …
#pragma omp target  device(gpu0) map(to:n, m, omega, ax, ay, b, f[0:n][0:m], \
    uold[0:n][0:m])  map(tofrom:u[0:n][0:m])
#pragma omp parallel for private(resid,j,i) reduction(+:error)
for (i=1;i<(n-1);i++)
  for (j=1;j<(m-1);j++)
  {
    resid = (ax*(uold[i-1][j] + uold[i+1][j])\
        + ay*(uold[i][j-1] + uold[i][j+1])+ b * uold[i][j] - f[i][j])/b;
    u[i][j] = uold[i][j] - omega * resid;
    error = error + resid*resid ;
  } // the rest code omitted  ...
}
```

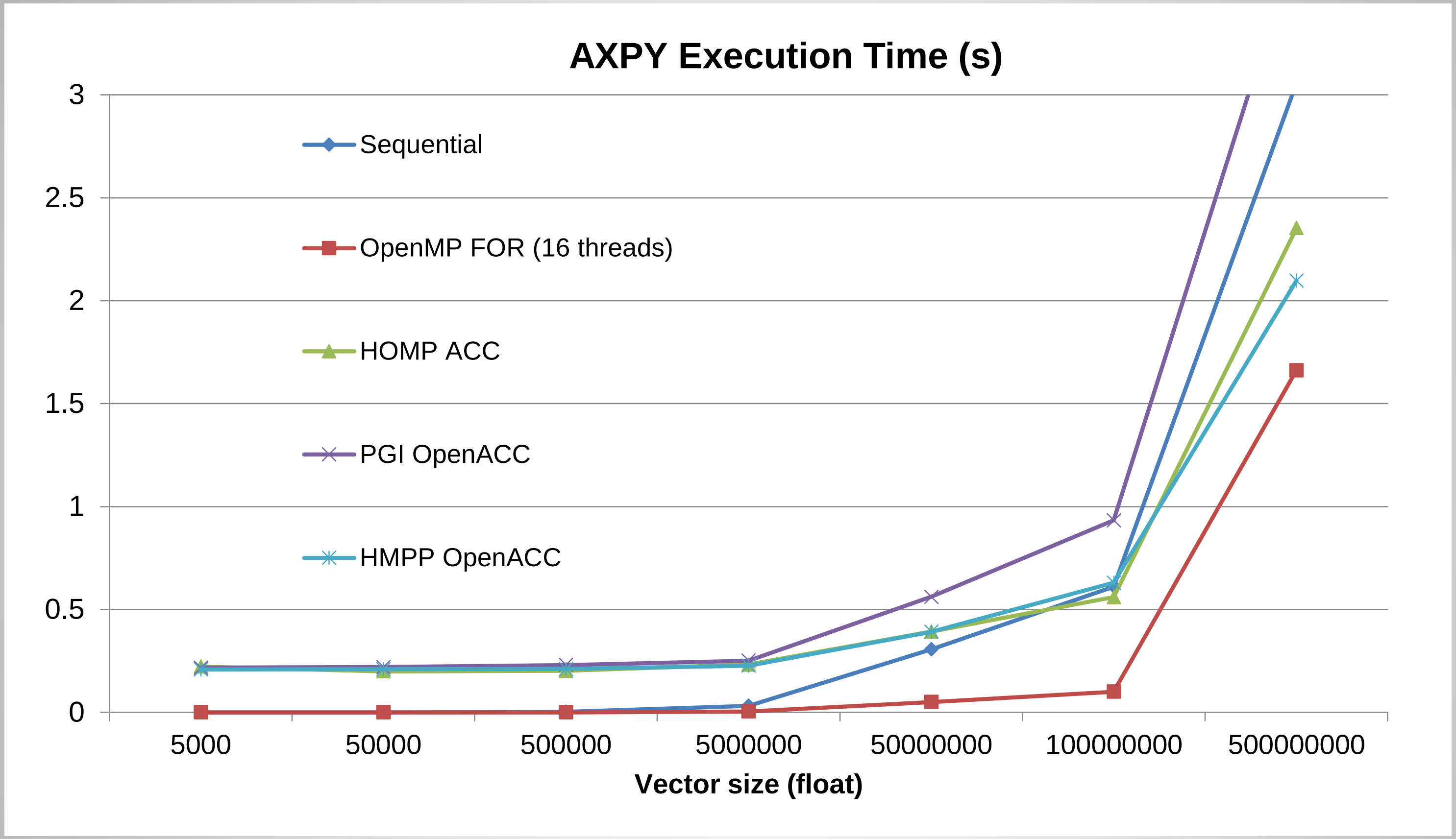
HOMP implementation: GPU kernel generation

```
__global__ void OUT__1__10117__(int n, int m, float omega, float ax, float ay, float b, \
            float *_dev_per_block_error, float *_dev_u, float *_dev_f, float *_dev_uold)
{   /* local variables for  loop , reduction, etc */
 int _p_j;    float _p_error;      _p_error = 0;    float _p_resid;
 int _dev_i, _dev_lower, _dev_upper;
/* Static even scheduling:  obtain loop bounds for current thread of current block */
 XOMP_accelerator_loop_default (1, n-2, 1, &_dev_lower, &_dev_upper);
