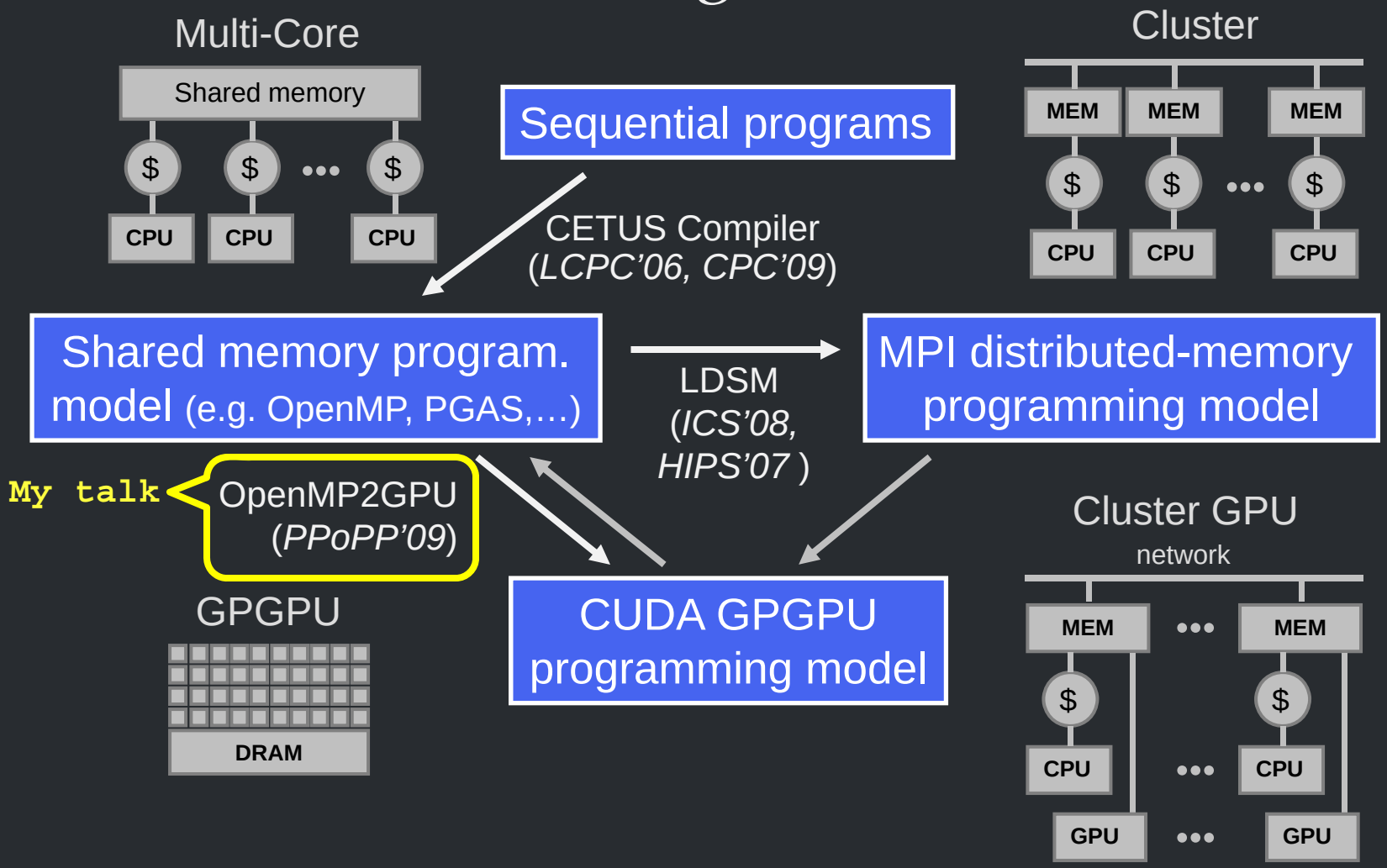
Compiling Shared Memory Programs for GPUs

