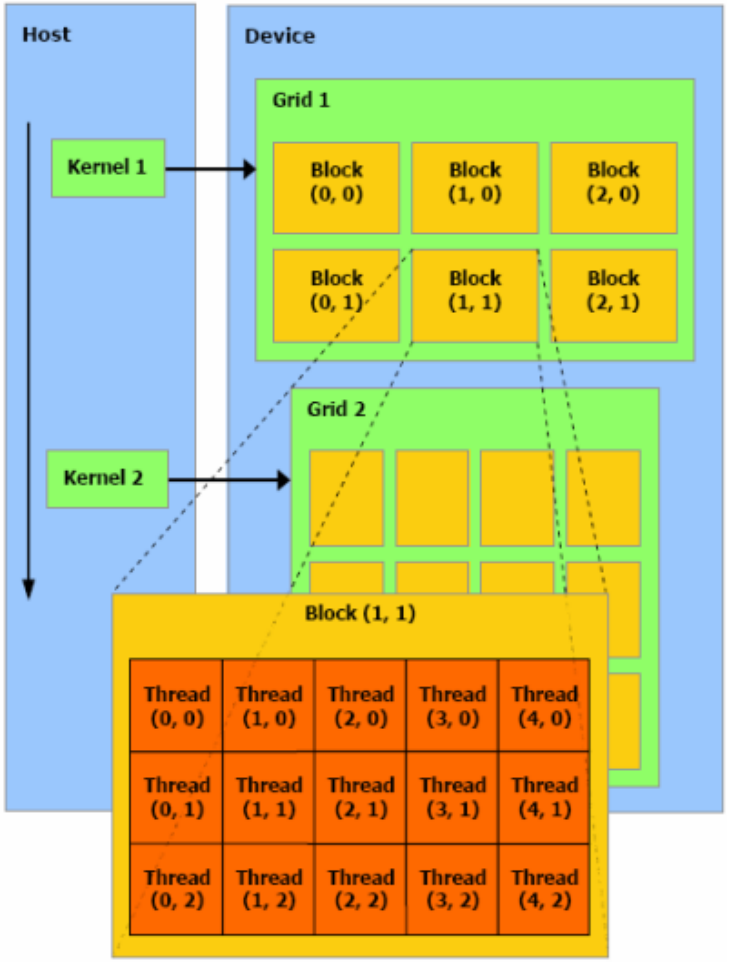
# Auto-tuning a High-Level Language Targeted to GPU Codes

By Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, John Cavazos

# GPU Computing

- Utilization of GPU gives speedup on many algorithms
  - Parallel programming on GPU using CUDA / OpenCL environments

# Directive-Based GPU Programming

- Compiler generates GPU kernels from sequential code w/ pragmas
- Advantages of using directives:
  - Preserves serial implementation of code
  - Focus on highlighting parallelism
  - Eases interaction between scientists and programmers
- Frameworks include HMPP and OpenACC


# GPU Code Optimization

- Code transformations may improve performance
  - Loop unrolling, tiling, permutation, fusion/fission, which loop(s) parallelized
- Constant tweaking required to get best performance
  - Resulting code may be brittle
  - Optimized code on one architecture may give poor performance on alternate architecture

