Performance Portability with OpenMP on NVIDIA GPUs

Carlo Bertolli

Advanced Compiler Technology Team

IBM T.J. Watson Research Center

Technical Work by Gheorghe-Teodor “Doru” Bercea

Team: Kevin O’Brien, Samuel Antao, Alexandre Eichenberger, Arpith Jacob, Zehra Sura, Tong Chen, Hyojin Sung, Georgios Rokos

DOE Centers of Excellence Performance Portability Meeting

April 20 2016

Research Goals

- **Obtain same GPU performance when writing CUDA and OpenMP 4**
  - What is the performance of a simple porting?
  - Can I tune my application to match CUDA?
- **Proxy** application analysis: **LULESH**
  - One of five DARPA challenge problems
  - Represents code that accounts for 30% of the runtime on DoE/DoD supercomputers
  - Already ported to CUDA →
- Broad Strategy
  - Look at many different kinds of applications
  - Develop optimization schemes and mechanisms for each class
  - Merge together in an optimizing compiler

> - Not 1-to-1 mapping between CUDA and OpenMP 3.1 versions
> - CUDA hand-transformations:
>   - Loop interchange
>   - Loop fusion
>   - etc..

LULESH OpeMP 4.0

- **Based on OpenMP 3.1 version**

```
#pragma omp parallel for
for (Index_t i = 0 ; i < numElem ; ++i)
{
  sigxx[i] = sigyy[i] = sigzz[i] =  - p[i] - q[i] ;
}
```

Data mapping

```
#pragma omp target data \
  map(to: p[:numElem], q[:numElem]) \
  map(from: sigxx[:numElem], sigyy[:numElem]) \
  map(from: sigzz[:numElem])
{
  #pragma omp target teams distribute parallel for
  for (Index_t i = 0 ; i < numElem ; ++i)
  {
    sigxx[i] = sigyy[i] = sigzz[i] =  - p[i] - q[i] ;
  }
}
```

- **target**: offload to GPUs
- **teams**: use many CUDA blocks
- **parallel**: use many CUDA threads
- **distribute** and **for**: block loop and schedule to blocks and threads

