Compiling for GPUs

Adarsh Yoga
Madhav Ramesh

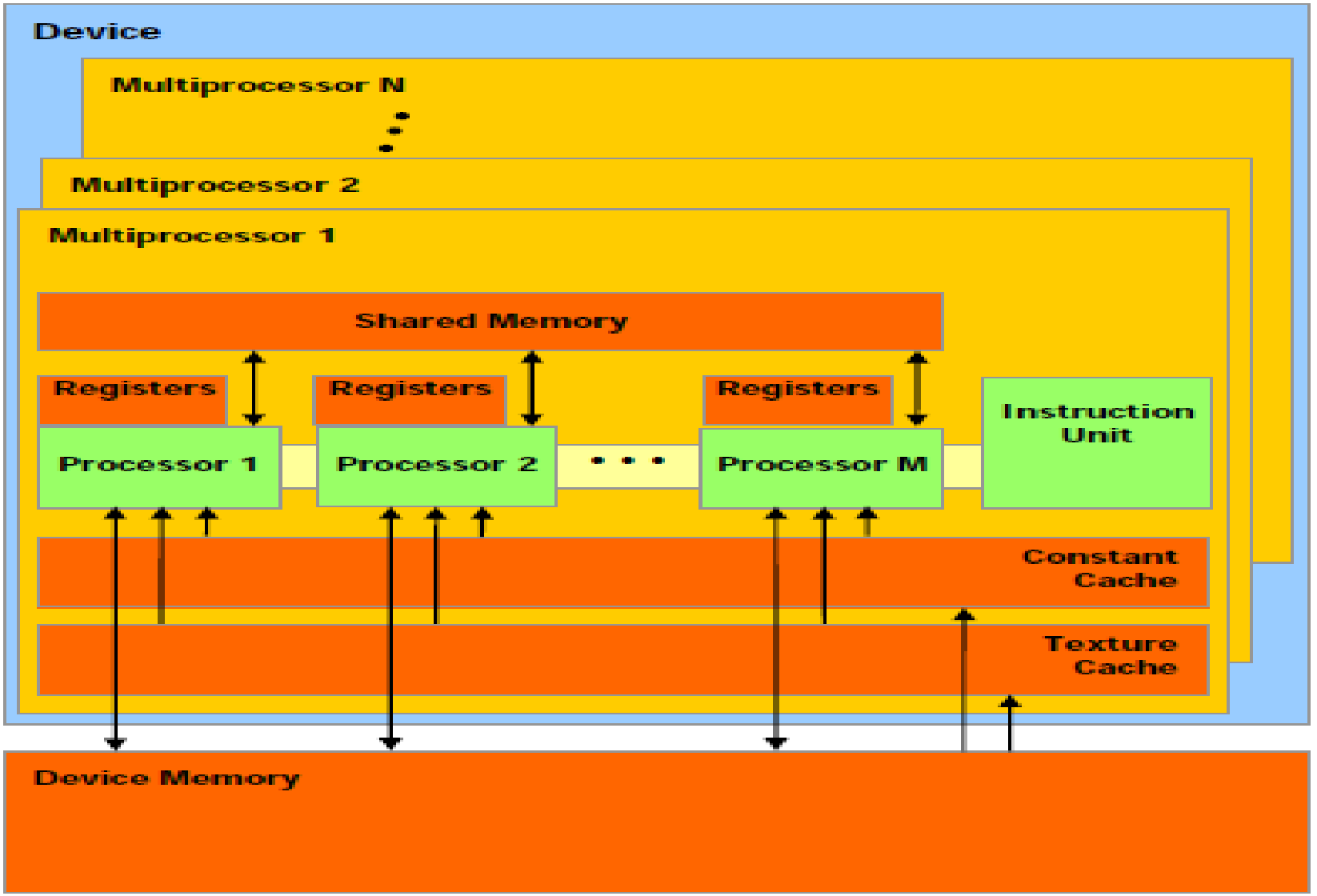
Agenda

- Introduction to GPUs
- Compute Unified Device Architecture (CUDA)
- Control Structure Optimization Technique for GPGPU
- Compiler Framework for Automatic Translation and Optimization from OpenMP to GPGPU