 for (_dev_i = _dev_lower; _dev_i<= _dev_upper; _dev_i ++) {
    for (_p_j = 1; _p_j < (m - 1); _p_j++) {  /*  replace  with device variables,  linearize  2-D array accesses */
     _p_resid = (((((ax * (_dev_uold[(_dev_i - 1) * 512 + _p_j] + _dev_uold[(_dev_i + 1) * 512 + _p_j])) \
              + (ay * (_dev_uold[_dev_i * 512 + (_p_j - 1)] + _dev_uold[_dev_i * 512 + (_p_j + 1)]))) \
              + (b * _dev_uold[_dev_i * 512 + _p_j])) - _dev_f[_dev_i * 512 + _p_j]) / b);
    _dev_u[_dev_i * 512 + _p_j] = (_dev_uold[_dev_i * 512 + _p_j] - (omega * _p_resid));
    _p_error = (_p_error + (_p_resid * _p_resid));
    } }  /* thread block level reduction for float type*/
 xomp_inner_block_reduction_float(_p_error,_dev_per_block_error,6);
}
```

HOMP implementation: CPU side handling

```
xomp_deviceDataEnvironmentEnter(); /* Initialize a new data environment, push it to a stack */
float *_dev_u = (float*) xomp_deviceDataEnvironmentGetInheritedVariable ((void*)u, _dev_u_size);
/* If not inheritable, allocate and register the mapped variable */