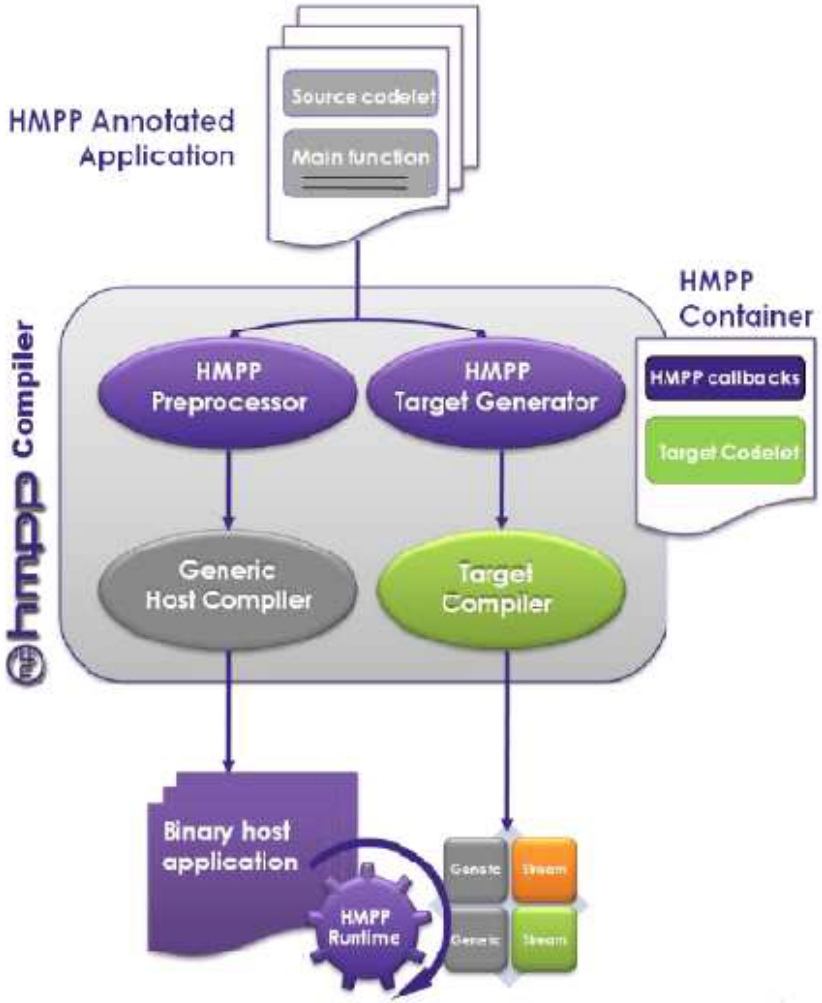
# Optimization Using HMPP Workbench

- Auto-tuning w/ HMPP Workbench to determine good transformations

- HMPP Workbench
  - Source-to-source compiler developed by CAPS Enterprise
  - Directive-based framework targeted to GPUs
  - Transforms sequential code to GPU code
  - Contains pragmas for code optimization


# HMPP Compiler

- Generates GPU code from pragmas
- Used to explore large optimization space



# Experimental Set-Up

- Goal: optimize code using particular transformations via pragmas

| Pragma | Experimental Parameter Values in Codes |
|---|---|
| permute | Depends on kernel. Different Ordering of loops. |
| unroll | Unroll factors 1 through 8 using 'contiguous' and 'split' options. |
| tile | Tiling factors 1 through 8. |
| parallel / noParallel | Depends on kernel. Determines which loops are parallelized for GPU processing. |
| remainder / guarded | Used each option with loop unrolling. 'Remainder' option allows generation of remainder loop. The 'guarded' option avoids this via guards in unrolled loop. |

# Experimental Set-Up

- Unroll/tiling transformations using pragmas

**(a) contiguous unroll**

```
#pragma hmppcg unroll 2, contiguous
for (i = 0; i < N; i++)
{
    B[i] = A[i];
}
```

```
for (i = 0; i < N/2; i++)
{
    B[2*i] = A[2*i];
    B[2*i + 1] = A[2*i + 1];
}
```

**(b) split unroll**

```
#pragma hmppcg unroll 2, split
for (i = 0; i < N; i++)
{
    B[i] = A[i];
}
```

```
for (i = 0; i < N/2; i++)
{
    B[i] = A[i];
    B[i + N/2] = A[i + N/2];
}
```

**(c) tiling**

```
#pragma hmppcg tile i:2
for (i = 0; i < N; i++)
{
    B[i] = A[i];
}
```

```
for (i = 0; i < N/2; i++)
{
    for (i_2 = 0; i_2 < 2; i_2++)
    {
        B[2*i + i_2] = A[2*i + i_2];
    }
}
```

# Experimental Set-Up

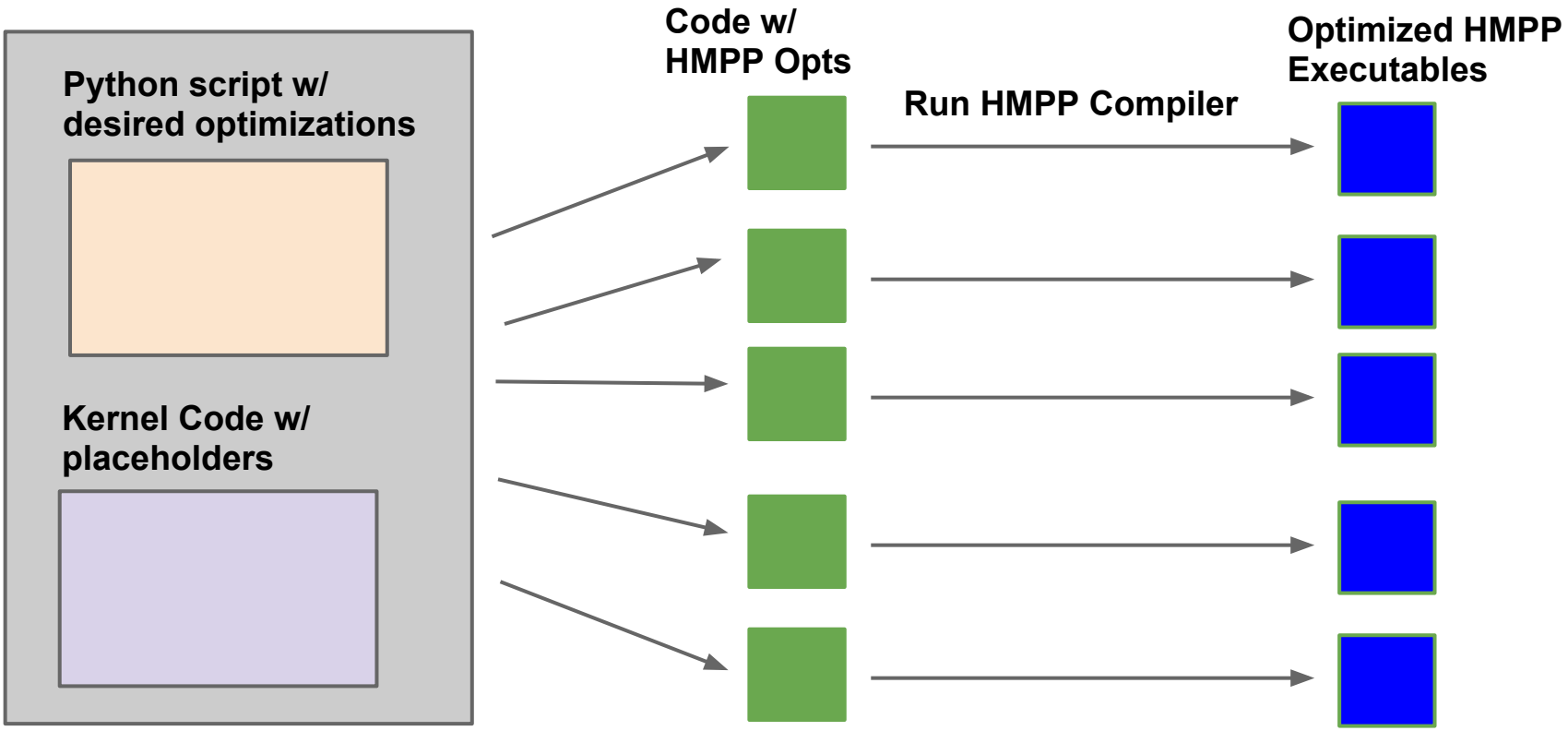
- HMPP-annotated codes generated w/ python script
  - Uses kernel code w/ placeholders for pragmas

```
%(permutePragma)
%(unrollTilePragma_iLoop)
%(parallelNoParallelPragma_iLoop)
for (i = 0; i < NI; i++)
{
      %(unrollTilePragma_jLoop)
      %(parallelNoParallelPragma_jLoop)
      for (j = 0; j < NJ; j++)
          {
              c[i][j] *= p_beta;
              %(unrollTilePragma_kLoop)
              %(parallelNoParallelPragma_kLoop)
              for (k = 0; k < NK; k++)
                {
                  temp = p_alpha * a[i][k] * b[k][j];
                  c[i][j] += temp;
                }
          }
}
```