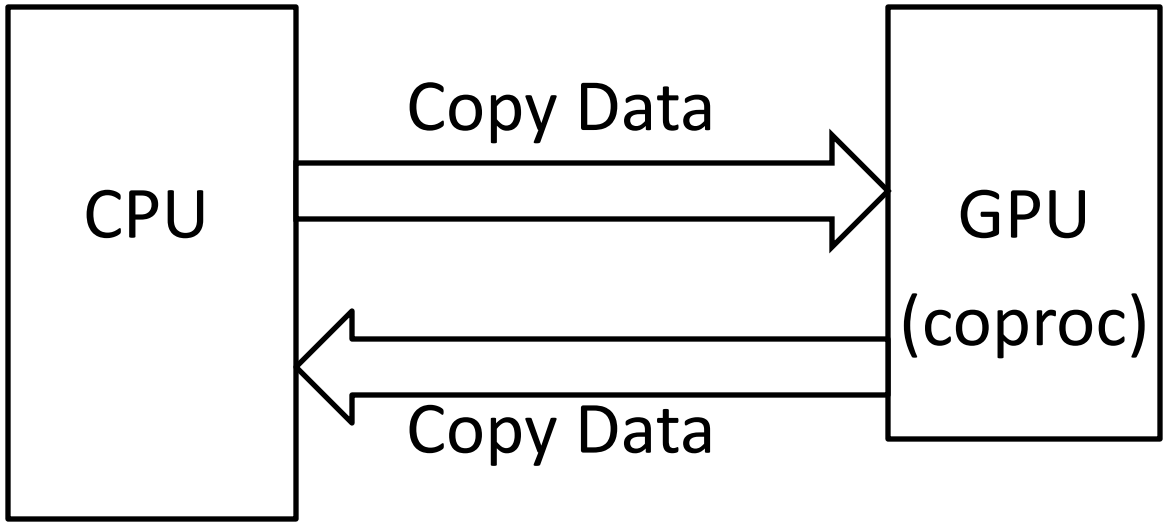
GPUs

- High performance many core processors
- Based on SIMD architecture – exploits data parallelism.
- Less control hardware is required since the since the same program is executed on many data elements.

GPU Hardware Model

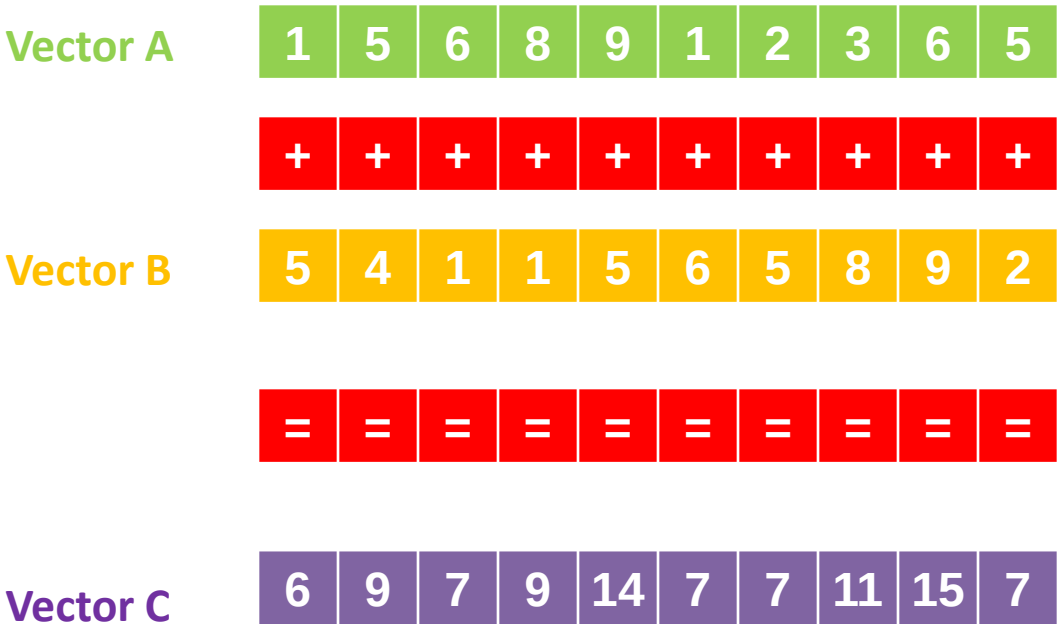


Programming Model



- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads.