if (_dev_u == NULL)
{   _dev_u = ((float *)(xomp_deviceMalloc(_dev_u_size)));
 /* Register this mapped variable original address, device address, size, and a copy-back flag */
 xomp_deviceDataEnvironmentAddVariable ((void*)u, _dev_u_size, (void*) _dev_u, true);
}
/* Execution configuration: threads per block and total block numbers */
int _threads_per_block_ = xomp_get_maxThreadsPerBlock();
int _num_blocks_ = xomp_get_max1DBlock((n - 1) - 1);
/* Launch the CUDA kernel ... */
OUT__1__10117__<<<_num_blocks_,_threads_per_block_,(_threads_per_block_ * sizeof(float ))>>> \
    (n,m,omega,ax,ay,b,_dev_per_block_error,_dev_u,_dev_f,_dev_uold);
error = xomp_beyond_block_reduction_float(_dev_per_block_error,_num_blocks_,6);
/* Copy back (optionally) and deallocate variables mapped within this environment, pop stack */
xomp_deviceDataEnvironmentExit();
```

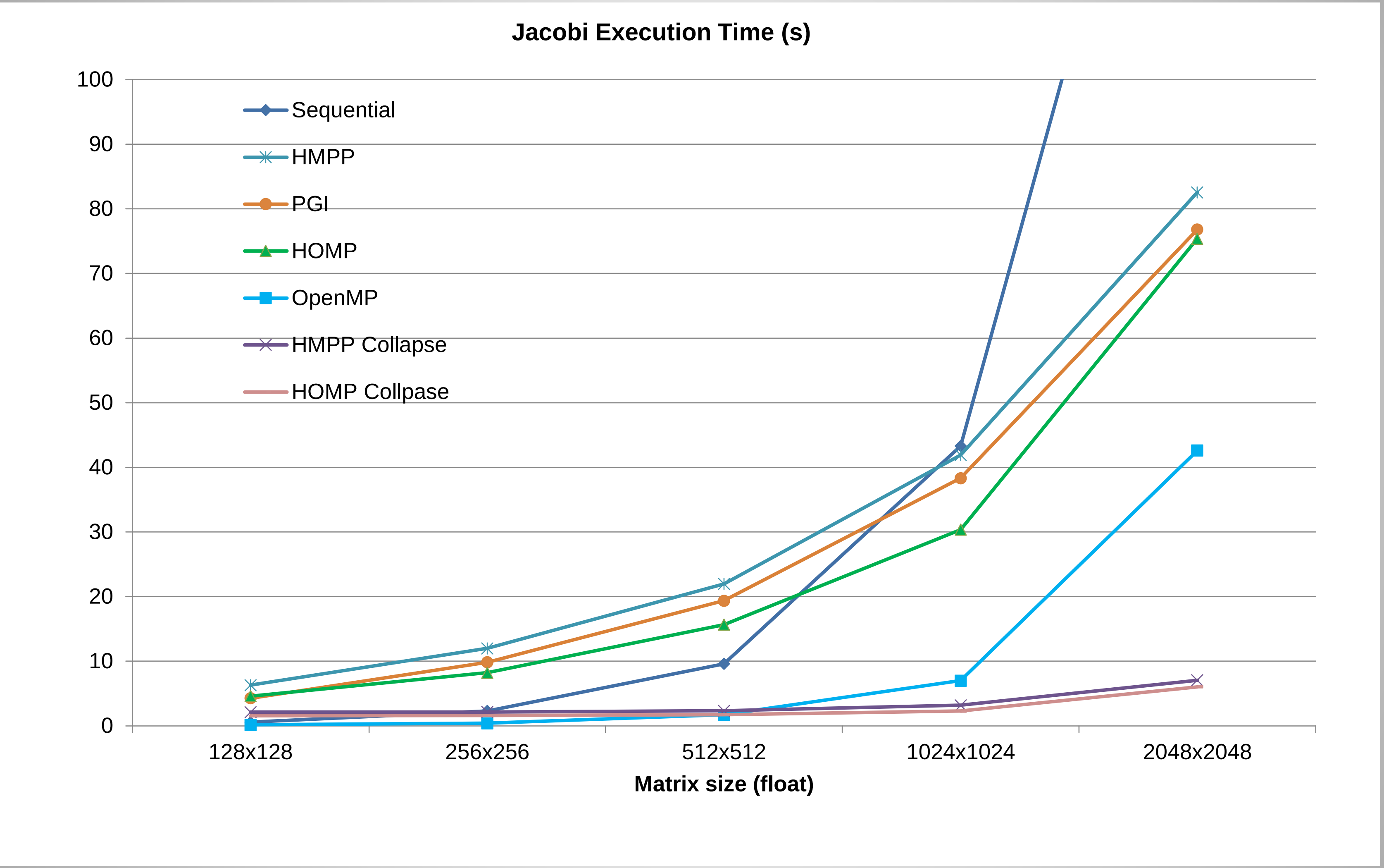
Preliminary results: AXPY (Y=a*X)



Hardware configuration:
- 4 quad-core Intel Xeon processors (16 cores) 2.27GHz with 32GB DRAM.
