OpenACC Parallelization and Optimization of NAS Parallel Benchmarks

Presented by Rengan Xu
GTC 2014, S4340
03/26/2014

Rengan Xu, Xiaonan Tian, Sunita Chandrasekaran,
Yonghong Yan, Barbara Chapman
HPC Tools group (http://web.cs.uh.edu/~hpctools/)
Department of Computer Science
University of Houston

Outline

- Motivation
- Overview of OpenACC and NPB benchmarks
- Parallelization and Optimization Techniques
- Parallelizing NPB Benchmarks
- Performance Evaluation
- Conclusion and Future Work

OpenUH
Where no research compiler has gone before

Motivation

- NPB is the benchmark close to real applications
- Parallelization techniques for improving performance

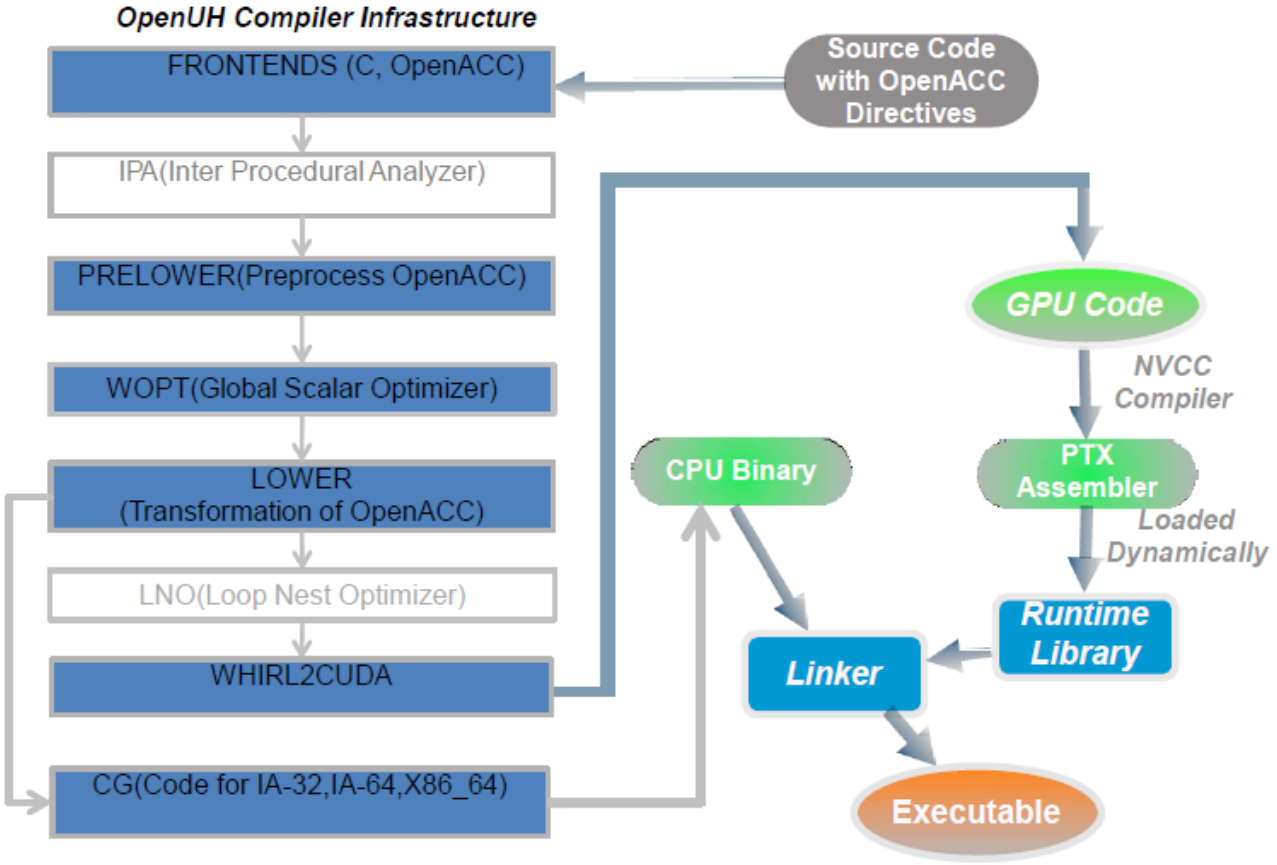
Overview of OpenACC

- Standard, a high-level directive-based programming model for accelerators
  - OpenACC 2.0 released late 2013
- Data Directive: copy/copyin/copyout/……
- Data Synchronization directive
  - update
- Compute Directive
  - Parallel: more control to the user
  - Kernels: more control to the compiler
- Three levels of parallelism
  - Gang
    - Worker
      - Vector

