LULESH and OpenACC:
To Exascale and Beyond!!!

Shaden Smith [1,2] Peter Robinson [2]

[1]University of Minnesota
[2]Lawrence Livermore National Laboratory, Weapons and Complex Integration

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

August 21, 2013

1. Introduction and Motivations

2. OpenACC

3. Challenges

4. Methodologies and Results

5. Conclusions

Exascale Architectures

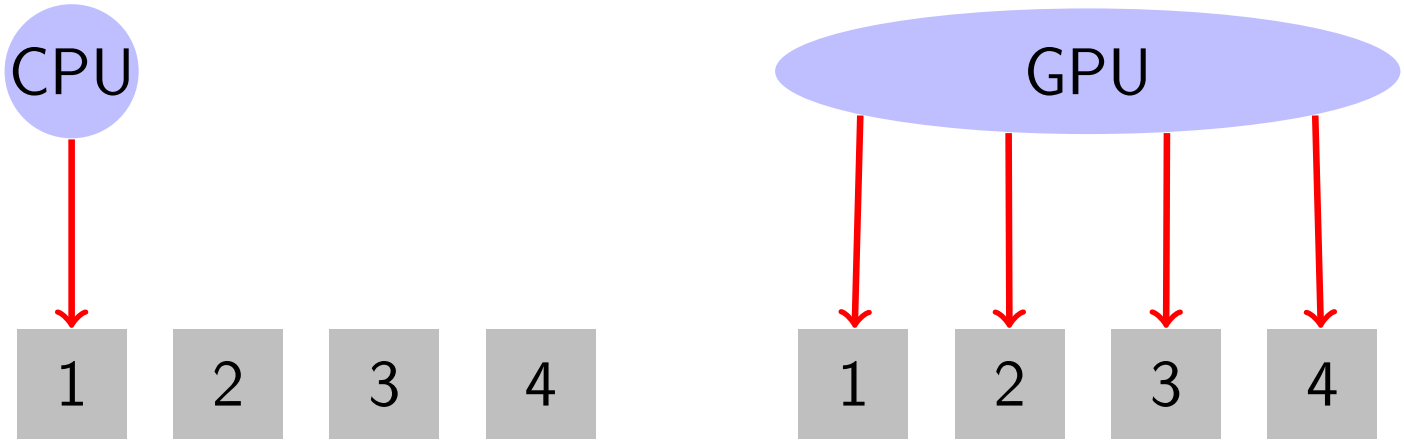
Heterogeneity

- Supercomputers will no longer have simple, homogeneous nodes with many CPU cores
- GPUs and other accelerators are dominating the horsepower of new systems

	Sequoia	**Titan**	**Tianhe-2**
PFLOPS	17.17	17.59	33.86
Architecture	BG/Q	AMD CPU + NVIDIA GPU	Intel CPU + MIC
Nodes/Cores	98.30K / 1.57M	18.68K / 0.56M	16.00K / 3.12M
Power	7.89MW	8.20MW	17.80MW

GPU Overview

- GPUs are massively parallel accelerators designed for graphics processing
- Very good at *stream processing*
  - Scan over a large list of data, doing identical math on each index
- The CPU and GPU do not share memory
  - The programmer must maintain copies on both



Proxy Applications

Motivation

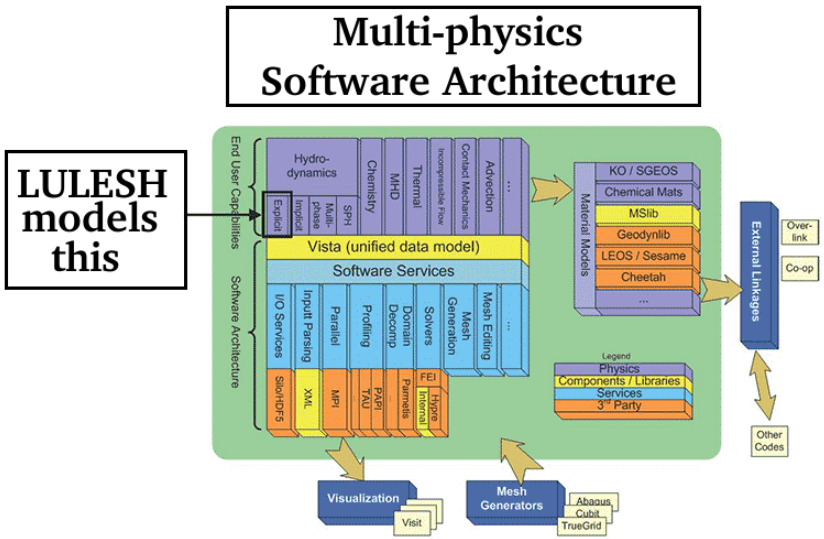
- Rewriting a large simulation code is a major investment
- Instead, extract a small but representative portion
- Can be modified and also released for public use
  - Great for hardware co-design!

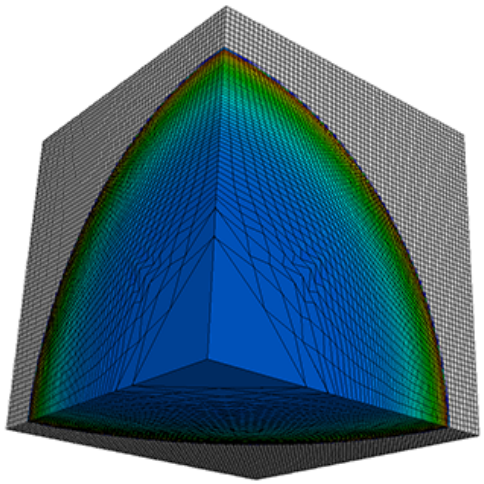
Proxy Apps

- AMG2013
- **LULESH**
- MCB
- UMT

LULESH Overview

- Data layout, memory access patterns, and computation are very similar to a typical multi-physics code's hydro kernel
- Only a few thousand lines of code, so it's easy to rewrite for new architectures and programming models





1. Introduction and Motivations

2. OpenACC

3. Challenges

4. Methodologies and Results

5. Conclusions

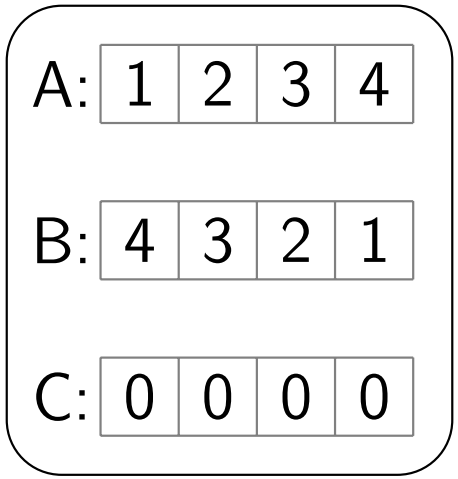
What is OpenACC?