- NVIDIA Tesla K20c GPU (Kepler architecture)

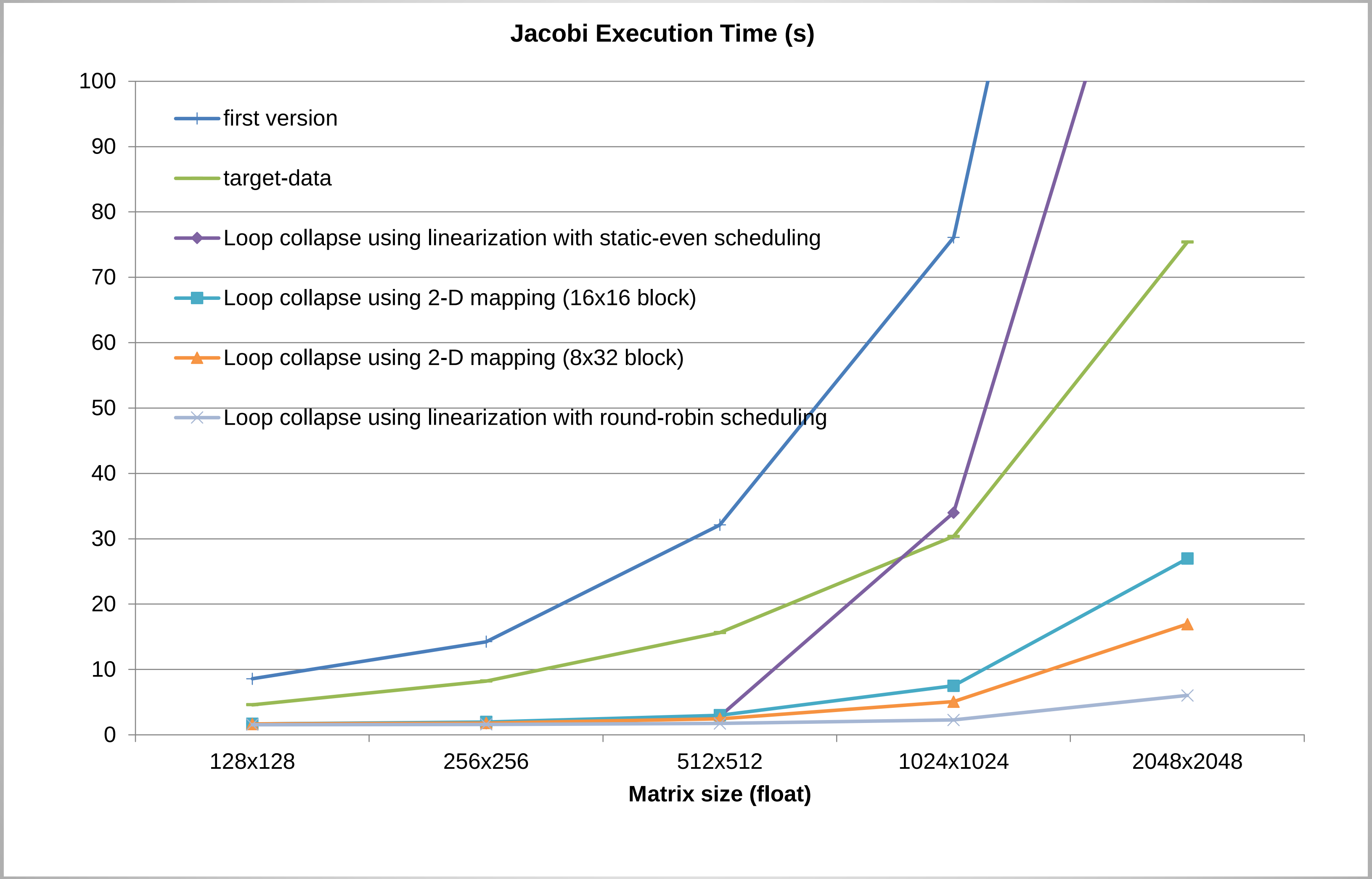
Software configuration:
- PGI OpenACC compiler version 13.4
- HMPP OpenACC compiler version 3.3.3
- GCC 4.4.7 and the CUDA 5.0 compiler

Jacobi



Jacobi Execution Time (s)
100
90
80
70
60
50
40
30
20
10
0
Sequential
HMPP
PGI
HOMP
OpenMP
HMPP Collapse
HOMP Collpase
128x128
256x256
512x512
1024x1024
2048x2048
Matrix size (float)

HOMP: multiple versions of Jacobi



* static-even scheduling on GPU can be really bad

Compared to hand-coded versions: matrix multiplication

Version/Matrix Size	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096
HOMP collapse (seconds)	0.208907	0.208024	0.213775	0.236833	0.398918	1.677306
CUDA SDK (seconds)	0.000034	0.000141	0.001069	0.008441	0.067459	0.538174
CUBLAS (seconds)	0.000024	0.000054	0.000207	0.001092	0.007229	0.052283
Ratio (HOMP/SDK)	6144.323529	1475.347518	199.9766137	28.05745765	5.913488193	3.116661154
Ratio (HOMP/CUBLAS)	8704.458333	3852.296296	1032.729469	216.880037	55.18301286	32.08128837