OpenUH
Where no research compiler has gone before

OpenUH: An Open Source OpenACC Compiler

- Link: http://web.cs.uh.edu/~openuh/



OpenUH
Where no research compiler has gone before

NAS Parallel Benchmarks (NPB)

- Well recognized for evaluating current and emerging multi-core/many-core hardware architectures
- 5 parallel kernels
  - IS, EP, CG, MG and FT
- 3 simulated computational fluid dynamics (CFD) applications
  - LU, SP and BT
- Different problem sizes
  - Class S: small for quick test purpose
  - Class W: workstation size
  - Class A: standard test problem
  - Class E: largest test problem

OpenUH
Where no research compiler has gone before

Steps to parallelize an application

- Profile to find the hotspot
- Analyze compute intensive loops to make it parallelizable
- Add compute directive to these loops
- Add data directive to manage data motion and synchronization
- Optimize data structure and array access pattern
- Apply Loop scheduling tuning
- Apply other optimizations, e.g. async and cache

OpenUH
Where no research compiler has gone before

Parallelization and Optimization Techniques

- Array privatization
- Loop scheduling tuning
- Memory coalescing optimization
- Scalar substitution
- Loop fission and unrolling
- Data motion optimization
- New algorithm adoption

OpenUH
Where no research compiler has gone before

Array Privatization

Before array privatization

(has data race)

```
#pragma acc kernels
for(k=0; k<=grid_points[2]-1; k++){
   for(j=0; j<grid_points[1]-1; j++){
      for(i=0; i<grid_points[0]-1; i++){
         for(m=0; m<5; m++){
            rhs[j][i][m] = forcing[k][j][i][m];
         }
      }
   }
}
```

After array privatization

(no data race, increased memory)

```
#pragma acc kernels
for(k=0; k<=grid_points[2]-1; k++){
   for(j=0; j<grid_points[1]-1; j++){
      for(i=0; i<grid_points[0]-1; i++){
         for(m=0; m<5; m++){
            rhs[k][j][i][m] = forcing[k][j][i][m];
         }
      }
   }
}
```

OpenUH
Where no research compiler has gone before

Loop Scheduling Tuning

Before tuning

```
#pragma acc kernels
for(k=0; k<=grid_points[2]-1; k++){
   for(j=0; j<grid_points[1]-1; j++){
      for(i=0; i<grid_points[0]-1; i++){
         for(m=0; m<5; m++){
            rhs[k][j][i][m] = forcing[k][j][i][m];
         }
      }
   }
}
```

After tuning

```
#pragma acc kernels loop gang
for(k=0; k<=grid_points[2]-1; k++){
  #pragma acc loop worker
   for(j=0; j<grid_points[1]-1; j++){
      #pragma acc loop vector
      for(i=0; i<grid_points[0]-1; i++){
         for(m=0; m<5; m++){
            rhs[k][j][i][m] = forcing[k][j][i][m];
         }
      }
   }
}
```

OpenUH
Where no research compiler has gone before

Memory Coalescing Optimization

Non-coalesced memory access

```
#pragma acc kernels loop gang
for(j=1; j <= gp12; j++){
  #pragma acc loop worker
   for(i=1; I <= gp02; i++){
      #pragma acc loop vector
      for(k=0; k <= ksize; k++){
           fjacZ[0][0][k][i][j] = 0.0;
      }
   }
}
```

Coalesced memory access (loop interchange)

```
#pragma acc kernels loop gang
for(k=0; k <= ksize; k++){
  #pragma acc loop worker
   for(i=1; i<= gp02; i++){
      #pragma acc loop vector
      for(j=1; j <= gp12; j++){
           fjacZ[0][0][k][i][j] = 0.0;
      }
   }
}
```

OpenUH
Where no research compiler has gone before

Memory Coalescing Optimization

Non-coalescing memory access

```
#pragma acc kernels loop gang
for(k=0; k<=grid_points[2]-1; k++){
  #pragma acc loop worker
   for(j=0; j<grid_points[1]-1; j++){
     #pragma acc loop vector
     for(i=0; i<grid_points[0]-1; i++){
        for(m=0; m<5; m++){
           rhs[k][j][i][m] = forcing[k][j][i][m];
         }
     }
   }
}
```