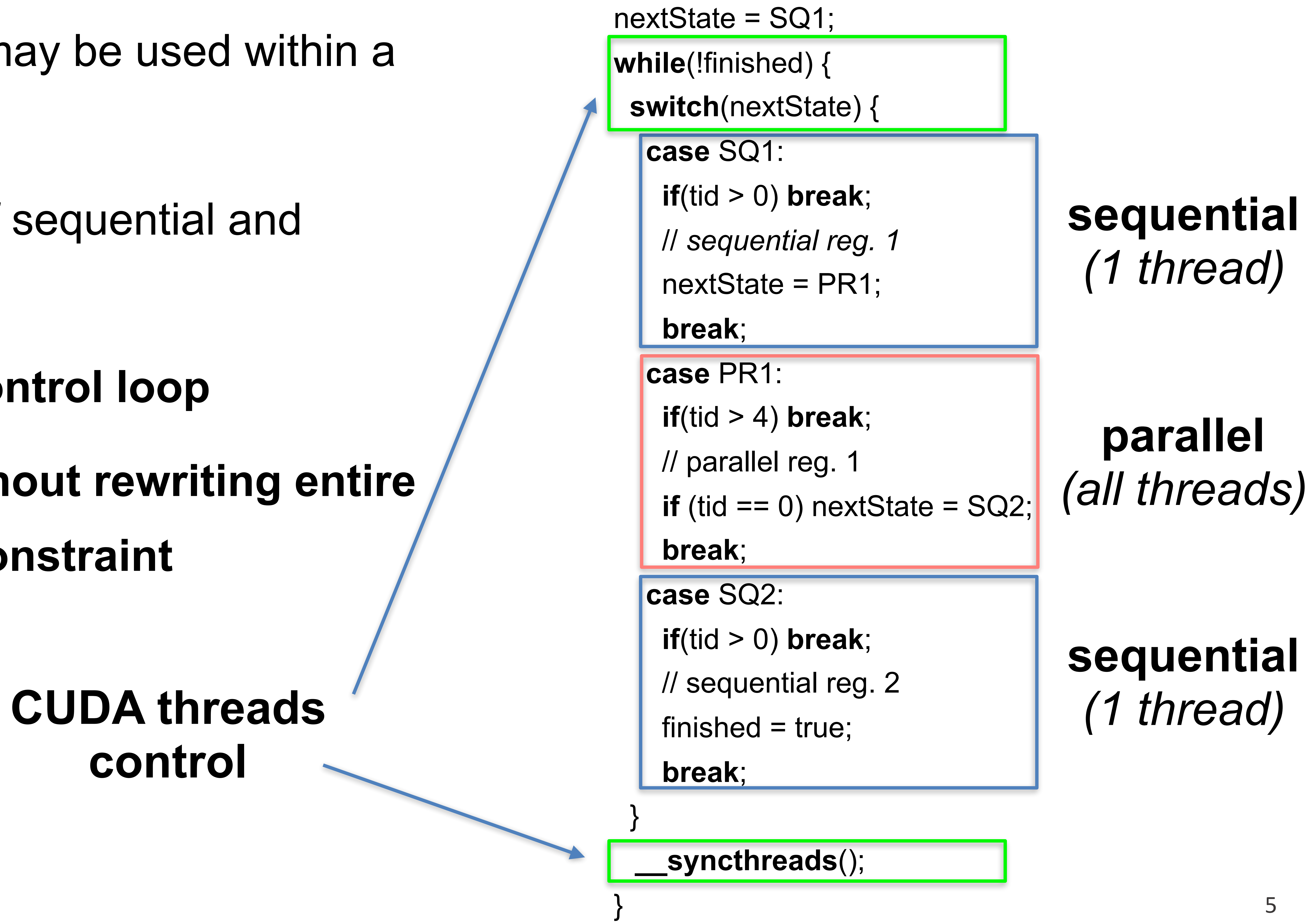
Basic OpenMP Implementation on NVIDIA GPUs

- Challenge: any OpenMP construct may be used within a target region
- This includes arbitrary sequences of sequential and parallel regions, tasks, locks, etc.
- General implementation scheme: **control loop**

```
#pragma omp target
{
 if(a[0]++ > K || b[1]++ < L) {
  #pragma omp parallel for
  for(int i = 0 ; i < K ; i++) {
   if(omp_get_thread_num() > 2) {
    #pragma omp simd
    for(int j = 0 ; j < L ; L++) { <S1> }
   } else {
    #pragma omp simd
    for(int j = 0 ; j < L ; L++) { <S2> }
   }
 } else {
  #pragma omp parallel for
  for(int i = 0 ; i < K ; i++) {
     <S3>
   }
 }
```

Basic OpenMP Implementation on NVIDIA GPUs

- Challenge: any OpenMP construct may be used within a target region
- This includes arbitrary sequences of sequential and parallel regions, tasks, locks, etc.
- General implementation scheme: **control loop**
- **Ease of integration into clang without rewriting entire C/C++ implementation is also a constraint**



```
nextState = SQ1;
while(!finished) {
  switch(nextState) {
    case SQ1:
      if(tid > 0) break;
      // sequential reg. 1
      nextState = PR1;
      break;
    case PR1:
      if(tid > 4) break;
      // parallel reg. 1
      if (tid == 0) nextState = SQ2;
      break;
    case SQ2:
      if(tid > 0) break;
      // sequential reg. 2
      finished = true;
      break;
  }
  __syncthreads();
}
```

sequential
(1 thread)

parallel
(all threads)

sequential
(1 thread)

CUDA threads control

First Naive Runs

Cuda Kernel Region	CUDA Runtime (usec)	Control Loop Runtime (usec)
Acceleration Calculation	3.2	712
Apply Boundary Acceleration	5.1	279
Position and Velocity Calculation	3.2	775 598
Kinematics and Monotonic Gradient Calculation	17	608 2546 1913
Monotonic Region Calculation	11	3760
Apply Material Properties to Regions	92	509 619 544