**GEMM code kernel w/ placeholders for pragmas**

# Experimental Set-Up

- Execution flow



# Experimental Set-Up

- Initial experiments on C2050 GPU
  - Fermi architecture
  - 448 cores


- CUDA 4.0
  - CUDA codes compiled w/ Open64-based compiler
  - OpenCL codes compiled w/ LLVM-based compiler

# Experimental Results

- 2D Convolution
  - Dimensions: 4096 X 4096

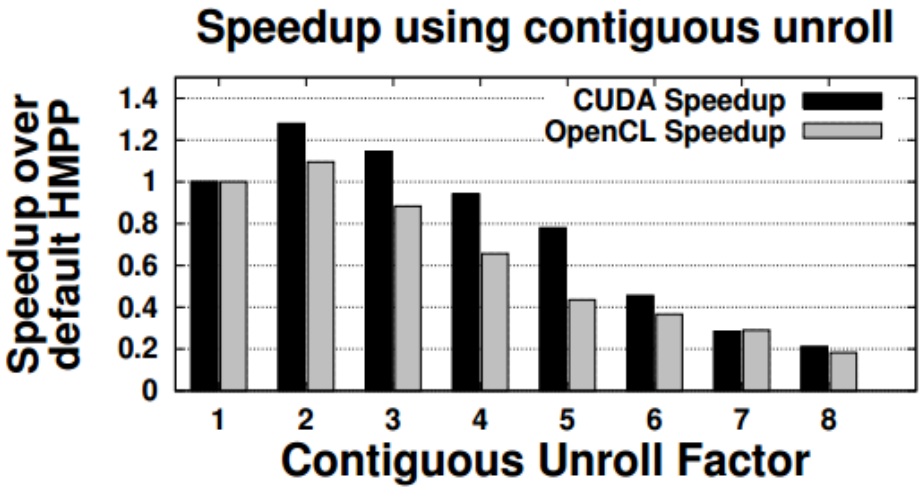
```
for (int i = 0; i < DIM_0 - 1; i++)
  for (int j = 0; j < DIM_1 - 1; j++)
    B[i][j] =
          C_11 * A[i-1][j-1]+C_12 * A[i+0][j-1] +
          C_13 * A[i+1][j-1]+C_21 * A[i-1][j+0] +
          C_22 * A[i+0][j+0]+C_23 * A[i+1][j+0] +
          C_31 * A[i-1][j+1]+C_32 * A[i+0][j+1] +
          C_33 * A[i+1][j+1];
```

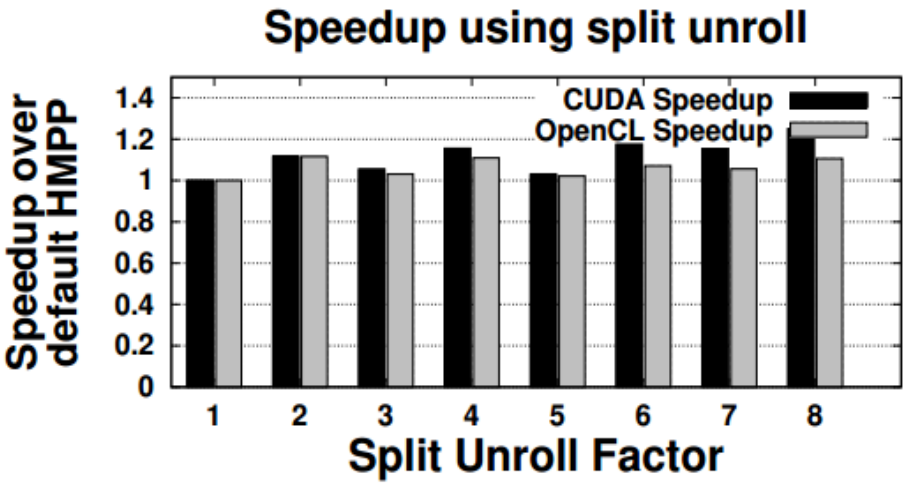
# Experimental Results

- 2D Convolution
  - Experiments using HMPP-generated CUDA and OpenCL code
  - Improved performance using initial loop order w/ unrolling/tiling on inner loop
    - Alternate loop order increases runtime
    - Unrolling/tiling on outer loop increases runtime