Vector addition (Sequential code)



Vector addition (Sequential code)

```
#include <stdio.h>
#define SIZE 500

void VecAdd(float *A, float *B, float *C){
        int i;
        for(i=0;i<SIZE;i++)
                C[i] = A[i] + B[i]
}
void main(){
        int i, size = SIZE * sizeof(float);
        float *A, *B, *C;

        A = (float*)malloc(size);
        B = (float*)malloc(size);
        C = (float*)malloc(size);

        VecAdd(A,B,C);

        free(A);
        free(B);
        free(C);
}
```

Declare Function

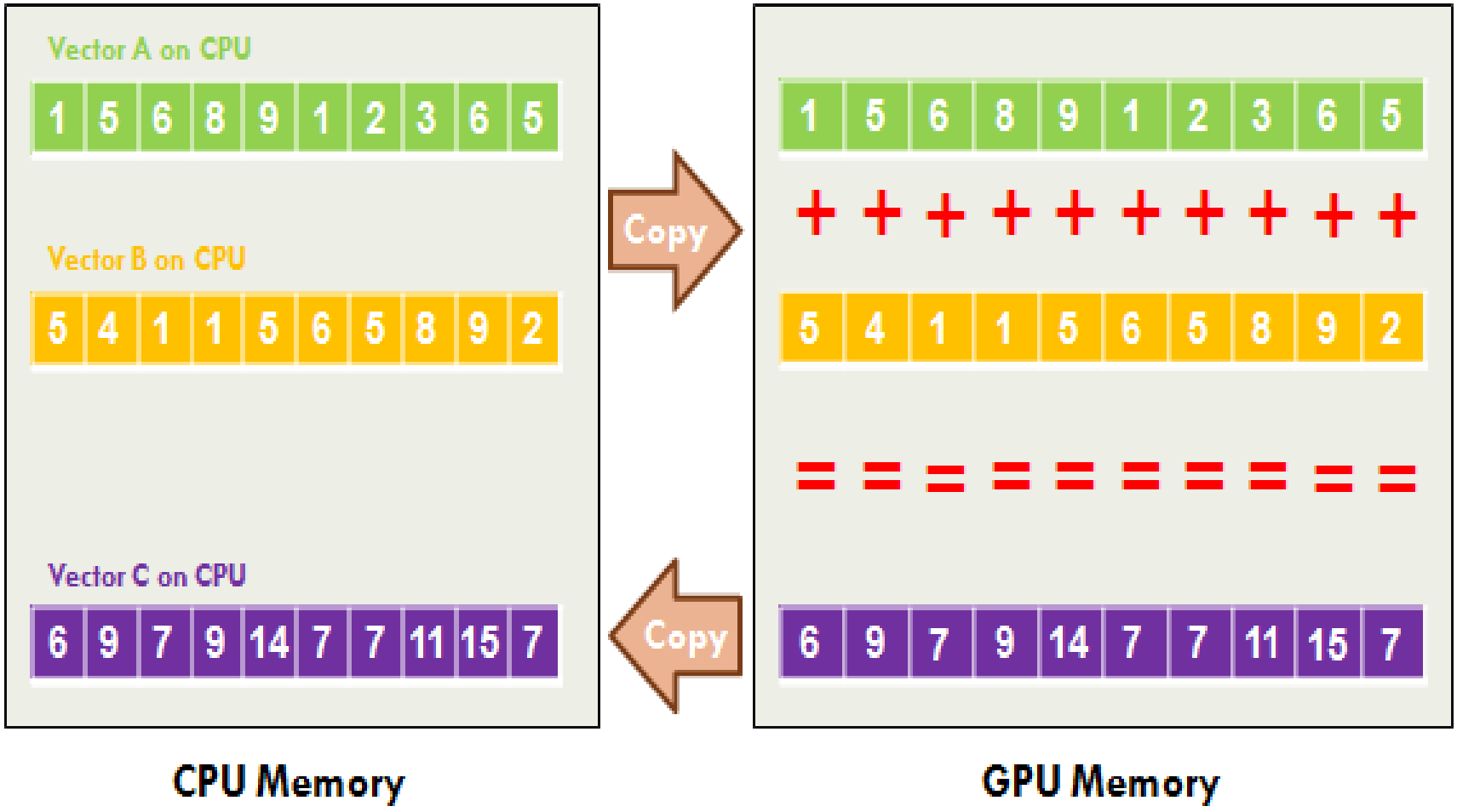
Declare Variables

Memory Allocate

Function Call

Memory De-Allocate

Vector Addition (GPU - CUDA)