- The CUDA SDK and CUBLAS versions: different algorithms, aggressive optimizations (use of shared memory within blocks and apply tiling to the algorithms)
  - Our version: naïve i-j-k order, with scalar replacement for C[i][j]
- Difference decreases for larger inputs

Discussion

- Positive
  - Compatible extensions: device, memory and execution models
  - Implementation is mostly straightforward
- Improvements
  - Multiple device support:
    - Current: device(id), portability concern
    - Desired: device_type(), num_devices(), data_distribute()
  - New clause: no-middle-sync
    - Avoid cumbersome explicit teams
  - Loop scheduling policy
    - May need per device default scheduling policy
  - Productivity: combined directives
    - omp target parallel: avoid initial implicit sequential task
  - Global barrier
    - #pragma omp barrier [league|team]
  - No (separated, delayed) examples!?

Future work

- Multiple accelerators
  - device_type, num_devices, distribute_map
- Exploit CUDA shared memory (programmable cache of accelerators)
- Target Intel MIC (Many Integrated Core) architecture
- Generate OpenCL codes
- Peer-to-peer execution model
  - Code and data offload without involving a host
- Choice between CPU threads, accelerator threads and vectorization

Thank You!

- Questions?

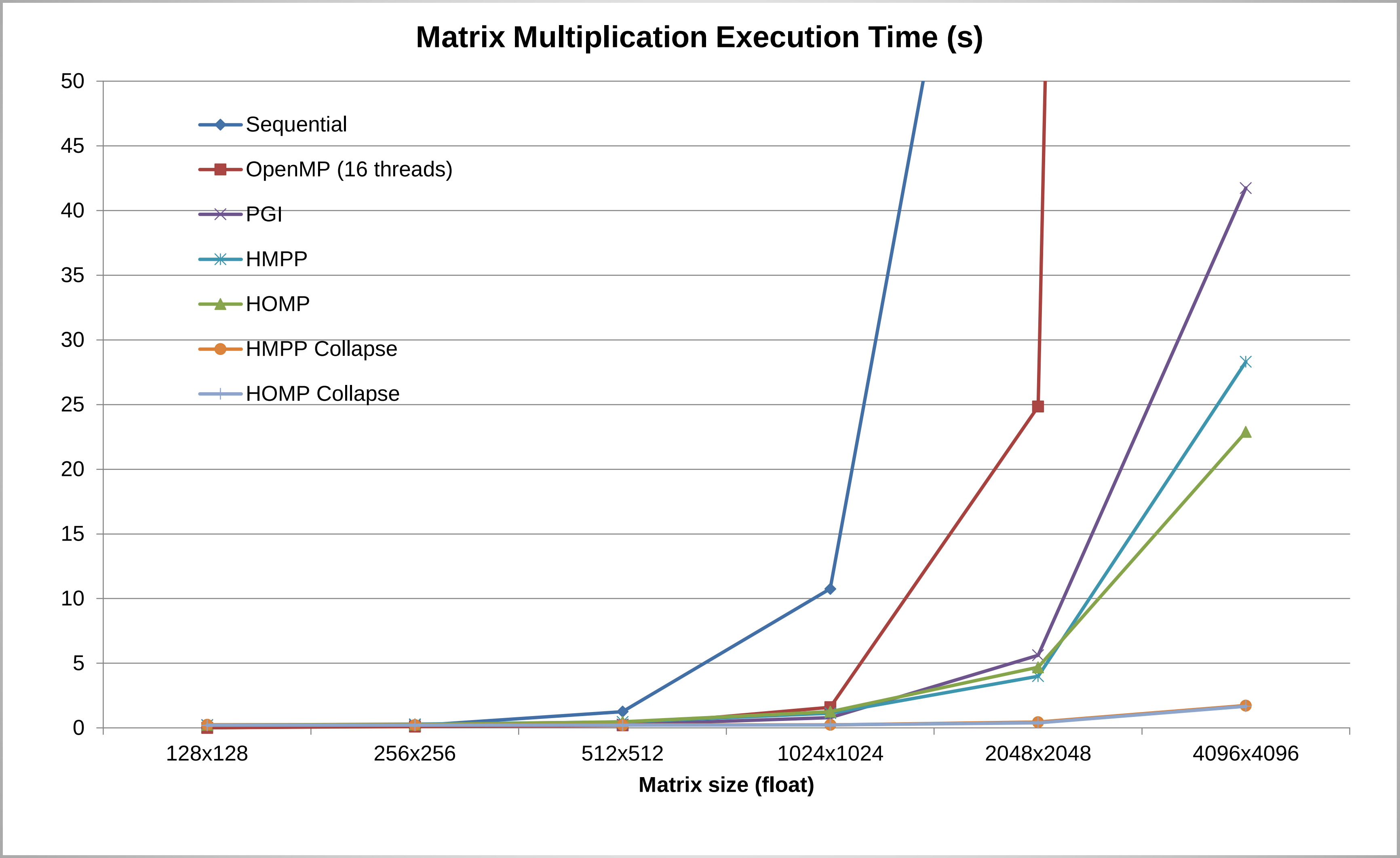
Beginner's lessons learned with CUDA

- Avoid oversubscription for CUDA threads per block
  - Max allowed thread per block (1024) vs. cores per multi-processor (192)
- Be aware of limitation of naïve mapping of 1 loop iteration to 1 CUDA thread
  - + No loop scheduling overhead
  - - Not enough threads for all iterations
- Loop scheduling policy:
  - Static even scheduling vs. round-robin scheduling (scheduling(static,1)): big difference in memory footprint for per CUDA thread (block) and memory coalescing
- 2-D mapping: match warp size (32)

Data reuse details

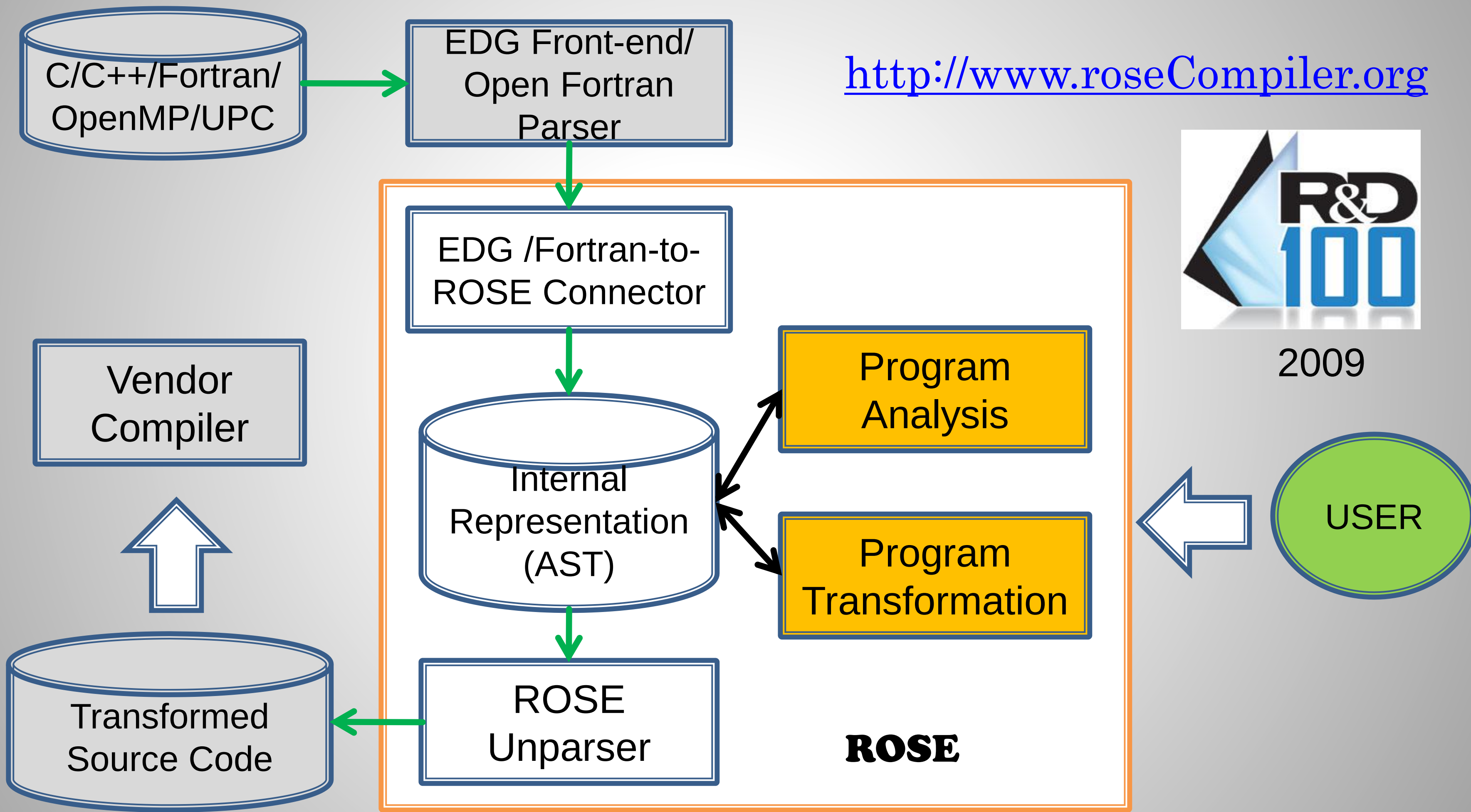
- Data reuse across nested data environments
  - Keep track via a stack of data environment data structures
  - Push stack
    - void xomp_deviceDataEnvironmentEnter ( )
  - Search stack
    - void* xomp_deviceDataEnvironmentGetInheritedVariable (void *host_var, int size)