# Experimental Results

- 2D Convolution
  - Results using contiguous and split unroll in inner loop:



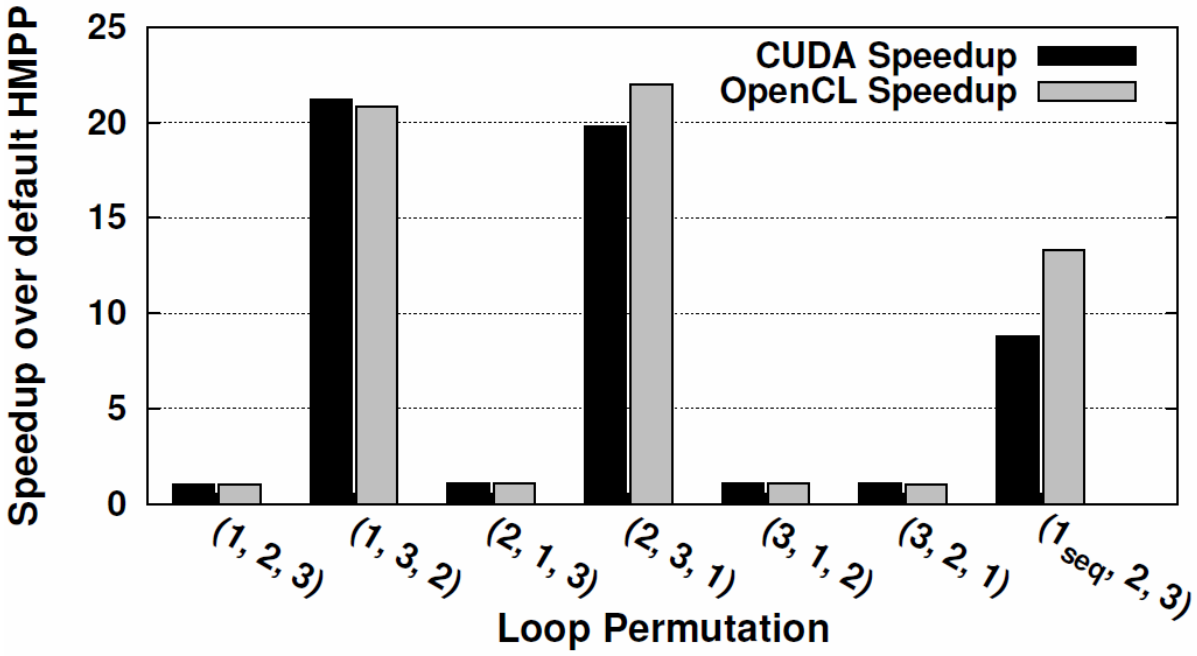
# Experimental Results

- 3D Convolution
  - Dimensions: 256 X 256 X 256

```
for (i = 1; i < NI - 1; ++i) // 0
{
    for (j = 1; j < NJ - 1; ++j) // 1
    {
        for (k = 1; k < NK - 1; ++k) // 2
        {
            B[i][j][k] = c11 * A[i - 1][j - 1][k - 1]
                + c13 * A[i + 1][j - 1][k - 1] + c21 * A[i - 1][j - 1][k - 1]
                + c23 * A[i + 1][j - 1][k - 1] + c31 * A[i - 1][j - 1][k - 1]
                + c33 * A[i + 1][j - 1][k - 1] + c12 * A[i + 0][j - 1][k + 0]
                + c22 * A[i + 0][j + 0][k + 0] + c32 * A[i + 0][j + 1][k + 0]
                + c11 * A[i - 1][j - 1][k + 1] + c13 * A[i + 1][j - 1][k + 1]
                + c21 * A[i - 1][j + 0][k + 1] + c23 * A[i + 1][j + 0][k + 1]
                + c31 * A[i - 1][j + 1][k + 1] + c33 * A[i + 1][j + 1][k + 1];
        }
    }
}
```

# Experimental Results

- 3D Convolution
  - Results using different permutations
    - No unrolling/tiling