- C/C++/Fortran API that supports offloading work to accelerator devices
- Uses pragmas to provide the compiler hints for parallel regions
  - Familiar interface for OpenMP programmers!

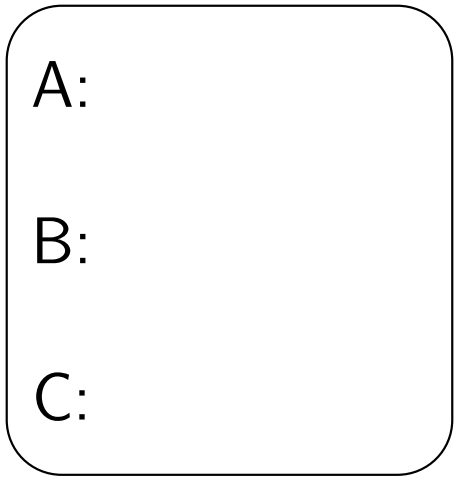
```
1  /* A, B, and C currently on CPU */
2  #pragma acc parallel loop copyin(A[0:N], \
3                                   B[0:N]) \
4                            copyout(C[0:N])
5  for(int i = 0; i < N; ++i) {
6    C[i] = A[i] * B[i];
7  }
```

OpenACC - Introduction

```
1  /* A, B, and C currently on CPU */
2  #pragma acc parallel loop copyin(A[0:N], \
3                                   B[0:N]) \
4                            copyout(C[0:N])
5  for(int i = 0; i < N; ++i) {
6    C[i] = A[i] * B[i];
7  }
```



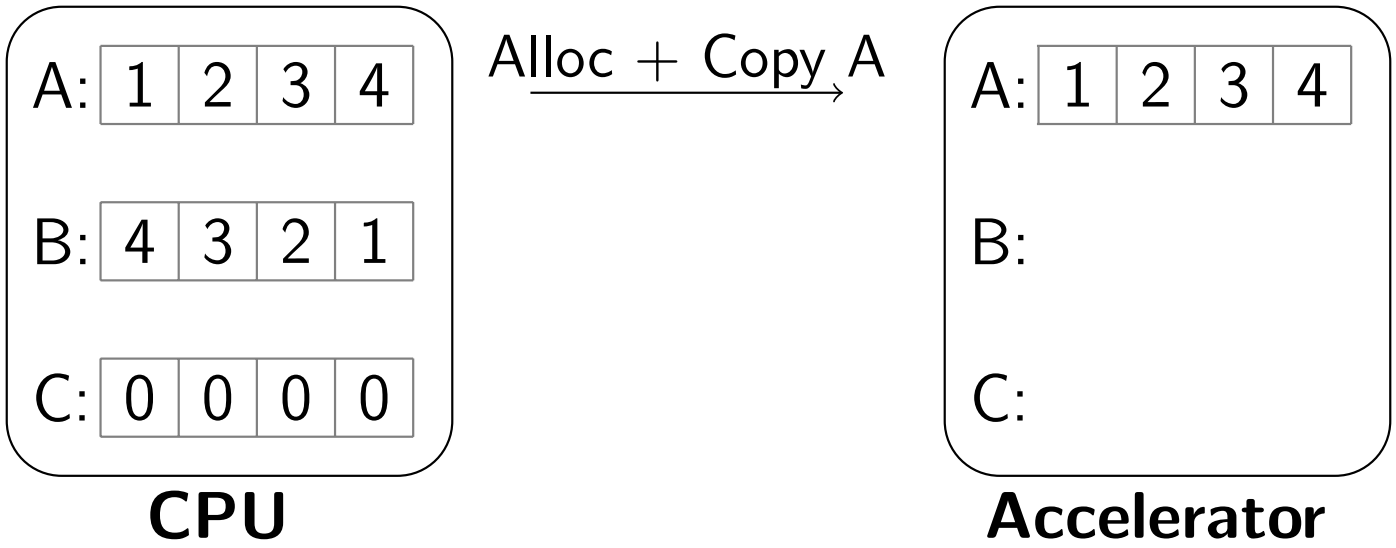
CPU



Accelerator

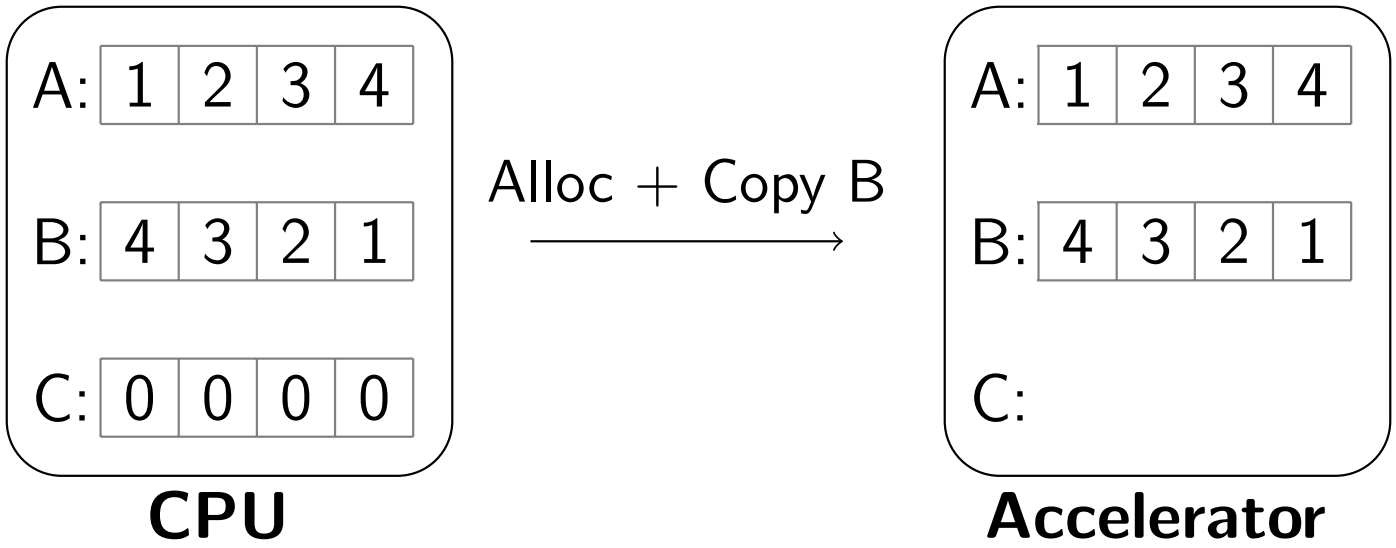
OpenACC - Introduction

```
1  /* A, B, and C currently on CPU */
2  #pragma acc parallel loop copyin(A[0:N], \
3                                   B[0:N]) \
4                            copyout(C[0:N])
5  for(int i = 0; i < N; ++i) {
6    C[i] = A[i] * B[i];
7  }
```