Seung-Jai Min

Future Technology Group

Lawrence Berkeley National Laboratory

Ph.D Research: Source-to-source Translation of Parallel Programs



Contents

- Motivation
- OpenMP for GPGPU Programming Model
- Two-phase Translation Framework
- Optimization Techniques
- Performance Evaluation
- Conclusions

Motivation

- GPGPUs as promising parallel platform for HPC
- CUDA has been introduced to make GPU programming easy
- However, GPU programming is still complex and error-prone

OpenMP vs. CUDA

Jacobi OpenMP version

```
#include "omp.h"

float a[SIZE_2][SIZE_2];
float b[SIZE_2][SIZE_2];
int main (int argc, char *argv[]) {
   int i, j, k;
   for (k = 0; k < ITER; k++) {
      #pragma omp parallel for private(i, j)
      for (i = 1; i <= SIZE; i++) {
         for (j = 1; j <= SIZE; j++) {
            a[i][j] = (b[i - 1][j] + b[i + 1][j] + b[i][j - 1] + b[i][j + 1]) / 4.0f;
         }
      }
      #pragma omp parallel for private(i, j)
      for (i = 1; i <= SIZE; i++) {
         for (j = 1; j <= SIZE; j++) {
            b[i][j] = a[i][j];
         }
      }
   }
   return 0;
}
```

Jacobi CUDA version

```
#include <cutil.h>
#define BLOCK_SIZE  256
int gpuNumThreads = BLOCK_SIZE;  int gpuNumBlocks;  float a[(2048+2)][(2048+2)];  float b[(2048+2)][(2048+2)];
__global__ void main_kernel0(size_t pitch_a, float * a, size_t pitch_b, float * b) {
   int j; int i;  int gtid = (threadIdx.x+(blockIdx.x*blockDim.x));   i=(gtid+1);
   if (i<=2048) {
      for (j=1; j<=2048; j ++ ) {
         ( * (((float * )(((char * )a)+(i*pitch_a)))+j))=((((( * (((float * )(((char * )b)+((i-1)*pitch_b)))+j))+( * (((float * )(((char
            * )b)+((i+1)*pitch_b)))+j)))+( * (((float * )(((char * )b)+(i*pitch_b)))+(j-1))))+( * (((float * )(((char
            * )b)+(i*pitch_b)))+(j+1))))/4.0);
      }
   }
}
__global__ void main_kernel1(size_t pitch_a, float * a, size_t pitch_b, float * b) {
   int j; int i;  int gtid = (threadIdx.x+(blockIdx.x*blockDim.x));  i=(gtid+1);
   if (i<=2048) {
      for (j=1; j<=2048; j ++ ) {
         ( * (((float * )(((char * )b)+(i*pitch_b)))+j))=( * (((float * )(((char * )a)+(i*pitch_a)))+j));
      }
   }
}
int main(int argc, char * argv[]) {
   int i; int j; int k;  float * gpu_a; size_t pitch_a; float * gpu_b; size_t pitch_b;
   CUT_DEVICE_INIT(argc, argv);
   CUDA_SAFE_CALL(cudaMallocPitch(((void *  * )( & gpu_a)), ( & pitch_a), ((2048+2)*sizeof (float)), (2048+2)));
   CUDA_SAFE_CALL(cudaMemcpy2D(gpu_a, pitch_a, a, ((2048+2)*sizeof (float)), ((2048+2)*sizeof (float)), (2048+2),
         cudaMemcpyHostToDevice));
   CUDA_SAFE_CALL(cudaMallocPitch(((void *  * )( & gpu_b)), ( & pitch_b), ((2048+2)*sizeof (float)), (2048+2)));
   CUDA_SAFE_CALL(cudaMemcpy2D(gpu_b, pitch_b, b, ((2048+2)*sizeof (float)), ((2048+2)*sizeof (float)), (2048+2),
         cudaMemcpyHostToDevice));
   for (k=0; k<100; k ++ ) {
      #pragma cetus parallel for shared(a, b) private(i, j) threadprivate()
      dim3 dimBlock0(gpuNumThreads, 1, 1);  gpuNumBlocks=8;  dim3 dimGrid0(gpuNumBlocks, 1, 1);
      main_kernel0<<<dimBlock0, dimGrid0>>>(pitch_a, gpu_a, pitch_b, gpu_b);
      #pragma cetus parallel for shared(a, b) private(i, j) threadprivate()
      dim3 dimBlock1(gpuNumThreads, 1, 1);  gpuNumBlocks=8;  dim3 dimGrid1(gpuNumBlocks, 1, 1);
      main_kernel1<<<dimBlock1, dimGrid1>>>(pitch_a, gpu_a, pitch_b, gpu_b);
   }
   CUDA_SAFE_CALL(cudaMemcpy2D(b, ((2048+2)*sizeof (float)), gpu_b, pitch_b, ((2048+2)*sizeof (float)), (2048+2),
         cudaMemcpyDeviceToHost));
   CUDA_SAFE_CALL(cudaMemcpy2D(a, ((2048+2)*sizeof (float)), gpu_a, pitch_a, ((2048+2)*sizeof (float)), (2048+2),
         cudaMemcpyDeviceToHost));
   CUT_EXIT(argc, argv);
}
```