# Experimental Results

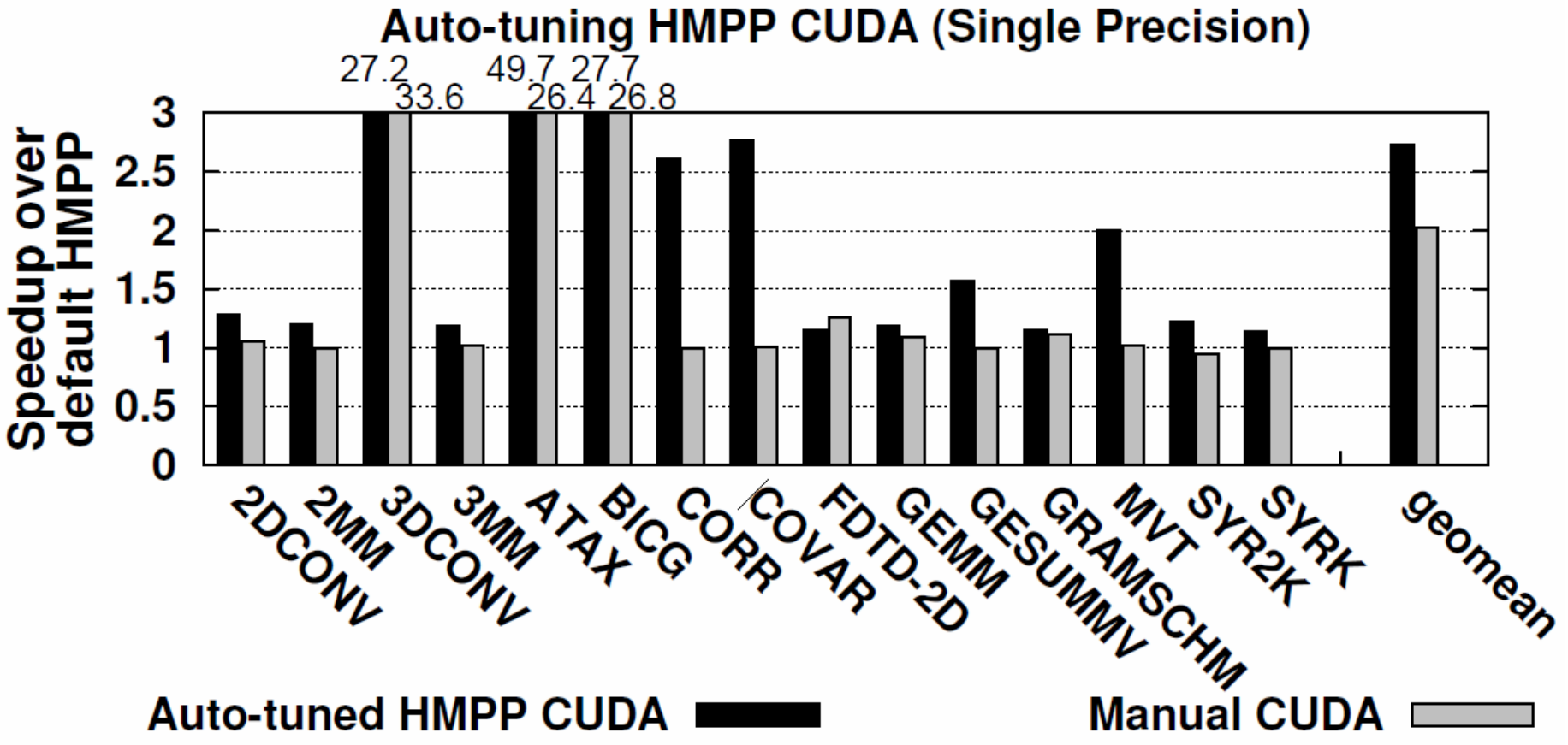
- 3D Convolution
  - Experiments with unrolling/tiling in best permutations
  - CUDA results using (1, 3, 2) permutation:
    - With no unrolling/tiling: 21.2x speedup
    - With unrolling loop ‘3’ by a factor of 4 using ‘contiguous’ and ‘guarded’ pragmas: 27.2x speedup
  - OpenCL results
    - Best found config. used (2, 3, 1) permutation without unrolling/ tiling
    - 22x speedup