Coalesced memory access (change data layout)

```
#pragma acc kernels loop gang
for(k=0; k<=grid_points[2]-1; k++){
  #pragma acc loop worker
   for(j=0; j<grid_points[1]-1; j++){
     #pragma acc loop vector
     for(i=0; i<grid_points[0]-1; i++){
        for(m=0; m<5; m++){
           rhs[m][k][j][i] = forcing[m][k][j][i];
        }
     }
   }
}
```

OpenUH
Where no research compiler has gone before

Scalar Substitution

Before scalar substitution

```
#pragma acc kernels loop gang
for(k=0; k<=grid_points[2]-1; k++){
  #pragma acc loop worker
   for(j=0; j<grid_points[1]-1; j++){
      #pragma acc loop vector
      for(i=0; i<grid_points[0]-1; i++){
         for(m=0; m<5; m++){
            rhs[m][k][j][i] = forcing[k][j][i][m];
         }
      }
   }
}
```

After scalar substitution

```
#pragma acc kernels loop gang
for(k=0; k<=gp2-1; k++){
  #pragma acc loop worker
   for(j=0; j<gp1-1; j++){
      #pragma acc loop vector
      for(i=0; i<gp0-1; i++){
         for(m=0; m<5; m++){
            rhs[m][k][j][i] = forcing[m][k][j][i];
         }
      }
   }
}
```

OpenUH
Where no research compiler has gone before

Loop Unrolling

Before loop unrolling

```
#pragma acc kernels loop gang
for(k=0; k<=gp21; k++){
  #pragma acc loop worker
   for(j=0; j<gp1-1; j++){
     #pragma acc loop vector
     for(i=0; i<gp0-1; i++){
        for(m=0; m<5; m++){
           rhs[m][k][j][i] = forcing[m][k][j][i];
         }
     }
  }
}
```

After loop unrolling

```
#pragma acc kernels loop gang
for(k=0; k<=gp2-1; k++){
  #pragma acc loop worker
   for(j=0; j<gp1-1; j++){
     #pragma acc loop vector
     for(i=0; i<gp0-1; i++){
           rhs[0][k][j][i] = forcing[0][k][j][i];
           rhs[1][k][j][i] = forcing[1][k][j][i];
           rhs[2][k][j][i] = forcing[2][k][j][i];
           rhs[3][k][j][i] = forcing[3][k][j][i];
           rhs[4][k][j][i] = forcing[4][k][j][i];
     }
  }
}
```

OpenUH
Where no research compiler has gone before

Loop Fission

Before loop fission

```
# pragma acc parallel loop gang
for(i3 = 1; i3 < n3 -1; i3 ++) {
  # pragma acc loop worker
  for (i2 = 1; i2 < n2 -1; i2 ++) {
    # pragma acc loop vector
    for (i1 = 0; i1 < n1; i1 ++) {
      …
    }
    # pragma acc loop vector
    for (i1 = 1; i1 < n1 -1; i1 ++) {
      …
    }
  }
}
```

After loop fission

```
# pragma acc parallel loop gang
for(i3 = 1; i3 < n3 -1; i3 ++) {
  # pragma acc loop worker
  for (i2 = 1; i2 < n2 -1; i2 ++) {
    # pragma acc loop vector
    for (i1 = 0; i1 < n1; i1 ++) {
      ...
    }
  }
}
```

```
# pragma acc parallel loop gang
for(i3 = 1; i3 < n3 -1; i3 ++) {
  # pragma acc loop worker
  for (i2 = 1; i2 < n2 -1; i2 ++) {
    # pragma acc loop vector
    for (i1 = 1; i1 < n1-1; i1 ++) {
      ...
    }
  }
}
```