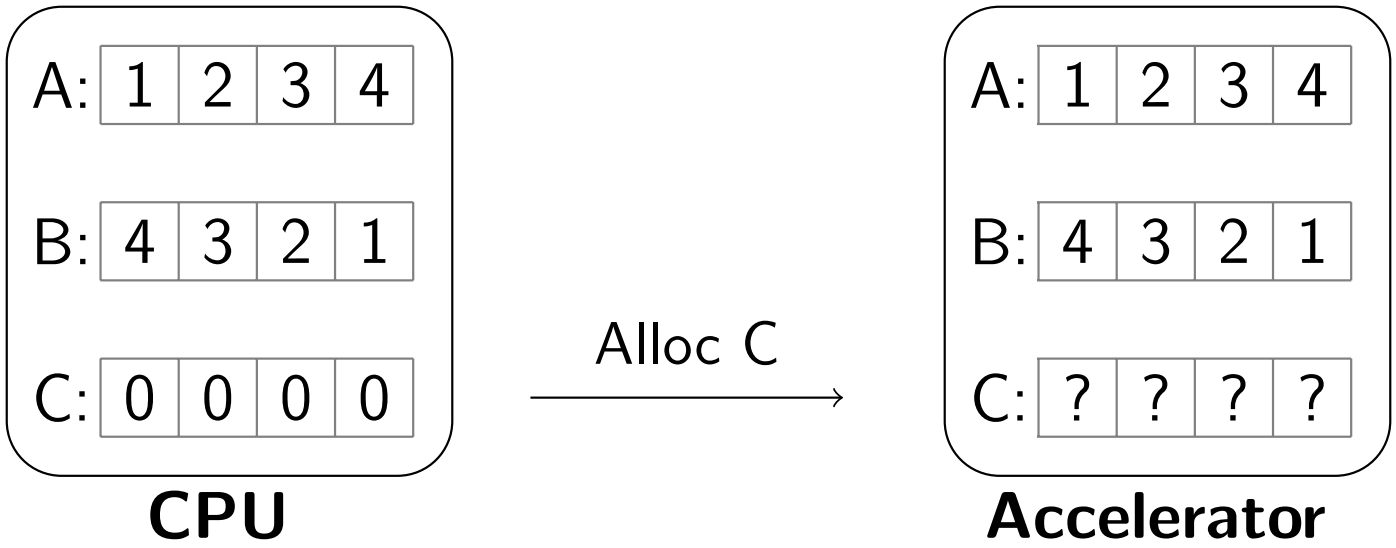
OpenACC - Introduction

```
1  /* A, B, and C currently on CPU */
2  #pragma acc parallel loop copyin(A[0:N], \
3                                   B[0:N]) \
4                            copyout(C[0:N])
5  for(int i = 0; i < N; ++i) {
6    C[i] = A[i] * B[i];
7  }
```



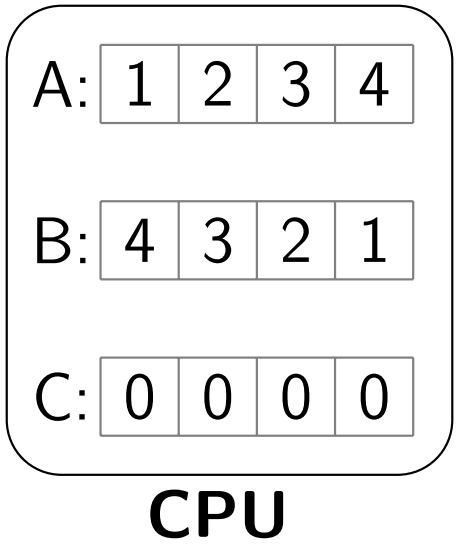
OpenACC - Introduction

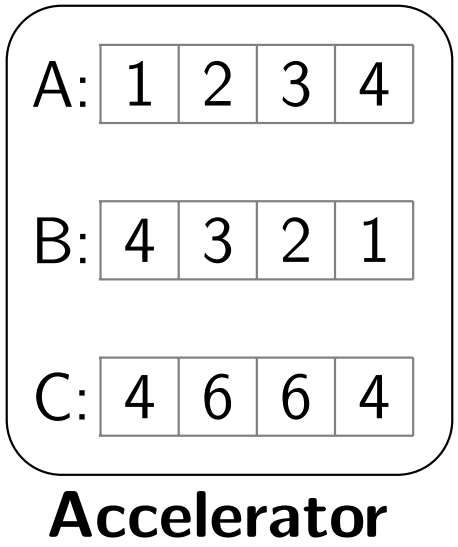
```
1  /* A, B, and C currently on CPU */
2  #pragma acc parallel loop copyin(A[0:N], \
3                                   B[0:N]) \
4                            copyout(C[0:N])
5  for(int i = 0; i < N; ++i) {
6    C[i] = A[i] * B[i];
7  }
```



OpenACC - Introduction

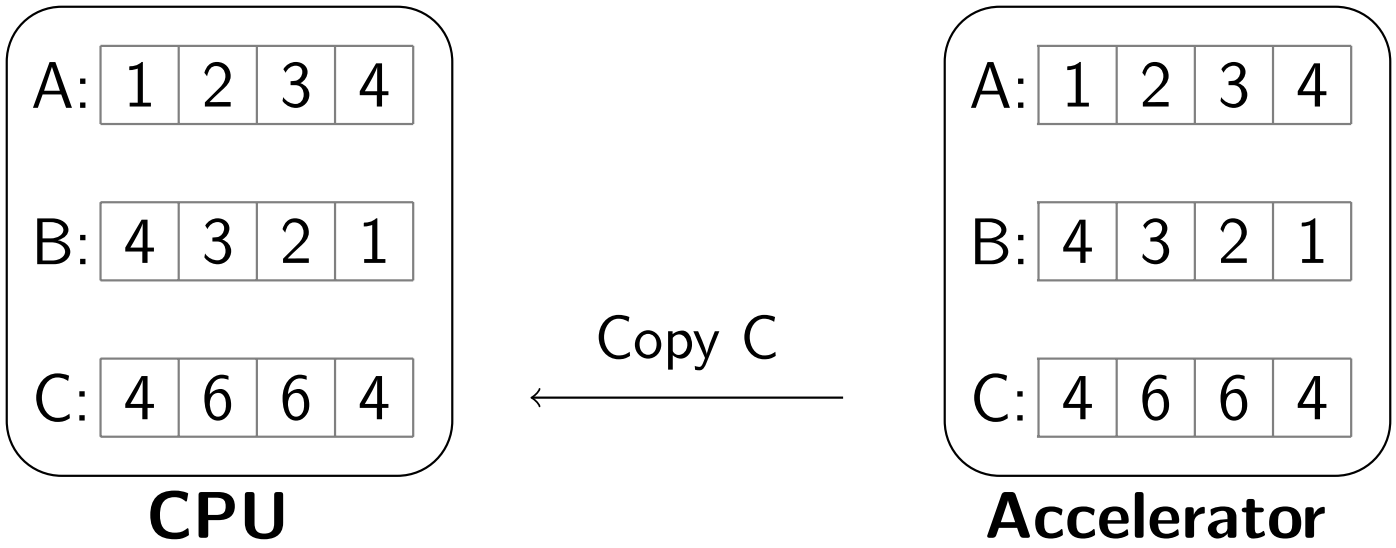
```
1  /* A, B, and C currently on CPU */
2  #pragma acc parallel loop copyin(A[0:N], \
3                                   B[0:N]) \
4                            copyout(C[0:N])
5  for(int i = 0; i < N; ++i) {
6    C[i] = A[i] * B[i];
7  }
```