Vector Addition (GPU - CUDA)

```
#include <stdio.h>
#define SIZE 500

__global__ void VecAdd(float* A, float* B, float* C){
   int idx = threadIdx.x;
   if(idx < SIZE)
        C[idx] = A[idx] + B[idx];
}
void main(){
        int i, size = SIZE * sizeof(float);
        float *h_A, *h_B, *h_C, *d_A, *d_B, *d_C;

        h_A = (float*)malloc(size);
        h_B = (float*)malloc(size);
        h_C = (float*)malloc(size);

        cudaMalloc((void**)&d_A, size);
        cudaMalloc((void**)&d_B, size);
        cudaMalloc((void**)&d_C, size);
```

Declare Kernel Function

Declare Variables

CPU Memory Allocate

GPU Memory Allocate

Vector Addition (GPU - CUDA)

```
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Data Transfer from CPU to GPU

```
VecAdd<<<1, SIZE>>>(d_A, d_B, d_C);
```

Kernel Call

```
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
```

Data Transfer from GPU to CPU

```
free(h_A);
free(h_B);
free(h_C);
```

CPU Memory De-Allocate

```
cudaFree(d_A);
cudaFree(d_B);
cudaFree(d_C);
```

GPU Memory De-Allocate

}

Control Structure Optimization Technique in GPGPU

Control-structure splitting

```
kernel(ptr1,ptr2,ptr3,ptr4,ptr_result){
   ...
   for i=0 to N
     x += ptr1[i] * ptr2[i];
     y += ptr3[i] / ptr4[i];
   end
   ...
}
```

Introduction

- CUDA programming framework made computational power of GPUs easier to utilize.
- But how to tune a CUDA program to work well on a GPU?
- The main optimizing task is finding the optimal numbers of threads and blocks that will keep the GPU occupied.

GPU Occupancy

- Occupancy is the number of threads running concurrently.

$$\text{Occupancy} = \frac{\text{Number of active threads}}{\text{Total number of threads}} \times 100$$

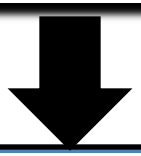
- Limit of occupancy – number of registers each thread requires.
- Control structures pose serious challenges to the GPU occupancy.

Splitting Technique

- CUDA compiler tries to optimize single-thread performance while ignoring overall resource pressure.
- Two techniques called loop and branch splitting which smartly increase code redundancy.
- The idea is to free the hardware resources by purposefully increasing the code size.

Loop Splitting

```
kernel(ptr1,ptr2,ptr3,ptr4,ptr_result){
  ...
  for i=0 to N
    x += ptr1[i] * ptr2[i];
    y += ptr3[i] / ptr4[i];
  end
  ...
}
```



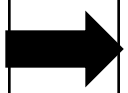
```
kernel(ptr1,ptr2,ptr3,ptr4,ptr_result){
  ...
  for i=0 to N
    x += ptr1[i] * ptr2[i];
  end
  for i=0 to N
    y += ptr3[i] / ptr4[i];
  end
  ...
}
```

Loop Splitting

- Optimization can be done when loop body contains multiple independent operations relying on different inputs.
- Leads to smaller loop bodies and hence reduces loop register pressure.
- In the example considered, at least two registers are freed which leads to an increase in occupancy.

Branch Splitting

```
branchedkenrnel(){
  load decision mask
  load input data used by both branches
  if decision mask[tid] == 0
    load input data for if branch
    perform calculations using 6 registers
  else if decision mask[tid] == 1
    load  input data for else branch
    perform calculations using 13 register
  end if
}
```