OpenUH
Where no research compiler has gone before

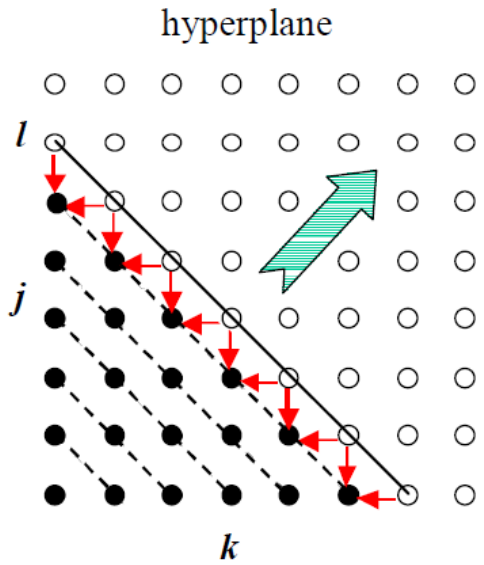
Data Movement Optimization

- In NPB, most of the benchmarks contain many global arrays live throughout the entire program
- Allocate memory at the beginning
- Update directive to synchronize data between host and device
- Async directive to overlap communication and computation

OpenUH
Where no research compiler has gone before

New algorithm adoption

- To exploit parallelism
  - We adopt Hyperplane[1] algorithm for LU



- To overcome GPU memory size limit (blocking EP)

OpenUH
Where no research compiler has gone before

1. Leslie Lamport. The Parallel Execution of DO Loops, in Communications of the ACM, Volumn 17, Issue 2, pp. 83-93

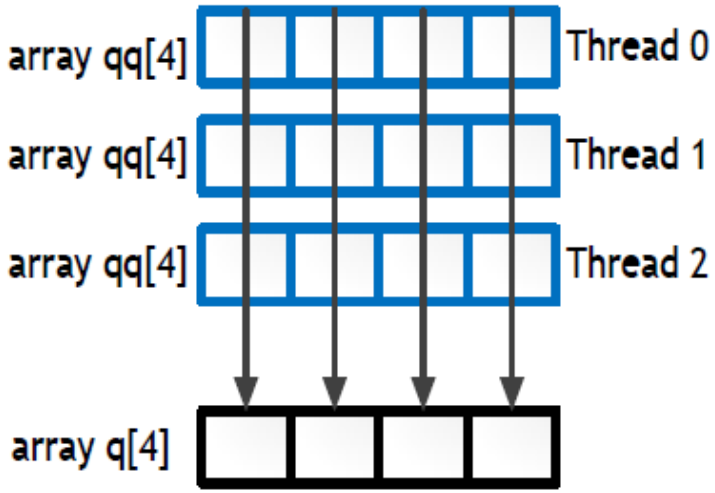
NAS Parallel Benchmarks (NPB)

- Well recognized for evaluating current and emerging multi-core/many-core hardware architectures
- 5 parallel kernels
  - IS, EP, CG, MG and FT
- 3 simulated computational fluid dynamics (CFD) applications
  - LU, SP and BT
- Different problem sizes
  - Class S: small for quick test purpose
  - Class W: workstation size
  - Class A,B,C: standard test problems
  - Class D,E,F: large test problems
- Use OpenACC 1.0 to create the benchmark suite

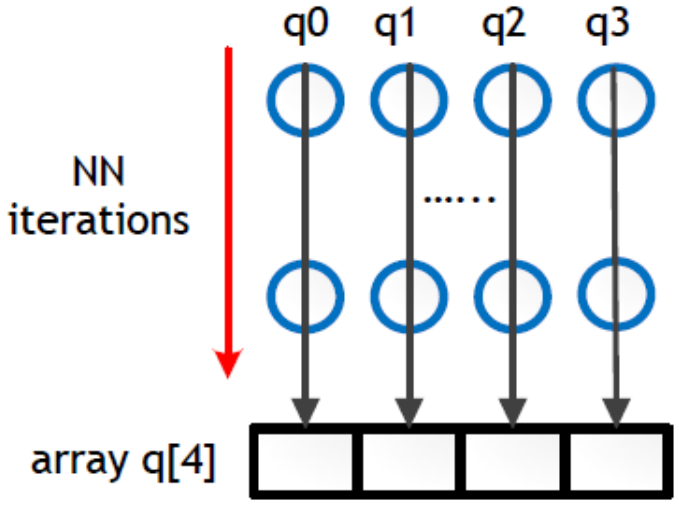
OpenUH
Where no research compiler has gone before