Two Missing Important Bits

- **Uncoalesced Accesses:**
  - By default, OpenMP schedules loops by contiguous chunks
  - Change default to schedule(static,1) assigns successive iterations to successive threads within same blocks
- **Tuning of number of blocks and block size per kernel**

After First Tuning

Cuda Kernel Region	CUDA Runtime (usec)	Control Loop Runtime (usec)
Acceleration Calculation	3.2	55
Apply Boundary Acceleration	5.1	43
Position and Velocity Calculation	3.2	54 45
Kinematics and Monotonic Gradient Calculation	17	511 211 140
Monotonic Region Calculation	11	365
Apply Material Properties to Regions	92	39 529 40

Occupancy / Register Allocation

- Many reasons:
  - A while loop with a switch inside may hit hard register allocation
  - In OpenMP 4.0 kernel parameters are passed as pointer to pointer
    - The kernel is allowed to do pointer arithmetic
    - This results in an additional register allocated for each parameter
    - Fixed by OpenMP 4.5 firstprivate-related rules
  - NVCC and LLVM backends for NVPTX are different:
    - nvcc uses libnvvm, which is shipped as a library
    - LLVM uses the open source code in the trunk
    - Different optimization strategies

Optimized Code Synthesis for **Combined Construct**

```
#pragma omp target teams distribute parallel for schedule(static,1) \
for( Index_t gnode=0 ; gnode<numNode ; ++gnode )
{
}
```

Compiler:

- Detect pragma combination
- Prove absence of nested pragmas
- Prove absence of function calls

```
for (int i = threadIdx.x + blockIdx.x * blockDim.x;
        i < n; i += blockDim.x * gridDim.x) {
  g_node = i;

  // codegen loop body
}
```

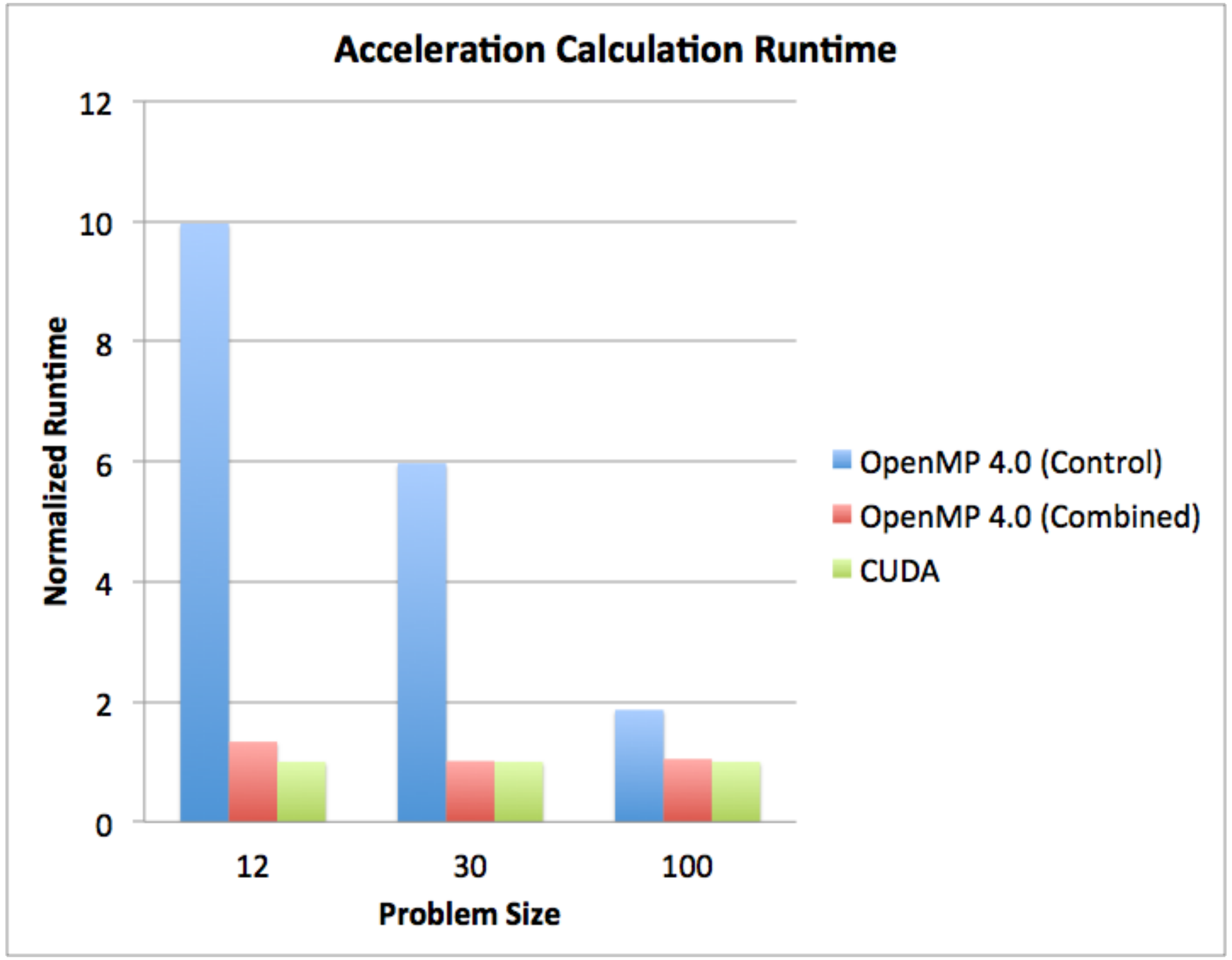
1-to-1 mapping of
CUDA grid to iteration space