OpenACC - Introduction

```
1  /* A, B, and C currently on CPU */
2  #pragma acc parallel loop copyin(A[0:N], \
3                                   B[0:N]) \
4                            copyout(C[0:N])
5  for(int i = 0; i < N; ++i) {
6    C[i] = A[i] * B[i];
7  }
```



Data Regions

- Data regions provide a means of specifying memory transfers
- Minimizing data movement between the CPU and accelerator is essential for performance

```
1  /* A, B, and C allocated on CPU */
2  #pragma acc data copyin(A[0:N], \
3                          B[0:N]) \
4                  copyout(C[0:N])
5  {
6    /* A, B, and C are now on accelerator */
7    compute_C(A,B,C);
8    compute_more_C(A,B,C);
9  }
10 /* C has now been updated on CPU */
```

OpenACC - Availability

Compiler Support

- Three compilers have implementations of OpenACC
  - PGI, CAPS, Cray
- Our code has only been tested with PGI thus far

LLNL Support

- edge and rzgpu both have pgi-accelerator available
  - Compile on edge84 and rzgpu2

1. Introduction and Motivations

2. OpenACC

3. Challenges

4. Methodologies and Results

5. Conclusions

Implicit Data Regions

- When functions are called from within a data region, the programmer must be aware of which memory is found on the accelerator
- It's easy to forget where your data is and instead access junk

```
1  #pragma acc data copyin(A[0:N], \
2                          B[0:N]) \
3                  copyout(C[0:N])
4  {
5    compute_C(A,B,C);
6    print_intermediate_results(C); /* OUCH! */
7    compute_more_C(A,B,C);
8  }
```

Thread-Local Arrays

- The OpenACC standard currently doesn't say what to do with local arrays in accelerated regions
- As of pgcc v13.6, these are treated as a shared resource among threads

Before

```
1   for(Index_t i = 0; i < N; ++i) {
2     Real_t scratch[4];
3     for(Index_t j = 0; j < 4; ++j) {
4       scratch[j] = x[i*4 + j];
5     }
6
7
8
9
10
11
12    /* do work */
13  }
```

After

Thread-Local Arrays

- The OpenACC standard currently doesn't say what to do with local arrays in accelerated regions
- As of pgcc v13.6, these are treated as a shared resource among threads

Before

```
1  for(Index_t i = 0; i < N; ++i) {
2    Real_t scratch[4];
3    for(Index_t j = 0; j < 4; ++j) {
4      scratch[j] = x[i*4 + j];
5    }
6
7
8
9
10
11
12   /* do work */
13 }
```

After

```
#pragma acc parallel loop copy(x[0:N*4])
for(Index_t i = 0; i < N; ++i) {
  Real_t scratch0;
  Real_t scratch1;
  Real_t scratch2;
  Real_t scratch3;

  scratch0 = x[i*4 + 0];
  scratch1 = x[i*4 + 1];
  scratch2 = x[i*4 + 2];
  scratch3 = x[i*4 + 3];
  /* do work */
}
```

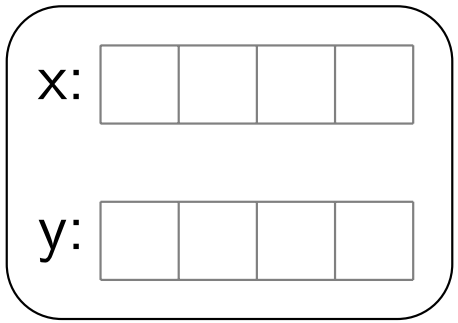
Runtime Errors

- Class members are often extracted before entering a data region
  - Currently you cannot access members within a pragma
- If these are not made `volatile`, they will be optimized away

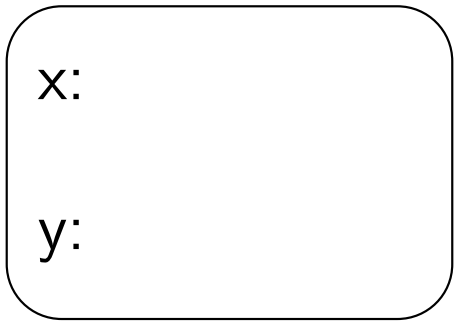
```
1  volatile Real_t *x = domain.x();
2           Real_t *y = domain.y(); /* y is optimized away */
3  #pragma acc data copyin(x[0:N], \
4                          y[0:N]) /* runtime error! */
5  {
6    accelerated_physics(domain);
7  }
```

Compiler Optimizations

```
1  volatile Real_t *x = domain.x();
2           Real_t *y = domain.y(); /* y is optimized away */
3  #pragma acc data copyin(x[0:N], \
4                          y[0:N]) /* runtime error! */
5  {
6    accelerated_physics(domain);
7  }
```



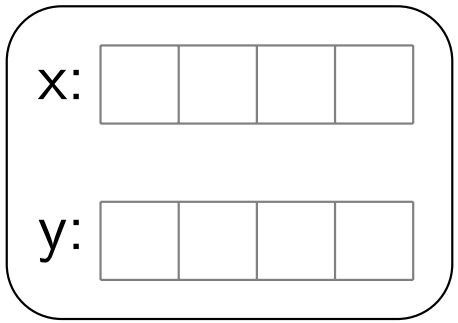
CPU



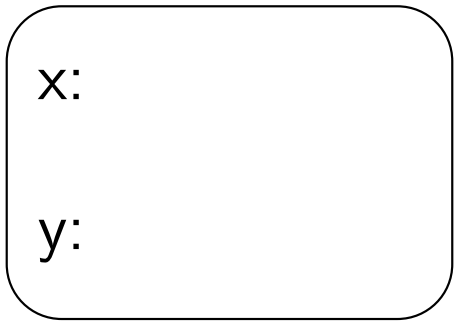
Accelerator

Compiler Optimizations

```
1  volatile Real_t *x = domain.x();
2           Real_t *y = domain.y(); /* y is optimized away */
3  #pragma acc data copyin(x[0:N], \
4                          y[0:N]) /* runtime error! */
5  {
6    accelerated_physics(domain);
7  }
```