Parallelizing NPB Benchmarks - EP

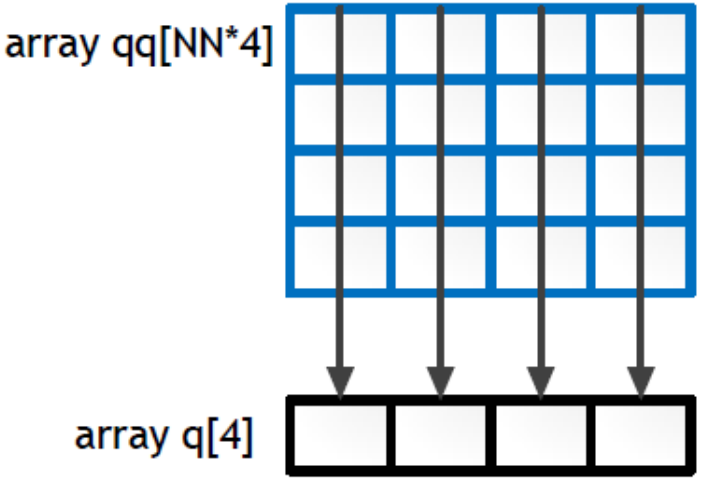
- Array reduction issue – every element of an array needs reduction



(a) OpenMP solution



(b) OpenACC solution 1



(c) OpenACC solution 2

OpenUH
Where no research compiler has gone before

Parallelizing NPB Benchmarks – FT and IS

- FFT: complex data type and too many function calls
- Solution for FFT:
  - Convert the array of structures (AoS) to structure of arrays (SoA)
  - Manually inline function calls in all compute regions

- IS: irregular memory access
  - Atomic operations
  - Inclusive scan and exclusive scan

OpenUH
Where no research compiler has gone before

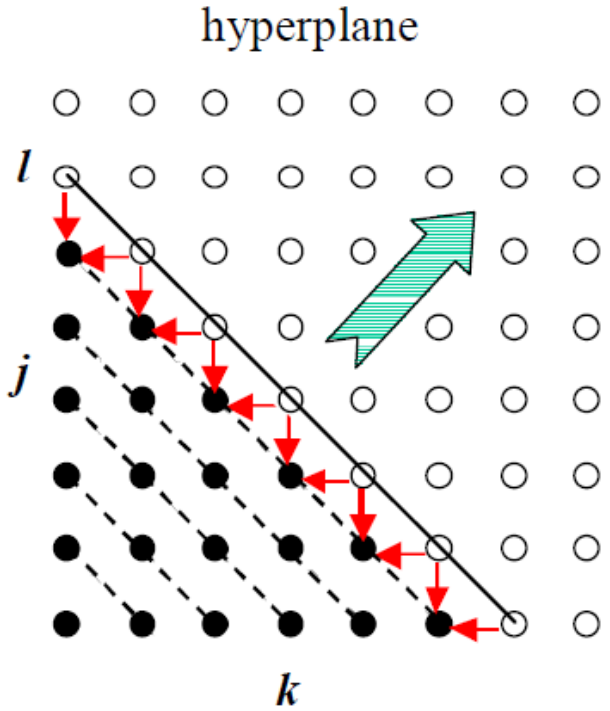
Parallelizing NPB Benchmarks - MG

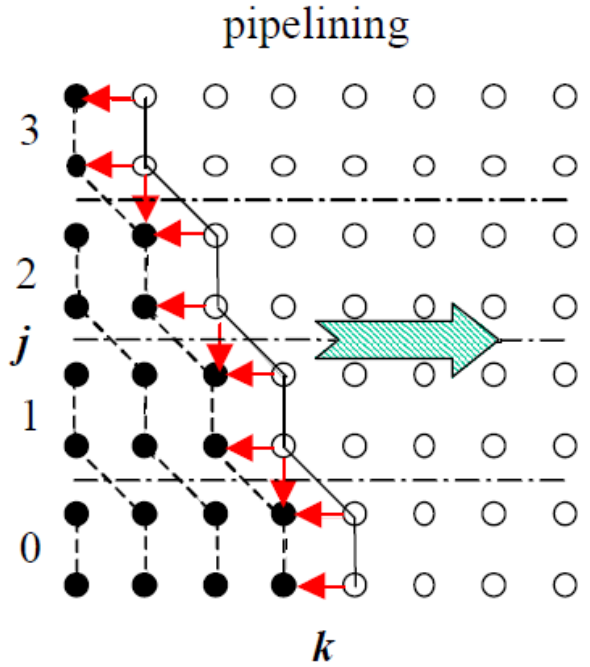
- Partial array copy extensively used
- In resid() routine, "*ov*" is read only and "*or*" is written only
  - They may point to the same memory (alias)
  - Using copyin for "*ov*" and copyout for "*or*" is implementation defined
  - Solution: put two arrays into one copy clause