Contributions

- Present the first compiler framework for automatic translation of OpenMP applications into CUDA-based GPGPU applications.
- Propose compiler optimization techniques to improve the memory accesses in stream architectures such as GPGPUs.
- Demonstrate that the proposed framework increases the performance up to 50X (12X on average) over baseline translations. (up to 328X over serial CPU version)

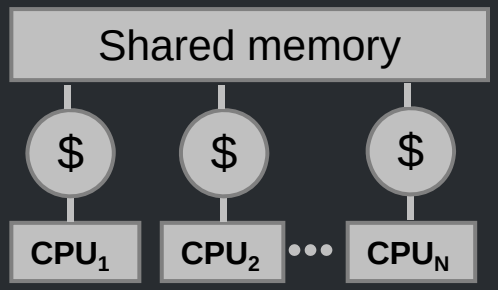
Is OpenMP the right programming model for GPGPUs?

YES!

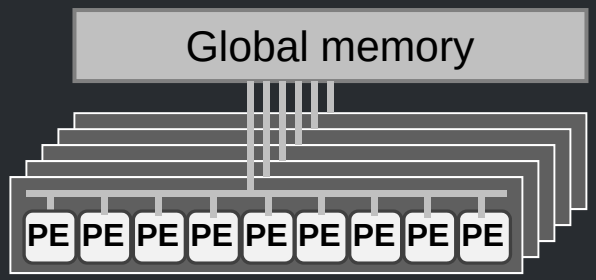
- Loop-level parallelism
  - an ideal target for utilizing GPU's massively parallel computing units
- Fork-join model
  - (master thread + worker threads) vs. (CPU thread + GPU threads)
- Incremental parallelization of applications

Challenge: SMP vs. Stream Architecture

- SMP (Symmetric Multi-Processor)
  - OpenMP programs are tuned to traditional SMP
- Stream Architectures: operate on a large data space (or stream) in parallel, typically in an SIMD manner
  - tuned for accesses to regular, consecutive elements of the data stream



OpenMP thread
(SIMD/MIMD, regular/irregular)



GPU thread
(SIMD, regular control/data)