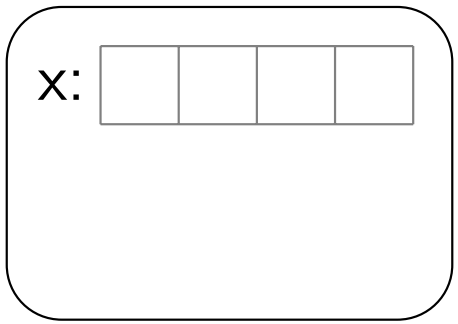
CPU



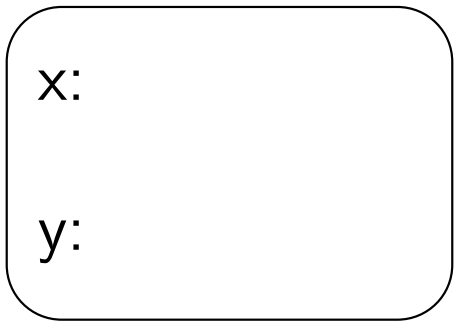
Accelerator

Compiler Optimizations

```
1  volatile Real_t *x = domain.x();
3  #pragma acc data copyin(x[0:N], \
4                          y[0:N]) /* runtime error! */
5  {
6    accelerated_physics(domain);
7  }
```



CPU



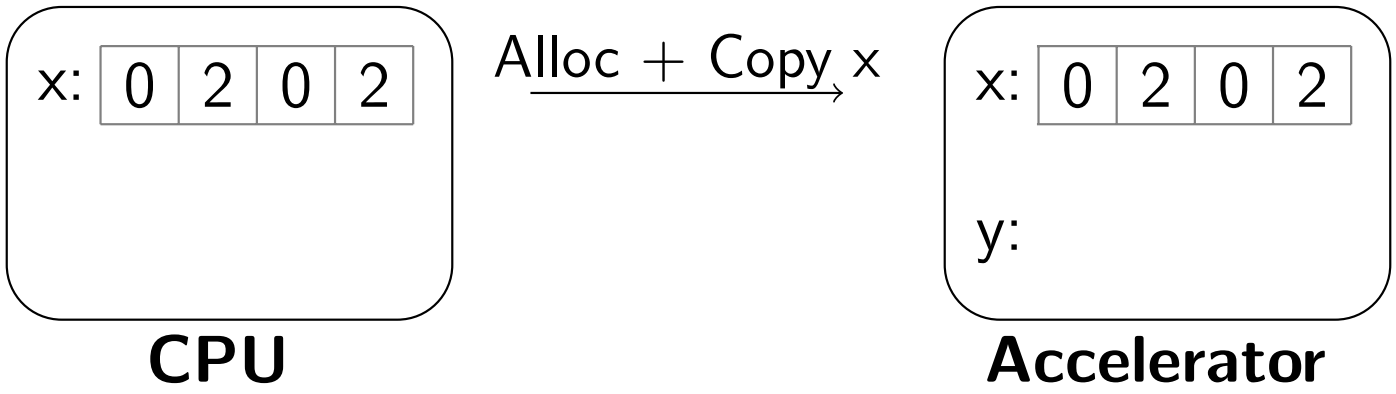
Accelerator

Compiler Optimizations

```
1  volatile Real_t *x = domain.x();
3  #pragma acc data copyin(x[0:N], \
4                          y[0:N]) /* runtime error! */
5  {
6    accelerated_physics(domain);
7  }
```

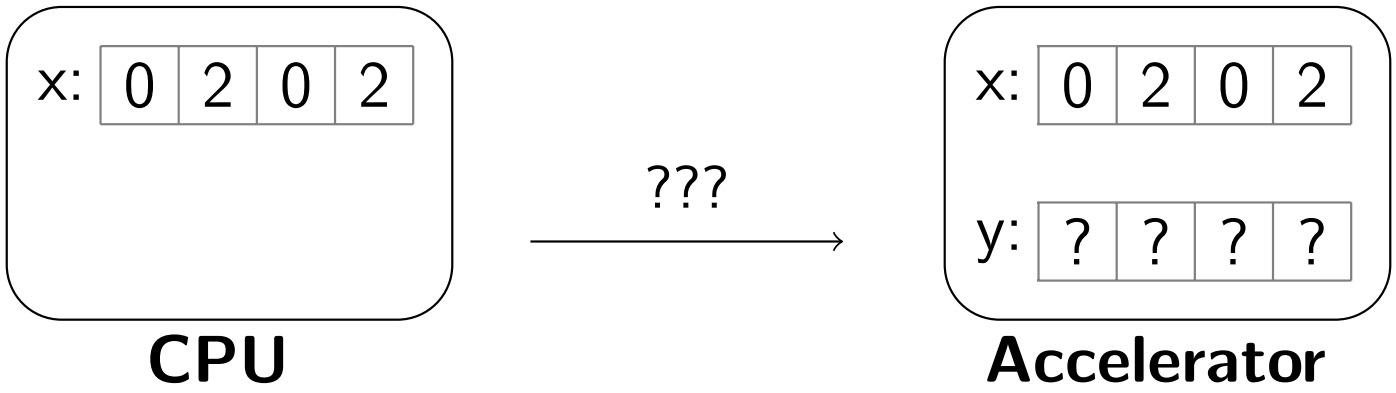
Compiler Optimizations

```
1  volatile Real_t *x = domain.x();
3  #pragma acc data copyin(x[0:N], \
4                          y[0:N]) /* runtime error! */
5  {
6    accelerated_physics(domain);
7  }
```



Compiler Optimizations

```
1  volatile Real_t *x = domain.x();
3  #pragma acc data copyin(x[0:N], \
4                          y[0:N]) /* runtime error! */
5  {
6    accelerated_physics(domain);
7  }
```