OpenUH
Where no research compiler has gone before

Parallelizing NPB Benchmarks - LU

- OpenMP uses pipepline, OpenACC uses hyperplane
- Array privatization and memory coalescing optimization





OpenUH
Where no research compiler has gone before

Haoqiang Jin, Michael Frumkin and Jerry Yan, The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report NAS-99-011. 1999

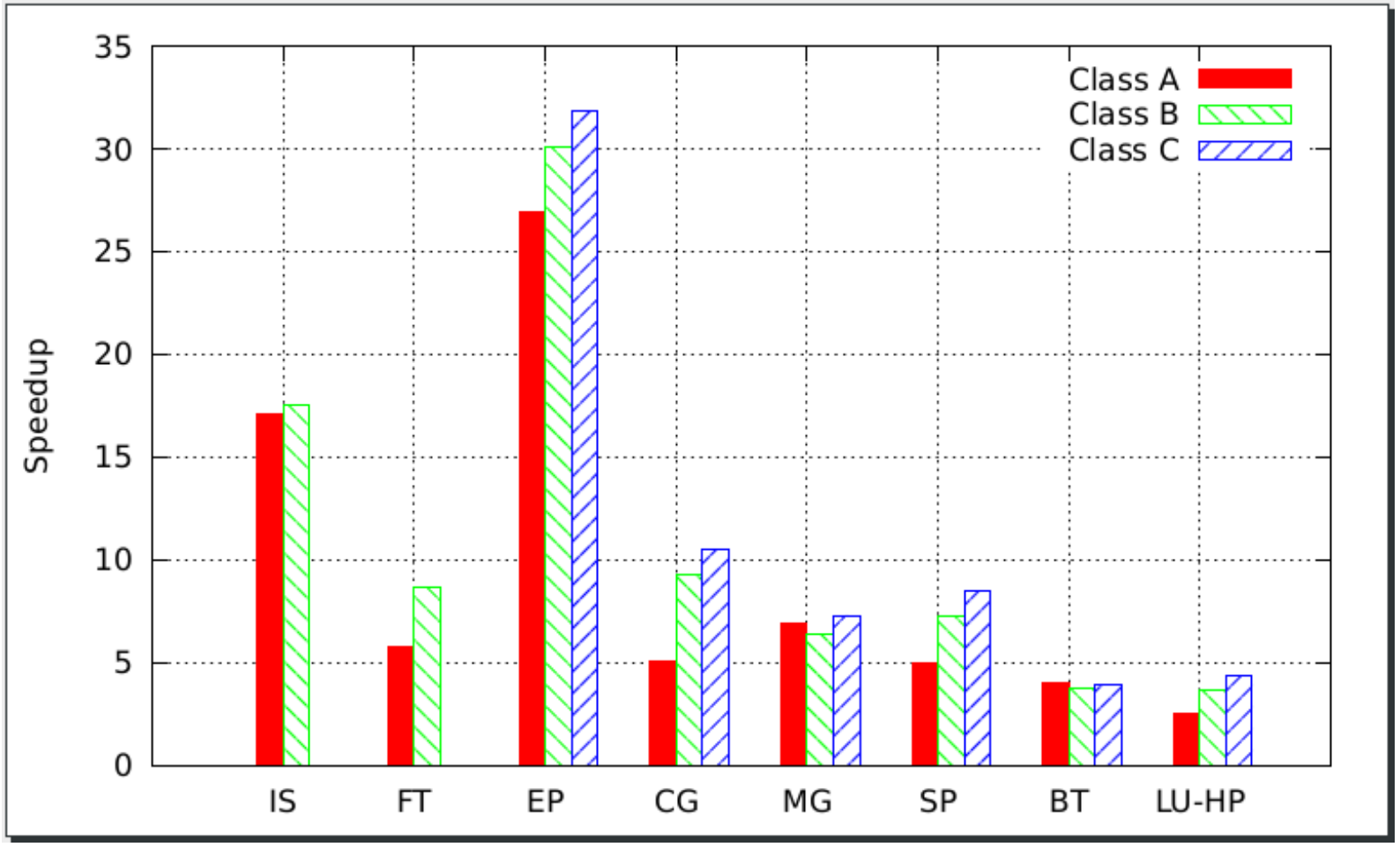
Performance Evaluation

- 16 cores Intel Xeon x86_64 CPU with 32 GB memory
- Kepler 20 with 5GB memory
- NPB 3.3 C version[1]
- GCC4.4.7, OpenUH
- Compare to serial and CUDA version