  - Register a new variable
    - void xomp_deviceDataEnvironmentAddVariable(void* host_var, int size, void* dev_var, bool copy_back)
  - Pop stack
    - void xomp_deviceDataEnvironmentExit ( )

Matrix multiplication

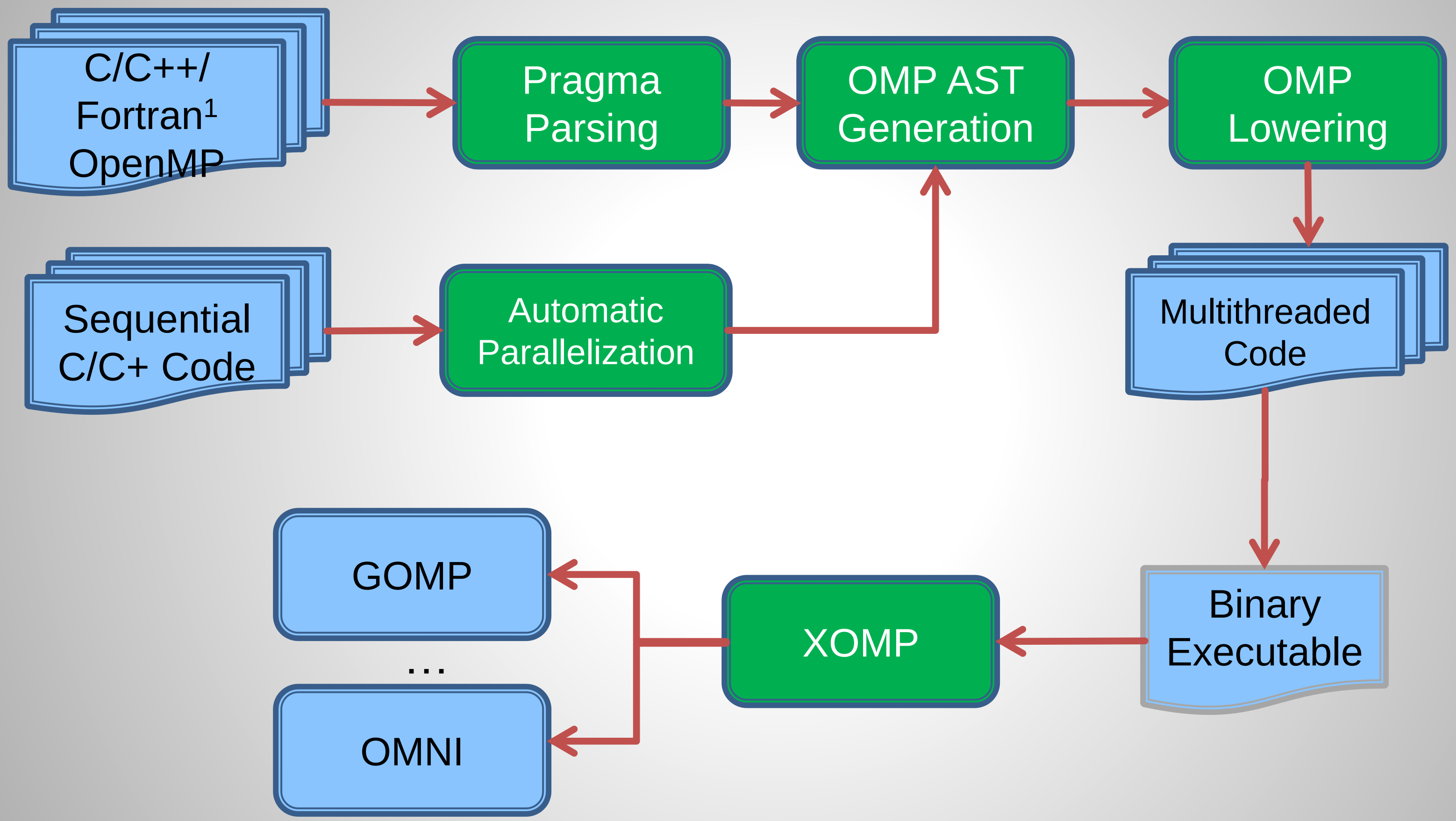


*HOMP-collapse slightly outperforms HMPP-collapse when linearization with round-robin scheduling is used.

ROSE: a source-to-source compiler infrastructure



OpenMP Support in ROSE: Overview



1. Fortran support is still work in progress

League, team, and contention group

- Thread teams: hierarchical organization of accelerator threads
  - Threads executing a parallel region
- League
  - set of thread teams
- Child thread vs. descendent thread
  - Child thread: single level parent-child relationship
    - Generated by a parallel construct: parent → parallel → child
    - No child threads for a target or a team construct
  - Descendent thread: one or multiple levels parent-child relation
- Contention group:
  - An initial thread and its descendent threads
  - May contain multiple thread teams (nested parallelism)

Reduction

- A two-level algorithm using contiguous reduction (like folding)
- Support multiple data types and reduction operations
  - Level 1: reduction within a thread block on GPU
    - xomp_inner_block_reduction_[type]()
  - Level 2: reduction across thread blocks on CPU
    - xomp_beyond block reduction_[type]()

Matrix multiplication kernel

```
#pragma omp target map(tofrom: c[0:N][0:M]), map(to: a[0:N][0:M], b[0:M][0:K], j, k)
#pragma omp parallel for private(i,j,k) collapse (2)
 for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
  {
     float t =0.0;
    for (k = 0; k < K; k++)
      t += a[i][k]*b[k][j];
    c[i][j] = t;
   }
```

Discussion (cont.)

- Improvements
  - Runtime overhead for data tracking
    - — OpenACC: explicit *present* clause
    - — OpenMP: nested data environments, overhead is negligible
  - Nested loops
    - — Collapse: choice between linearization vs. 2-D/3-D mapping
  - Array sections
    - — Different notations for C/C++ and Fortran
    - — Uniformed notation: easier for users and compiler developers

Matrix multiplication: time breakdown