# Experimental Results

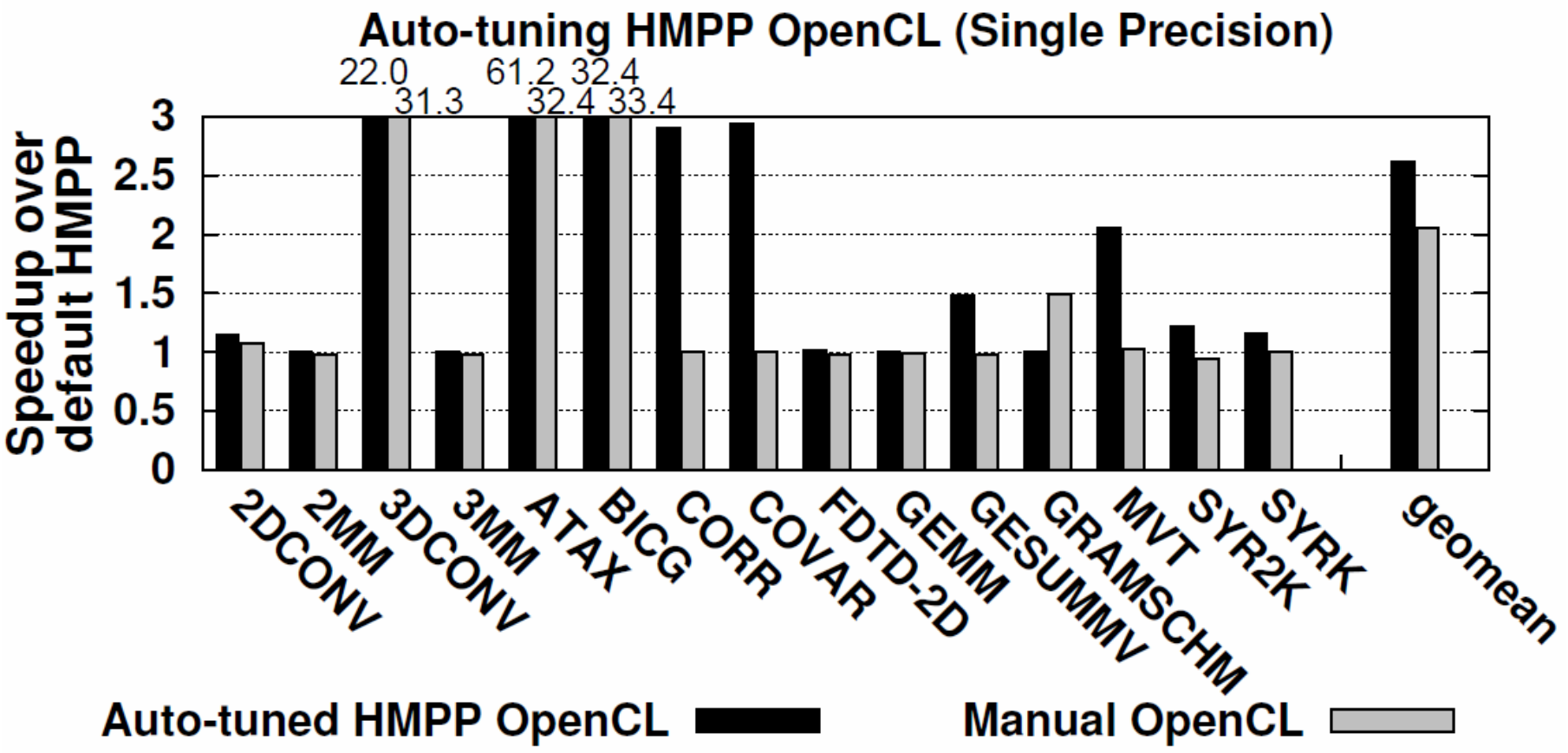
- **Polybench Benchmark Suite**
  - Codes for linear algebra, data-mining, and stencils
  - Converted codes to CUDA / OpenCL using HMPP
    - Optimized codes using HMPP pragmas
    - Search space of many possible transformations
  - Constructed hand-written CUDA/OpenCL kernels

**Available at http://www.cse.ohio-state.edu/~pouchet/software/polybench/**

# Polybench Suite w/ CUDA



# Polybench Suite w/ OpenCL



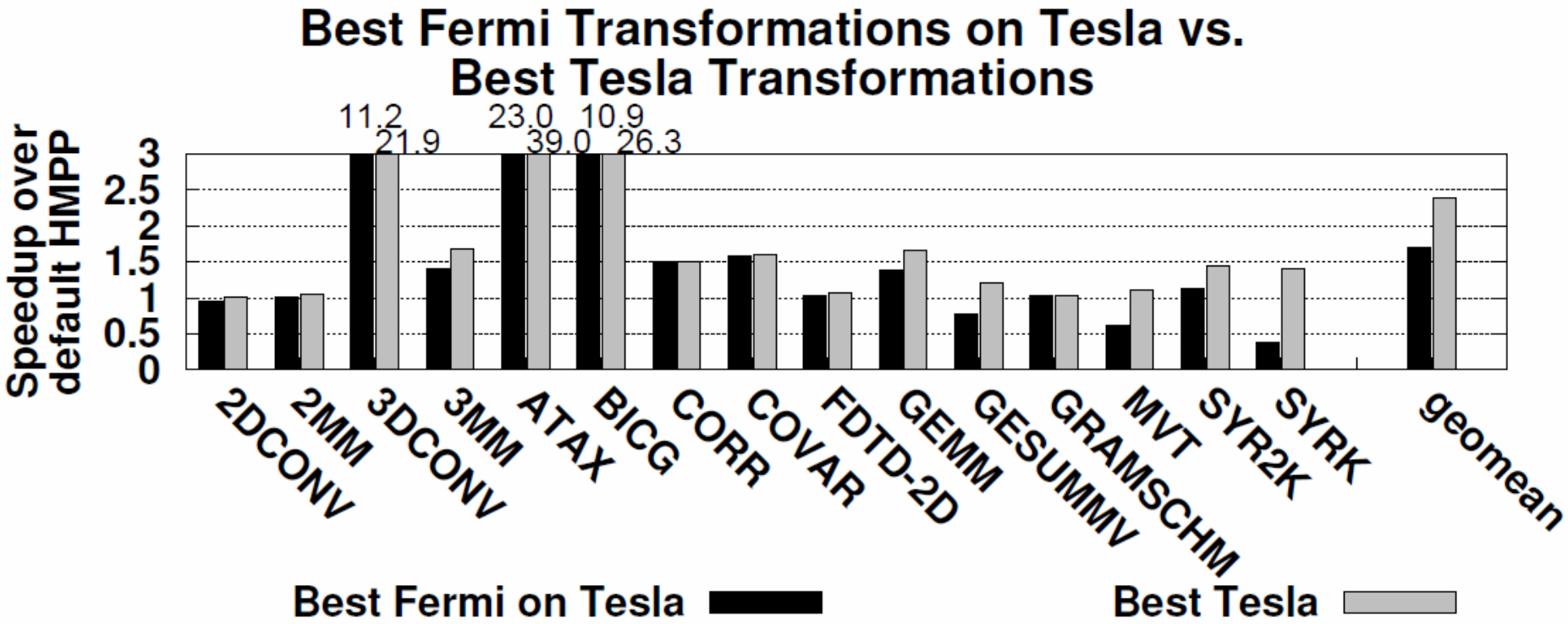
# Best found transformations on selected codes