OpenUH
Where no research compiler has gone before

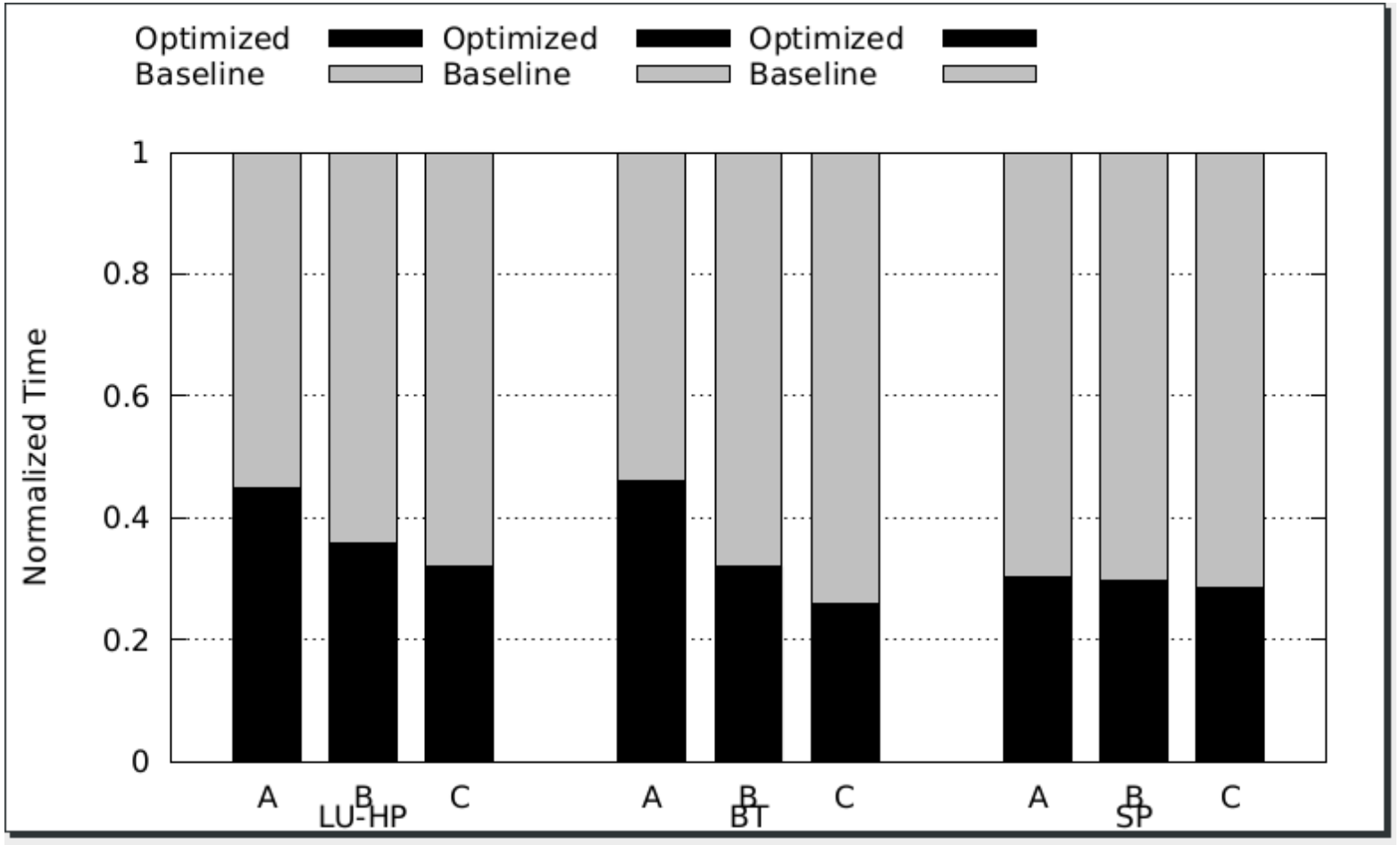
1. http://aces.snu.ac.kr/Center_for_Manycore_Programming/SNU_NPB_Suite.html