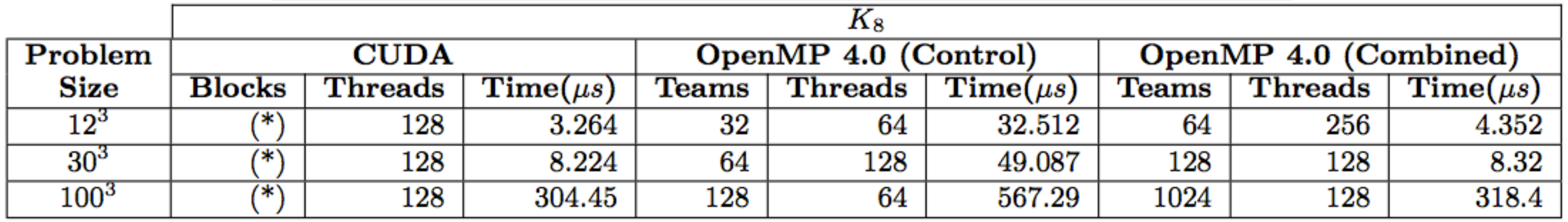
CUDA-style notation

Performance of combined Construct

Cuda Kernel Region	CUDA Runtime (usec)	Control Loop Runtime (usec)	% diff
Acceleration Calculation	3.2	4.3	35%
Apply Boundary Acceleration	5.1	4.8	-6%
Position and Velocity Calculation	3.2	4.8 4.1	178%
Kinematics and Monotonic Gradient Calculation	17	6.5 58 40	514%
Monotonic Region Calculation	11	15	36%
Apply Material Properties to Regions	92	3 314 3.1	247%

Small Kernels: Acceleration Calculation





	K_8								
Problem Size	CUDA			OpenMP 4.0 (Control)			OpenMP 4.0 (Combined)		
	Blocks	Threads	Time(μs)	Teams	Threads	Time(μs)	Teams	Threads	Time(μs)
12^3	(*)	128	3.264	32	64	32.512	64	256	4.352
30^3	(*)	128	8.224	64	128	49.087	128	128	8.32
100^3	(*)	128	304.45	128	64	567.29	1024	128	318.4