| Code | Best Found Transformations (CUDA) | Best Found Transformations (OpenCL) |
|---|---|---|
| **ATAX** | Reverse order of 2nd nested loop set and tile 1st and 2nd loop w/ factor 4 | Reverse order of 2nd nested loop set and tile 1st and 2nd loops w/ factor 2 |
| **CORR** | Parallelize 8th loop rather than 7th loop and tile 9th loop w/ factor 4 | Parallelize 8th loop rather than 7th loop and unroll 9th loop using ‘contiguous’ and ‘remainder’ options w/ factor 2 |
| **GEMM** | Unroll 3rd loop using ‘split’ and ‘guarded’ options with factor 3 | Unroll 3rd loop using ‘contiguous’ and ‘guarded’ options with factor 8 |

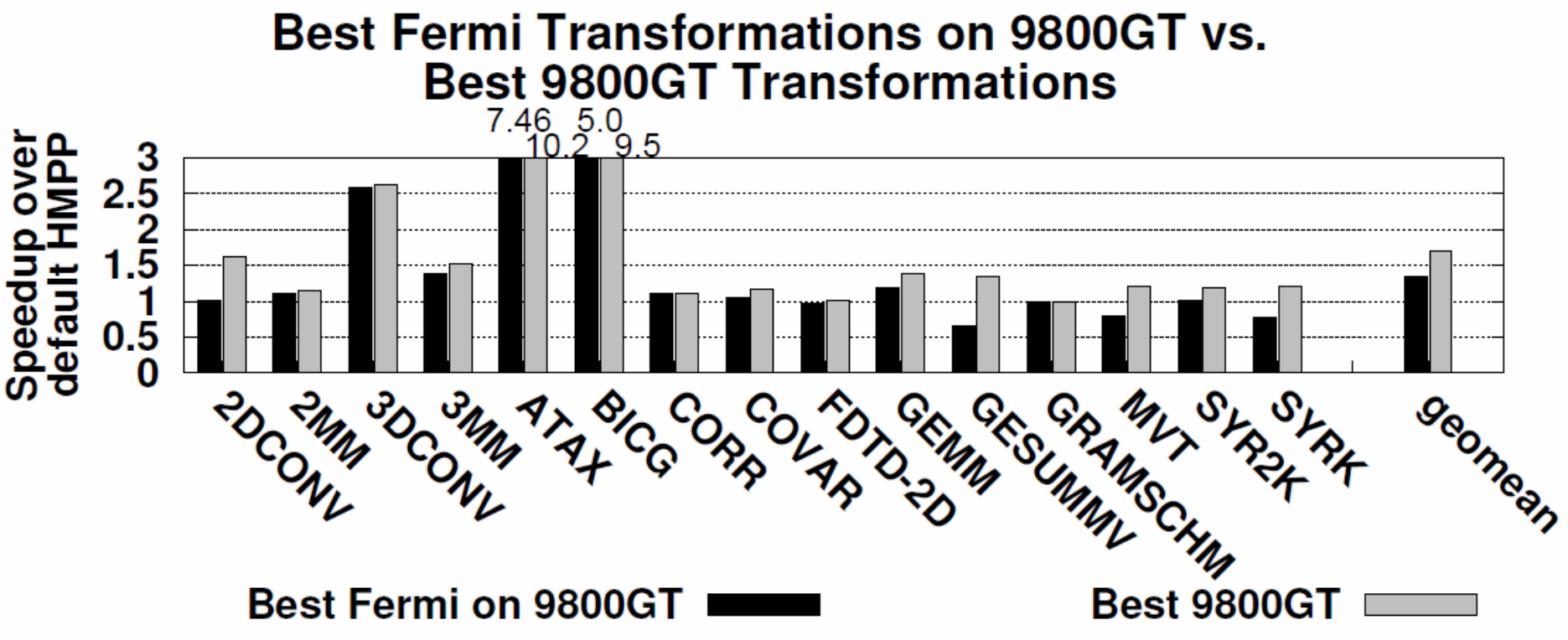
# HMPP Auto-tuning Results Discussion

- Important to find best permutation for memory coalescence
- Particular loops parallelized can be significant
  - Default HMPP configuration may not be optimal
- Applying unrolling to innermost loop often contributes to best speedup
  - Unrolling outermost loop often hurts performance