Performance Evaluation of OpenUH OpenACC NPB – compared to serial



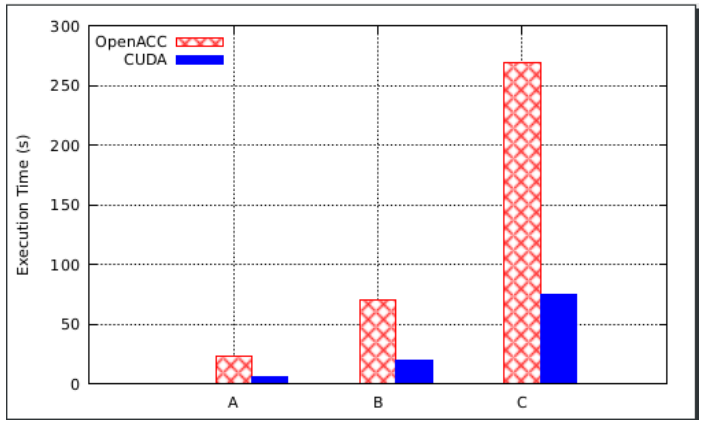
OpenUH
Where no research compiler has gone before

Performance Evaluation of OpenUH OpenACC NPB – effectiveness of optimization

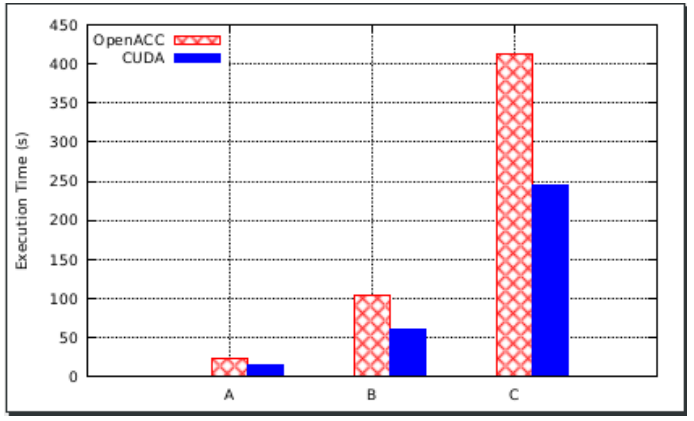


OpenUH
Where no research compiler has gone before

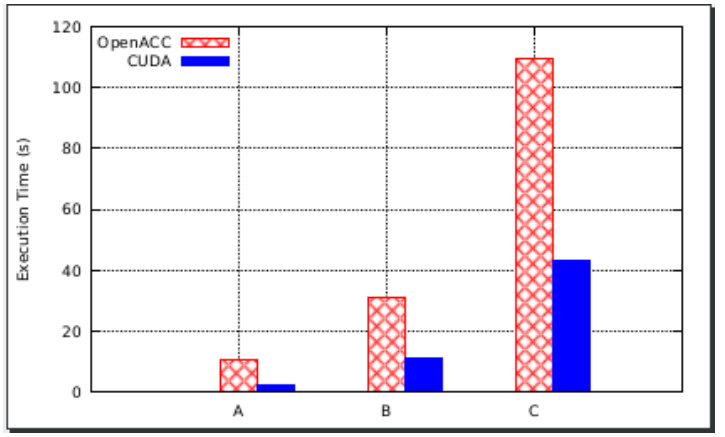
Performance Evaluation OpenUH OpenACC NPB – OpenACC vs CUDA[1]



LU-HP



BT



SP

OpenUH
Where no research compiler has gone before

1. http://www:tu-chemnitz:de/informatik/PI/forschung/download/npb-gpu

Conclusion and Future Work

- Conclusion
  - Discussed different parallelization techniques for OpenACC
  - Demonstrated speedup of OpenUH OpenACC over serial code
  - Compared the performance between OpenUH OpenACC and CUDA
  - Contributed 4 NPB benchmarks to SPEC ACCEL V1.0 (released on March 18, 2014)
    - http://www.hpcwire.com/off-the-wire/spechpg-releases-new-hpc-benchmark-suite/
    - Looking forward to making more contributions to future SPEC ACCEL suite
  - Performance comparison between different compilers: in session S4343 (Wednesday, 03/26, 17:00 – 17:25, Room LL20C)
- Future Work
  - Explore other optimizations
  - Automate some of the optimizations in OpenUH compiler

OpenUH
Where no research compiler has gone before