Large Kernels

- Reduce loop count in OpenMP 3.0 (45) to about 18 in OpenMP 4.0 (simple fusion)
- Direct correspondences with CUDA still hard to come by for complex loops.
- Ideal candidate: loops applying material properties

Cuda Kernel Region	CUDA Runtime (usec)	Control Loop Runtime (usec)
Apply Material Properties to Regions	**92**	**3** **314** **3.1**

1. **Fuse Loops**

Apply Material Properties to Regions	**92**	**525.6**

2. **Reorder values to have fewer divergent warps**

Apply Material Properties to Regions	**92**	**466.3**

3. **Loop over cells instead of regions**

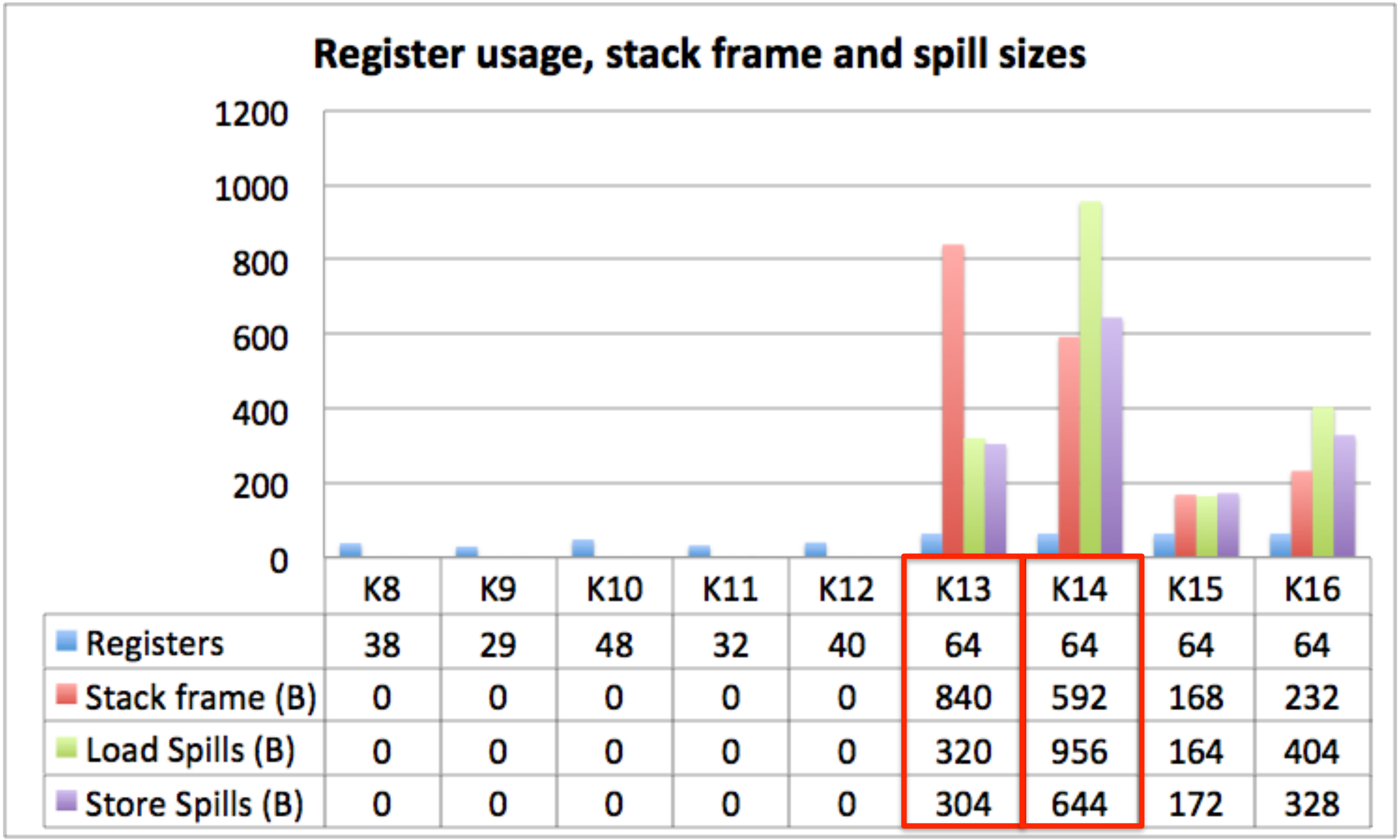
Apply Material Properties to Regions	**92**	**102.8**