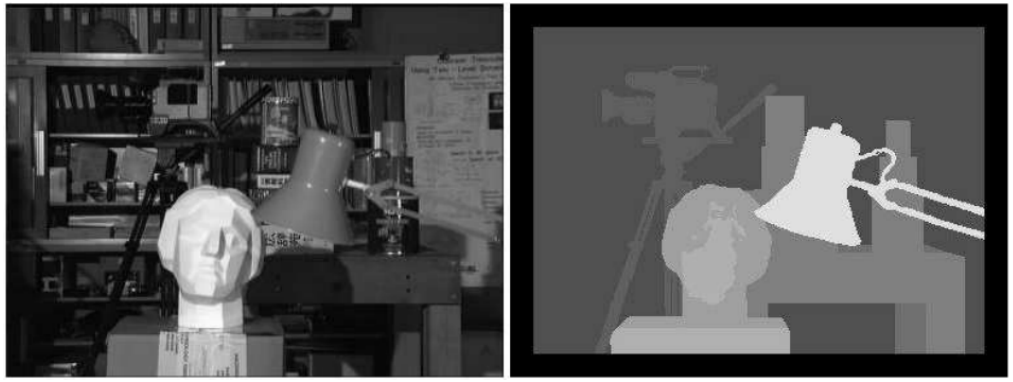
# Results on GTX 280 (Tesla)



# Results on 9800 GT

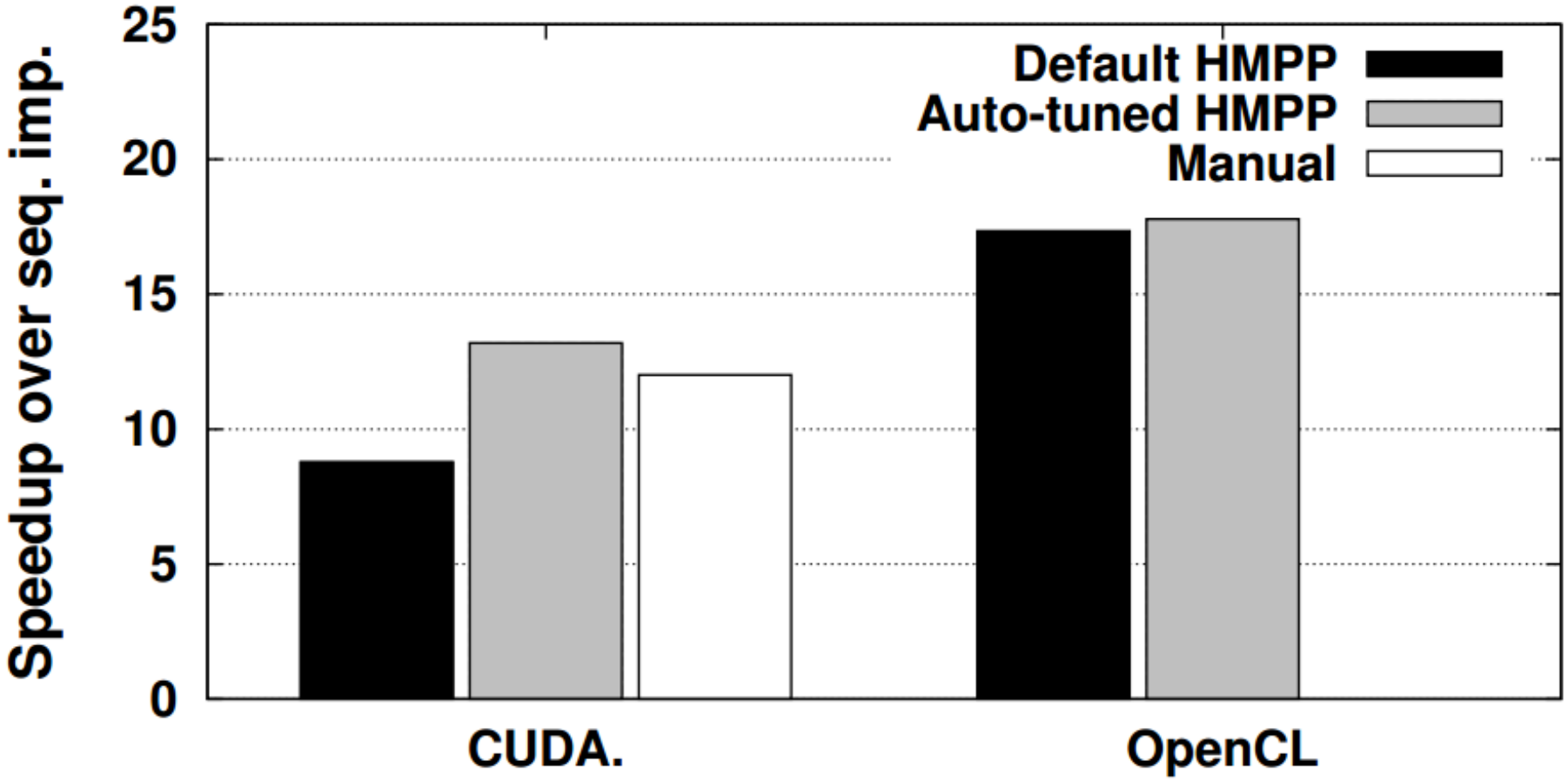


# Belief Propagation for Stereo Vision



- Computes disparity map from stereo set of images
- Parallelize code available online using HMPP
  - Optimize using HMPP pragmas
  - Compare to manual CUDA implementation

# Results for Belief Propagation



# Future Work

- Use additional code transformations
- Run experiments on additional GPU and other many-core architectures
- Develop model to optimize any input kernel

# Conclusions

- Developed optimized GPU kernels using auto-tuning w/ HMPP
  - Codes available online at http://www.cse.ohio-state.edu/~pouchet/software/polybench/GPU
- Improved runtime over default
  - Method works across architectures