1. Introduction and Motivations

2. OpenACC

3. Challenges

4. Methodologies and Results

5. Conclusions

Experimental Methodologies

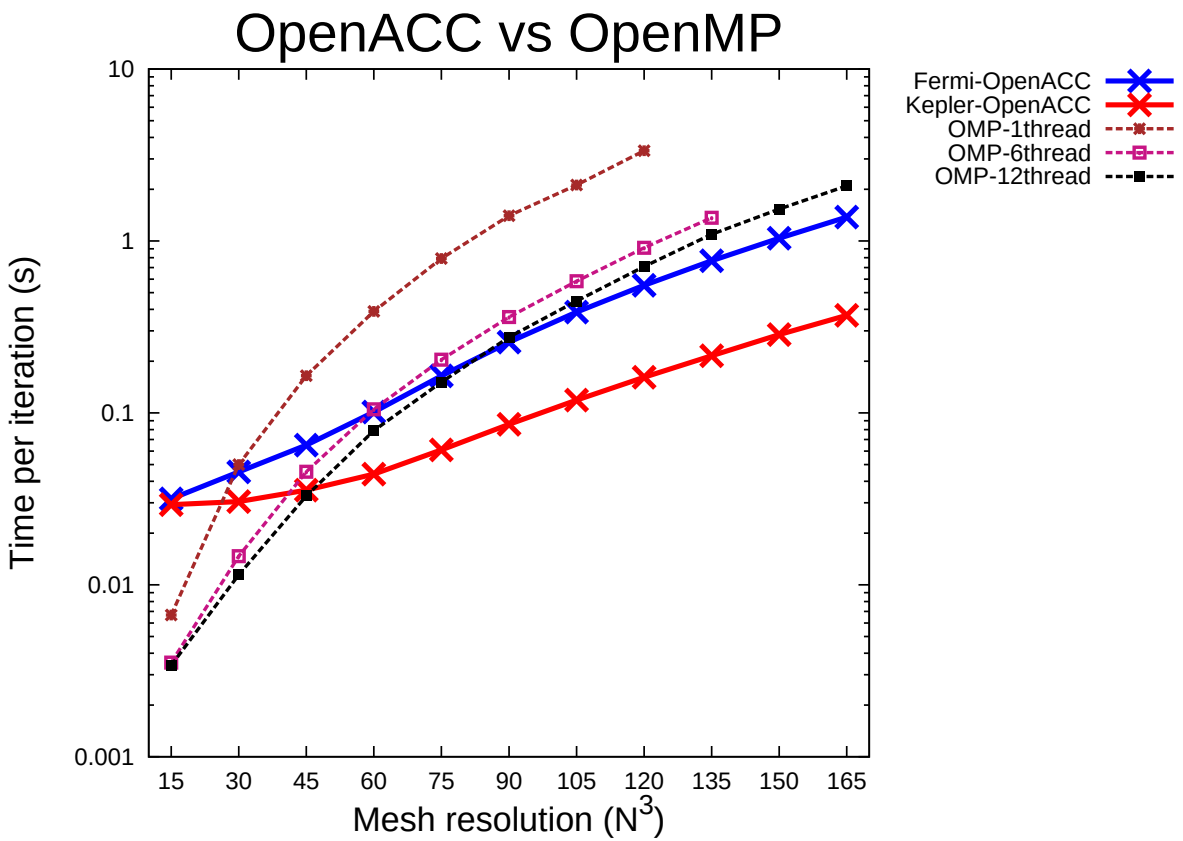
Completed Tasks

- OpenACC rewrite of LULESH
- Also supports MPI
- Falls back to OpenMP if not compiled with OpenACC
  - This lets us measure the runtime effects of the loop unrolling and other changes we made

Measurements of Interest

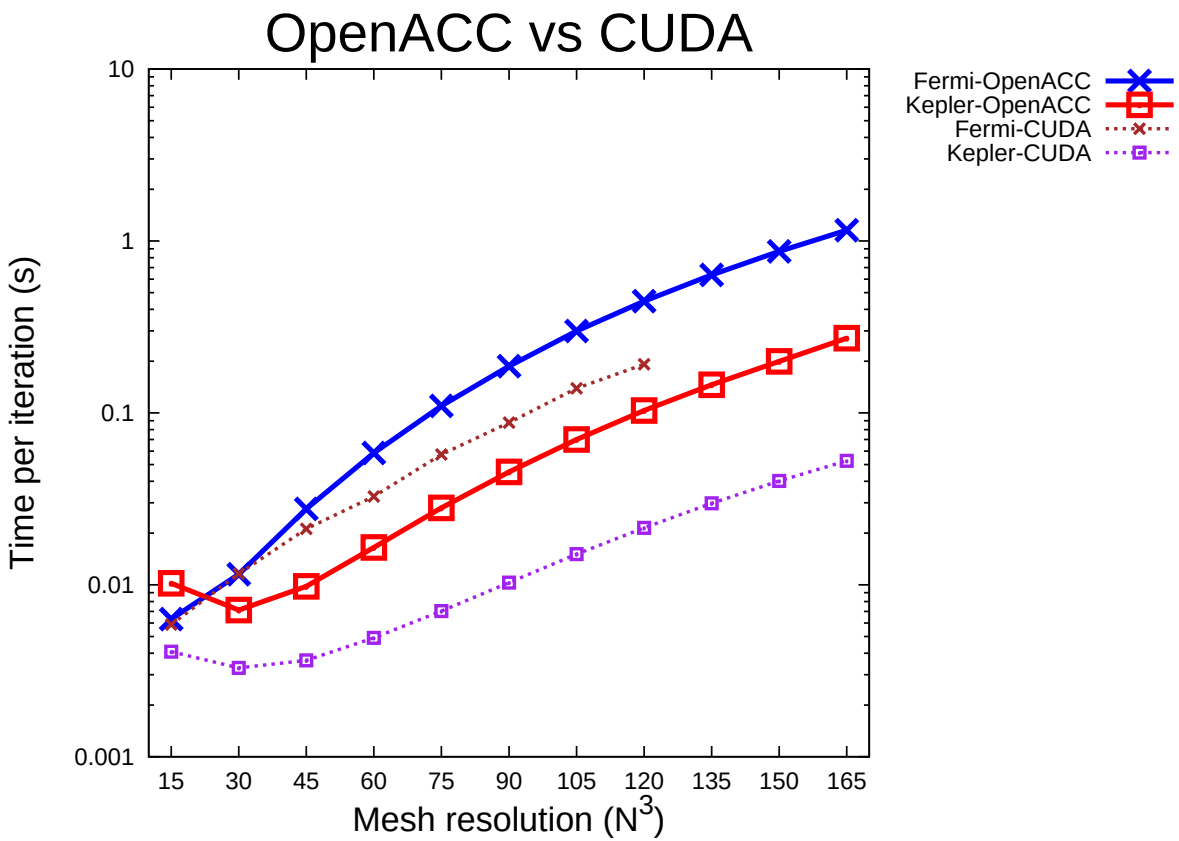
- OpenACC vs OpenMP
- OpenACC vs CUDA
- Weak scaling
- Strong scaling