Performance of Combined Construct

Cuda Kernel Region	CUDA Runtime (usec)	Control Loop Runtime (usec)	% diff
Acceleration Calculation	3.2	4.3	35%
Apply Boundary Acceleration	5.1	4.8	-6%
Position and Velocity Calculation	3.2	4.8 4.1	178%
Kinematics and Monotonic Gradient Calculation	17	6.5 58 40	514%
Monotonic Region Calculation	11	15	36%
Apply Material Properties to Regions	92	102.8	11%

PTXAS Report

Kernel ID	Kernel Name
K13	CalcKinematicsForElems
K14	CalcMonotonicQGradientsForElems



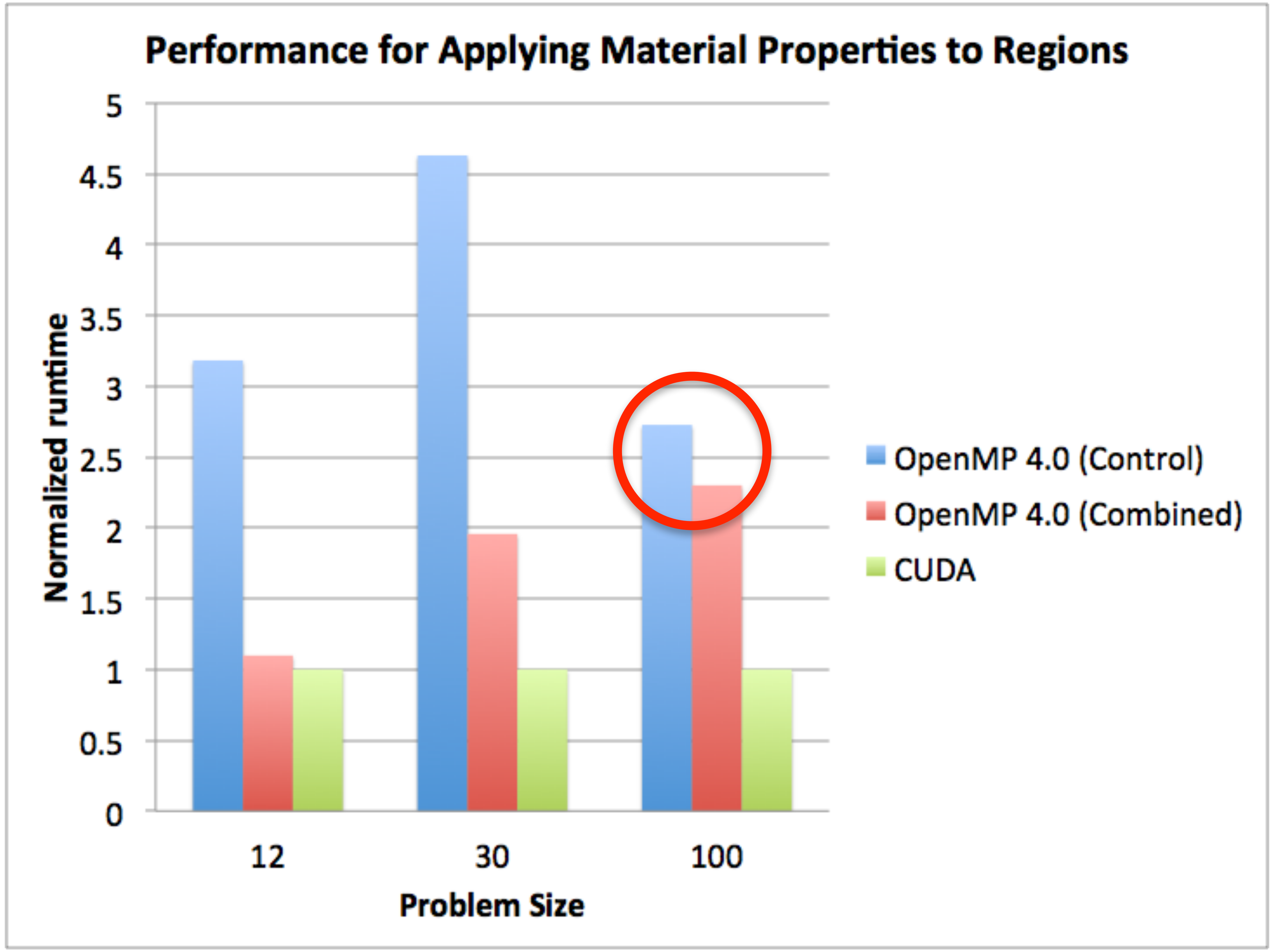
	K8	K9	K10	K11	K12	K13	K14	K15	K16
🟦 Registers	38	29	48	32	40	64	64	64	64
🟥 Stack frame (B)	0	0	0	0	0	840	592	168	232
🟩 Load Spills (B)	0	0	0	0	0	320	956	164	404
🟪 Store Spills (B)	0	0	0	0	0	304	644	172	328