Difference in Optimization Techniques

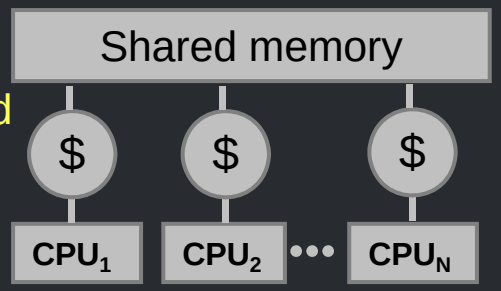
OpenMP for SMP

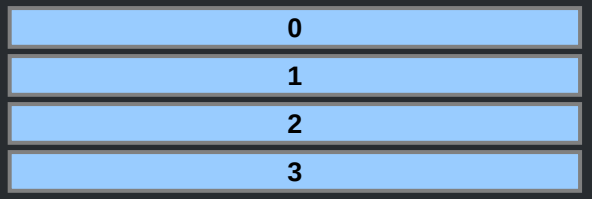
```
#pragma omp for
for (i=0; i<100; i++) {
   for (j=0; j<100; j++)
      a[i][j] = b[i][j];
}
```

OpenMP for Stream Arch.

```
for (i=0; i<100; i++) {
#pragma omp for schedule(1)
   for (j=0; j<100; j++)
      a[i][j] = b[i][j];
}
```

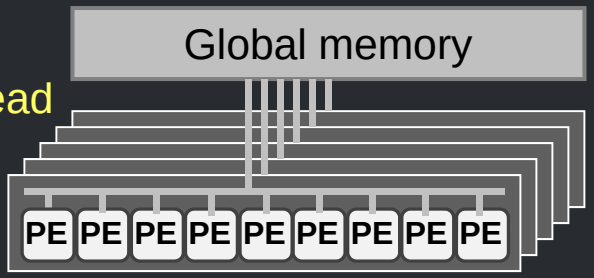
Intra-thread Locality





Block distribution
(- falase-sharing, +data locality)

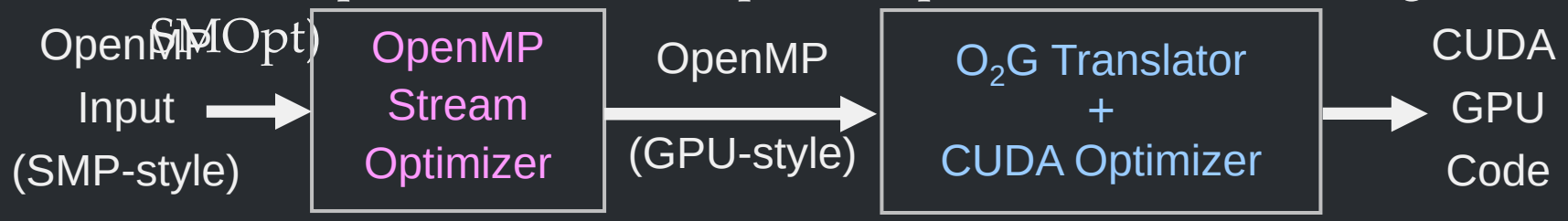
Inter-thread locality



k=1, cyclic distribution
(coalesced memory accesses)

Solution: Two-Step OpenMP to GPU Compilation Framework

- Step 1: OpenMP Stream Optimizer
  - Convert SMP-style OpenMP to GPU-style OpenMP
  - Compile-time transformations to optimize global memory accesses
- Step 2: OMP-to-GPU Translation
  - O_2G baseline translator: interpretation of OpenMP semantics
  - CUDA optimizer : CUDA specific optimizations (Unrolling, SMOpt)



Step1: OpenMP Stream Optimizer

- Problem:
  - Which OpenMP region should be executed on GPU device?
- Continuous Memory Access (Stride-One access)
  - An array access is *continuous* in a loop nest if its subscript expression increases monotonically with stride one
- OpenMP Stream Optimizations
  - Transform the input code to expose as many continuous memory accesses as possible
  - The OMP2GPU translator applies coalesced memory accesses for continuous memory accesses