Runtime Comparisons



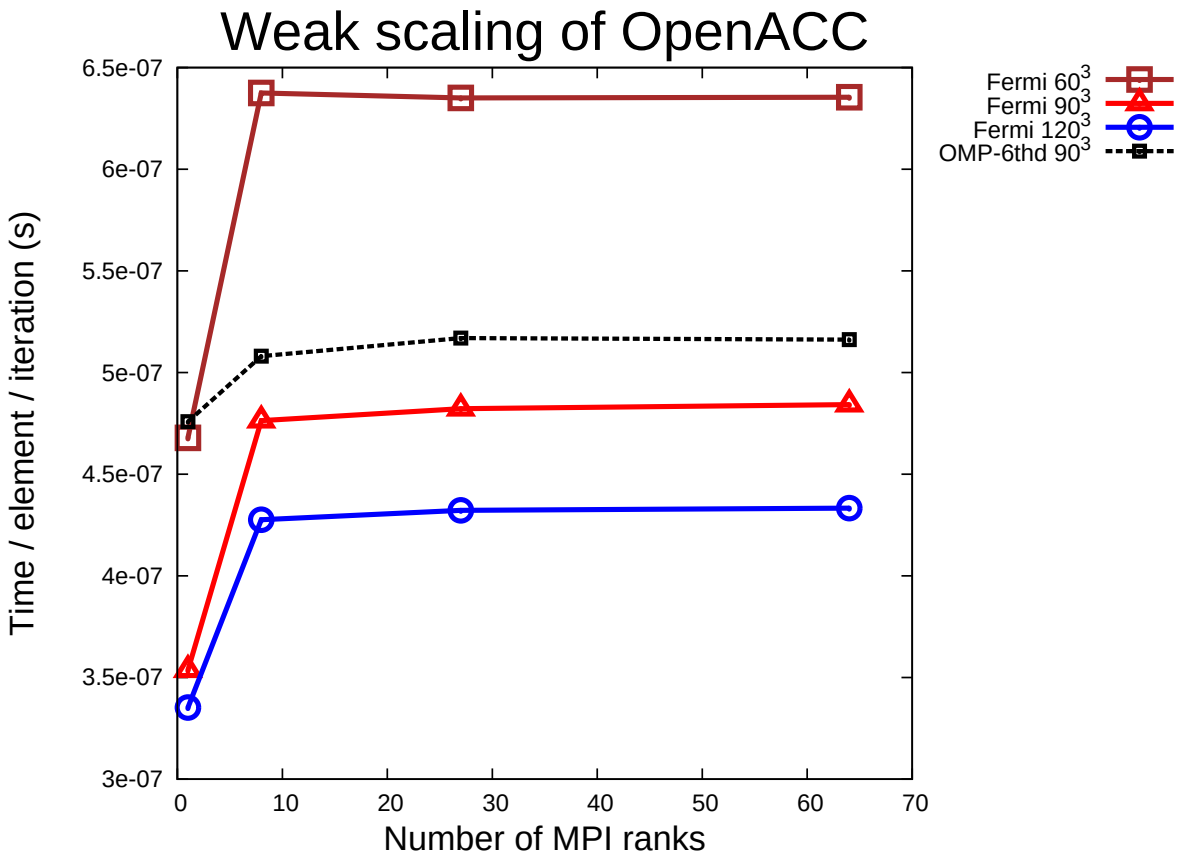
- OpenMP times were taken using up to 12 threads on a dual hex-core Intel Westmere system

Runtime Comparisons



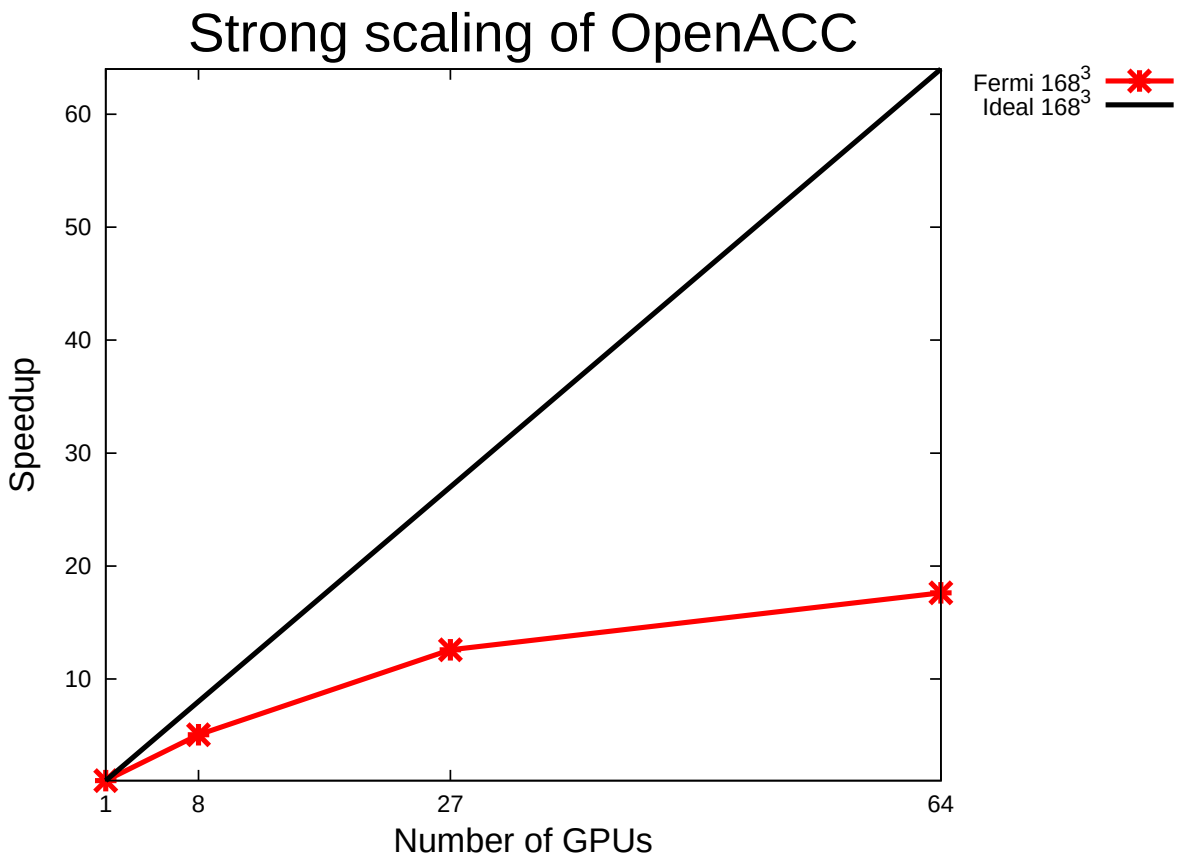
- We used a single, balanced region to emulate the computations done by the CUDA version of LULESH