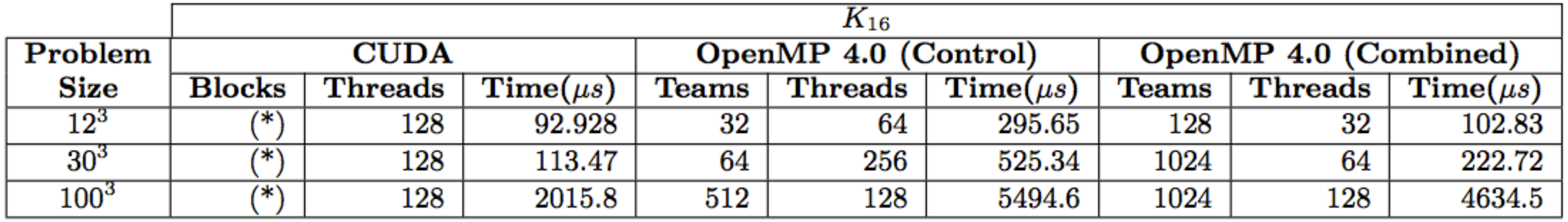
Conclusion

- Good performance can be achieved for simpler kernels
  - Requires optimized compiler synthesis
  - How many patterns do we need?
- More complex kernels may require hand tuning over baseline
  - Register allocation figure is of paramount importance
  - Use libnvvm’s code synthesis to improve register allocation?
  - Other factors like coalescing may play a relevant role in a “bad register allocation” situation

Fallback

Large Loop Performance by Problem Size





	K_{16}								
Problem Size	CUDA			OpenMP 4.0 (Control)			OpenMP 4.0 (Combined)		
	Blocks	Threads	Time(μs)	Teams	Threads	Time(μs)	Teams	Threads	Time(μs)
12^3	(*)	128	92.928	32	64	295.65	128	32	102.83
30^3	(*)	128	113.47	64	256	525.34	1024	64	222.72
100^3	(*)	128	2015.8	512	128	5494.6	1024	128	4634.5

Background

- Work by IBM's Advanced Compiler Technology team
- OpenMP 4.0 implementation based on **Clang/LLVM**® compilation toolchain
- Targets node with IBM® Power® processors plus Nvidia® GPUs
  - All tests on IBM 8247-42L system: Power 8 + Kepler K40m
- All tools available as open source
- IBM Proprietary OpenMP optimized implementation through Lightweight OpenMP library (**LOMP**)
  - **Lomp** only available for OpenPower nodes and other IBM processors
- Ongoing implementation, transitioning to OpenMP 4.5
  - Beta OpenMP 4.5 will be available to DoE Labs around mid-April