Step 2: OMP2GPU Baseline Translation

- GPU kernel extraction
  - Candidate: OpenMP parallel loops and parallel sections
- Work Partitioning
  - Each iteration of *omp for* loop is assigned to a GPU thread
  - Each section in *omp sections* is assigned to a GPU thread
- Data Mapping
  - OpenMP Shared and Threadprivate data → GPU global memory
  - OpenMP Private data → GPU registers or local memory

Compile-Time Stream Optimization (1)

- Parallel Loop Swap for Regular Case

```
#pragma omp for
  for (i=0; i<100; i++)
    for (j=0; j<100; j++)
      a[i][j] = b[i][j];
```

→

```
#pragma omp for
  for (i=0; i<100; i++)
#pragma omp for
    for (j=0; j<100; j++)
      a[i][j] = b[i][j];
```

↙

```
  for (i=0; i<100; i++)
#pragma omp for schedule(1)
    for (j=0; j<100; j++)
      a[i][j] = b[i][j];
```

→

```
for (i=0; i<100; i++)
  for (tid=0; tid<100; tid++)
    a[i][tid] = b[i][tid];
```

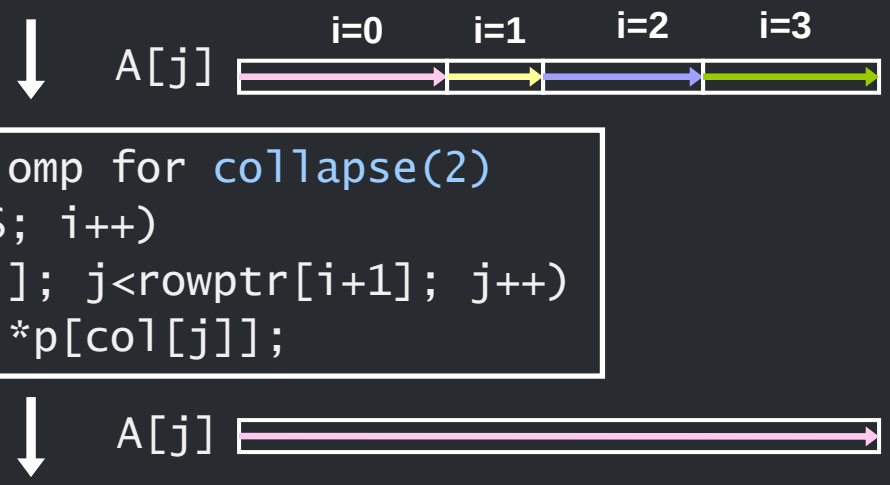
Compile-Time Stream Optimization (2)

- Loop Collapse for Irregular Case

```
#pragma parallel omp for
for (i=0; i<NROWS; i++)
  for (j=rowptr[i]; j<rowptr[i+1]; j++)
    l_w[i] = A[j]*p[col[j]];
```