Scaling Study



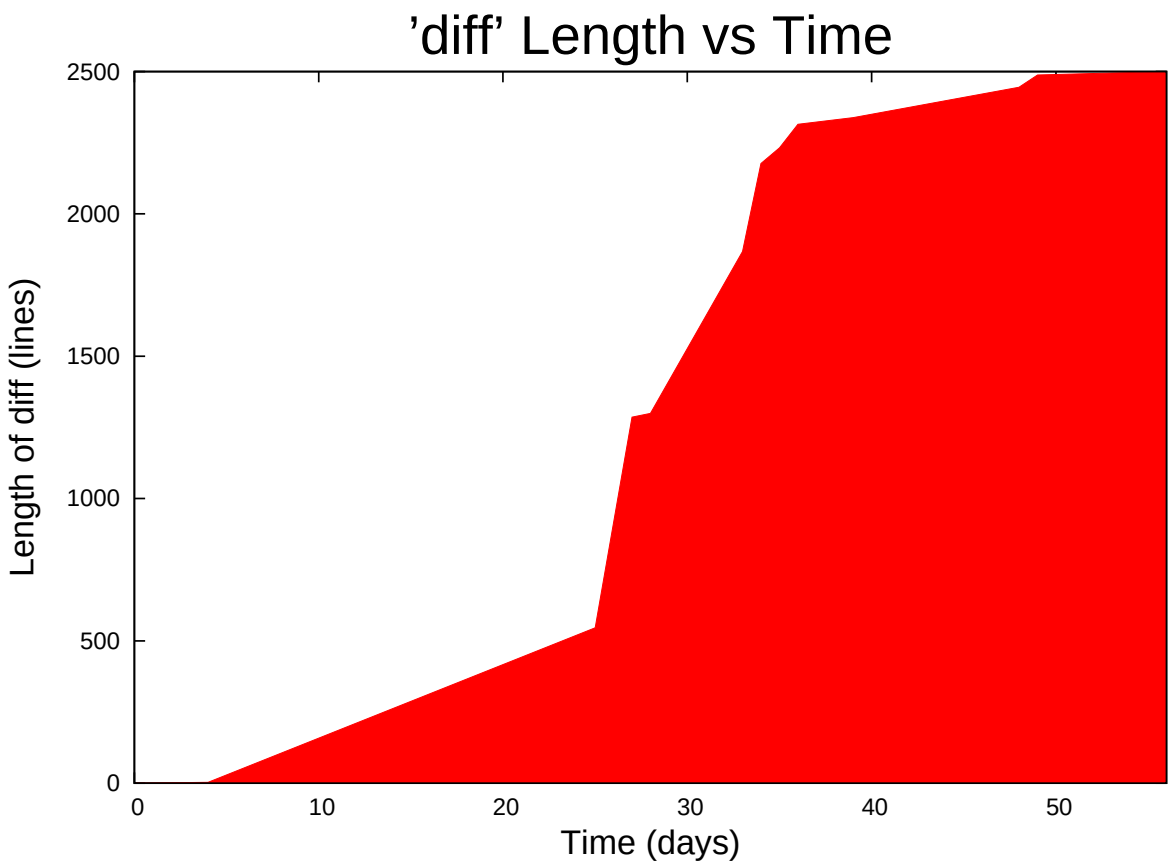
- For simplicity, LULESH's decomposition requires scaling with a cubic number of processes
- Weak scaling works well once hardware is fully saturated

Scaling Study



- Strong scaling is difficult due to decomposition and the large GPU overhead for small problem sizes

Programmability



- Due to large codebase changes (e.g. loop unrolling) large commits were often necessary

1. Introduction and Motivations

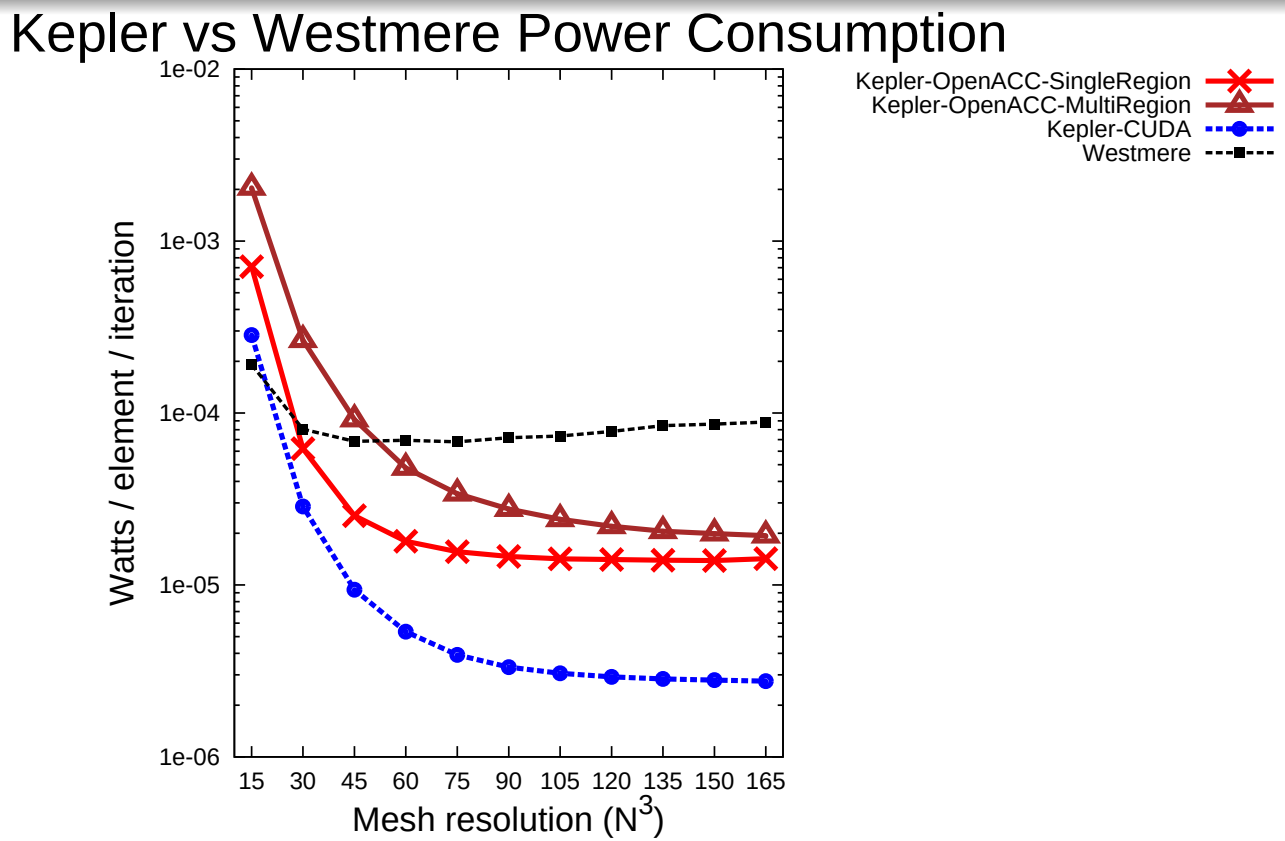
2. OpenACC

3. Challenges

4. Methodologies and Results

5. Conclusions

OpenACC Evaluation



Is OpenACC cost effective?

- Power is a major concern for future HPC systems
- Kepler K20Xm TDP: 235W
- Westmere Xeon E7 TDP: 95W * 2 sockets

OpenACC Evaluation

Do we recommend OpenACC?

- In the future, possibly
  - Let the standard and implementations mature first
  - Right now the required code changes are too expensive
- What about OpenMP v4?
  - The new OpenMP standard supports SIMD constructs as well
  - OpenACC is intended to be merged with OpenMP

Thank you!