check if i, j are incremented by one and rowptr are monotonic

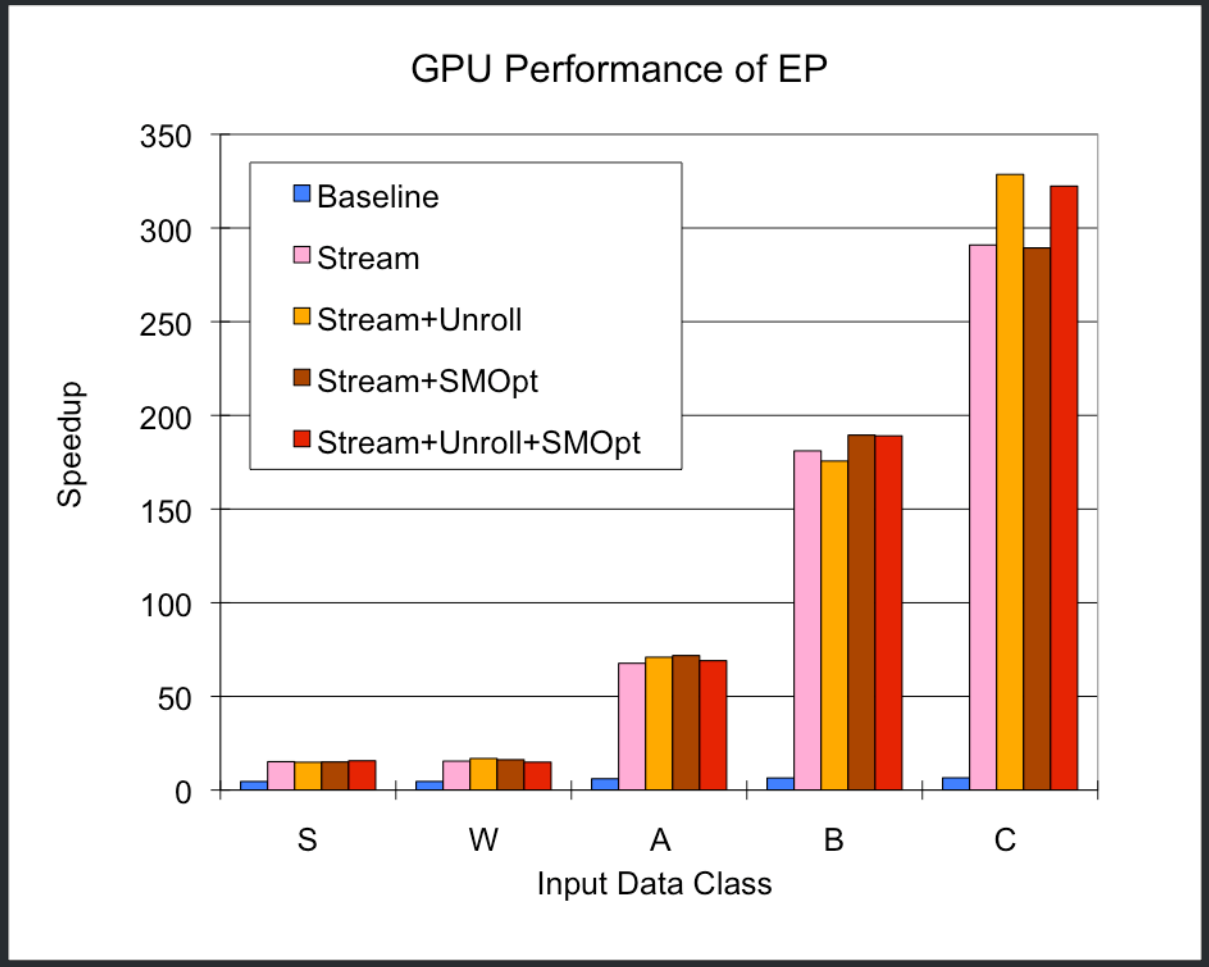
Does A[j] have stride-1 access in this doubly nested-loop?



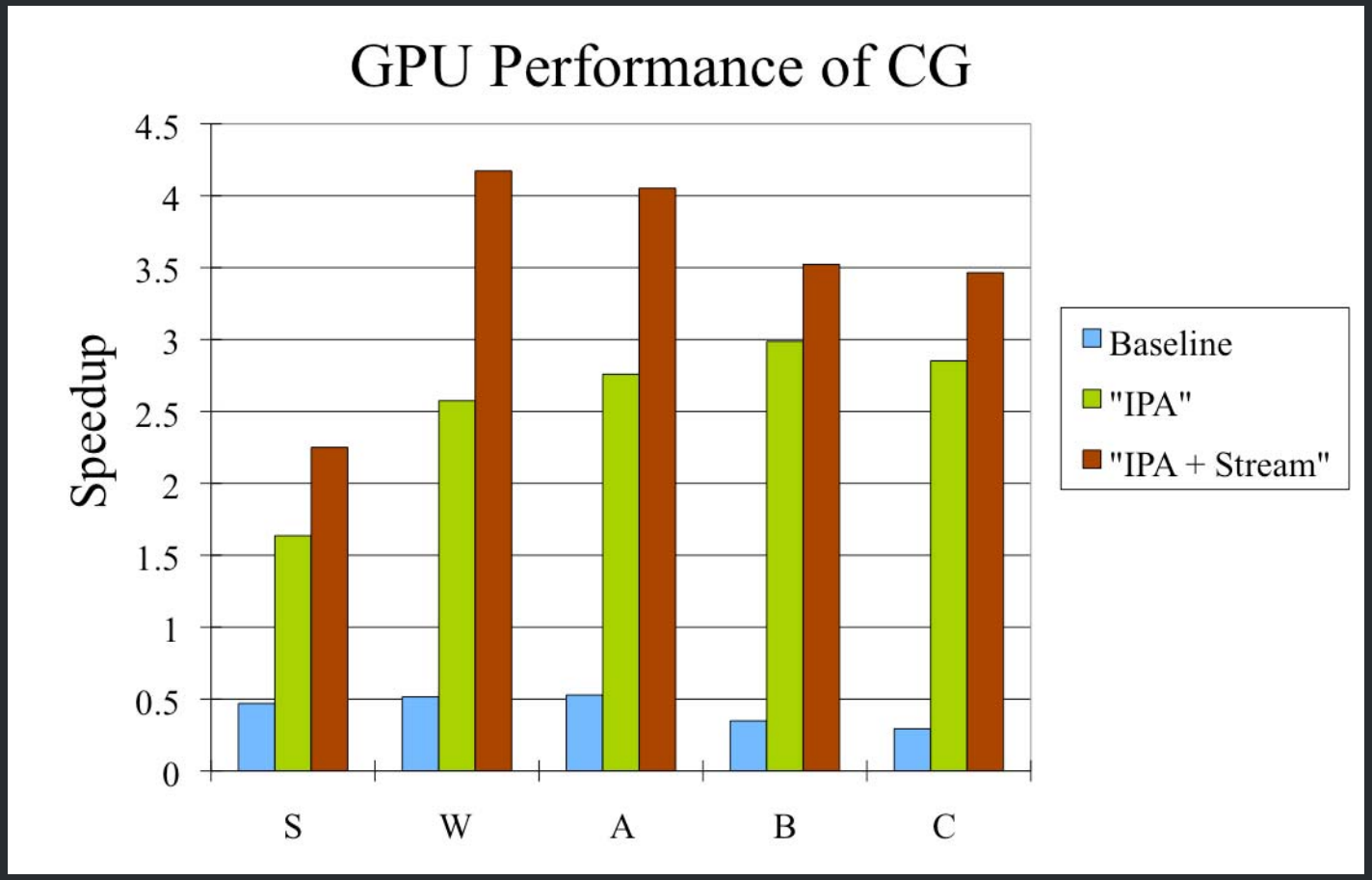

```
#pragma parallel omp for collapse(2)
for (i=0; i<NROWS; i++)
  for (j=rowptr[i]; j<rowptr[i+1]; j++)
    l_w[i] = A[j]*p[col[j]];
```

```
for (tid1=0; tid1<rowptr[NUM_ROWS]; tid1++)
  l_w[tid1] = A[tid1]*p[col[tid1]];
```

Performance - Regular Case



Performance – Irregular Case



Conclusion

- We have described a compiler framework toward automatic translation of OpenMP programs to CUDA GPGPU programs
- We have identified key transformation techniques to enable efficient GPU global memory accesses: *parallel loop swap* for regular codes and *loop collapsing* for irregular codes.
- Experiment results from two kernels, JACOBI and SPMUL, and two NAS OpenMP Parallel Benchmarks, EP and CG, show that the proposed optimizations work well on both regular and irregular applications, leading to performance improvement of up to 50X over the baseline translation (up to 328X over serial)