```
ifkernel(){
  load decision mask
  if decision mask[tid] == 0
    load all input data
    perform calculations using 6 registers
  end if
}

elsekernel(){
  load decision mask
  if decision mask[tid] == 1
    load all input data
    perform calculations using 13 register
  end if
}
```

Branch Splitting

- When one branch of the “if” statement has lower occupancy the whole “if” statement will always run with that occupancy.
- This technique splits a branch of the initial kernel into two kernels.
- Rules to determine if branch splitting is of any benefit:
  - Does not run at 100% occupancy
  - And contains two or more branches
  - And utilize different amounts of hardware

Speedup – Branch splitting

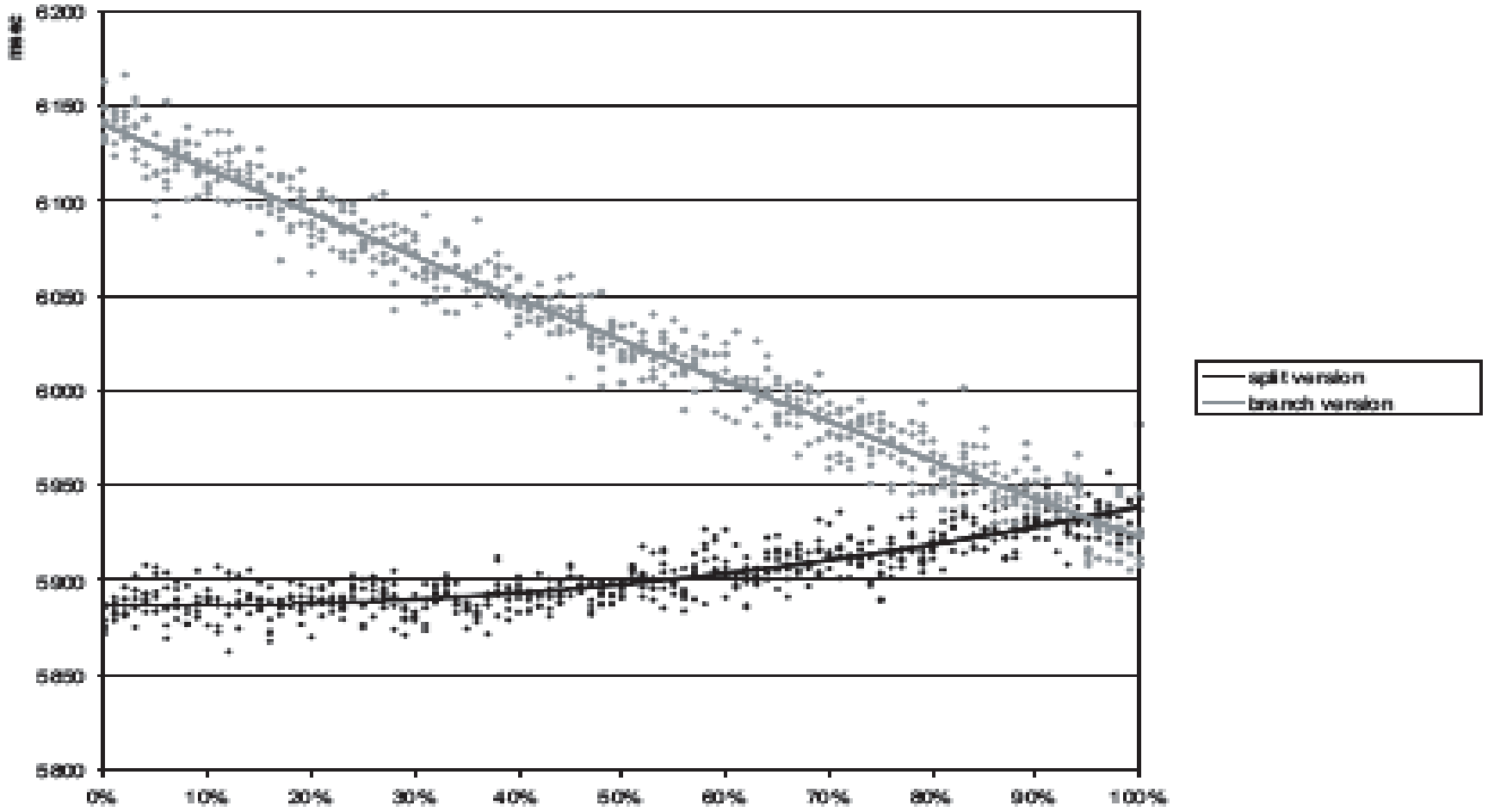
$$speedup = \frac{T}{\sum_{i=1}^{n} \frac{t_i}{\frac{\rho_i}{\rho_{\min}}} + overhead}$$

- Where T is the runtime of the worst case of the branched version.
- ρ_i is the occupancy possible for branch i when run on its own.
- $\rho_{\min}$ is the occupancy the branched version gets executed with.

Synthetic Benchmark

- Synthetic benchmark with full control over the branch using decision mask to show performance changes.
- Fewer registers used by “If” branch than “Else” branch. So possible occupancy of 100% and 67% respectively.
- Overall runtime of the kernel execution was measured over a fixed data set.

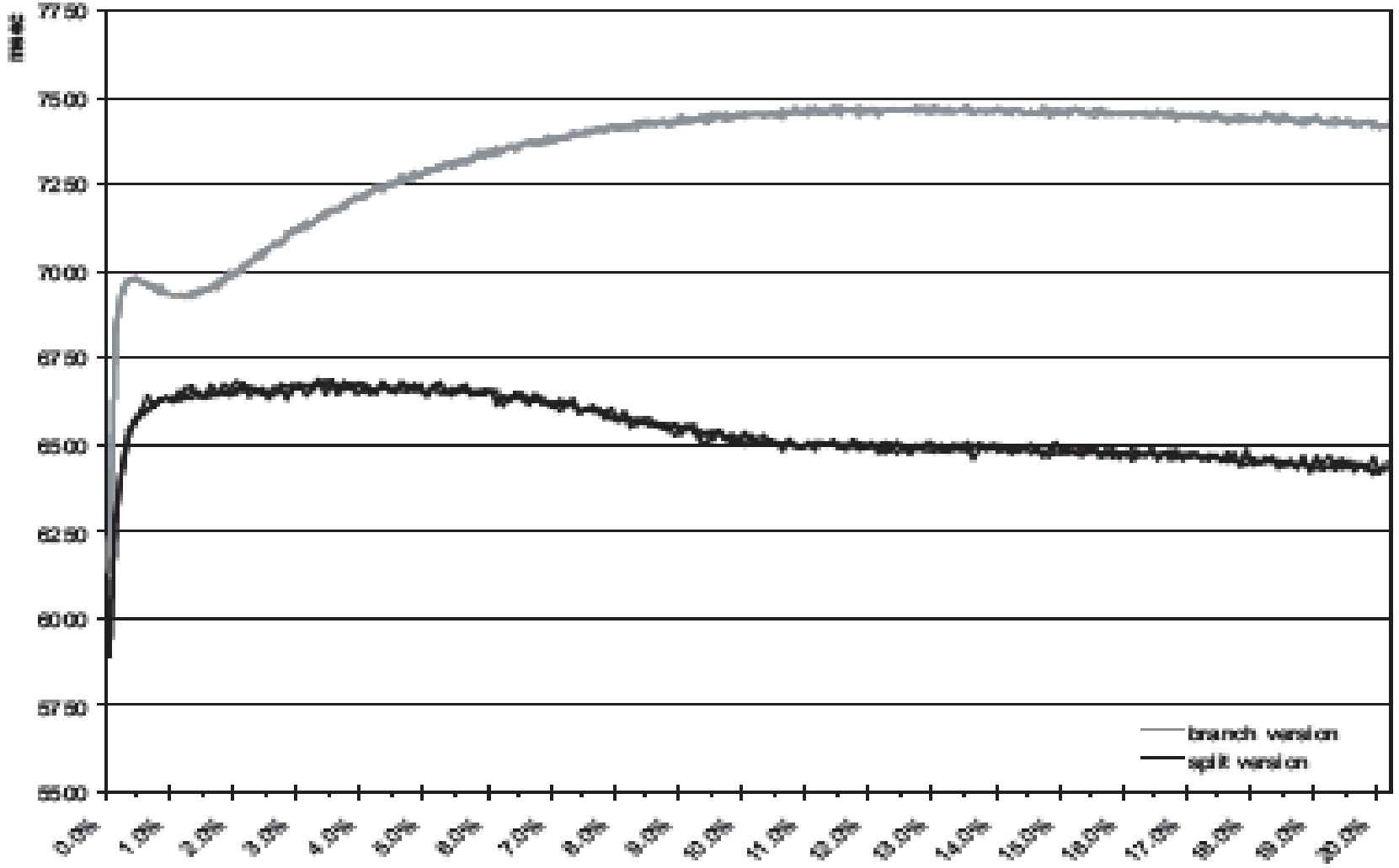
Linear Decision Mask



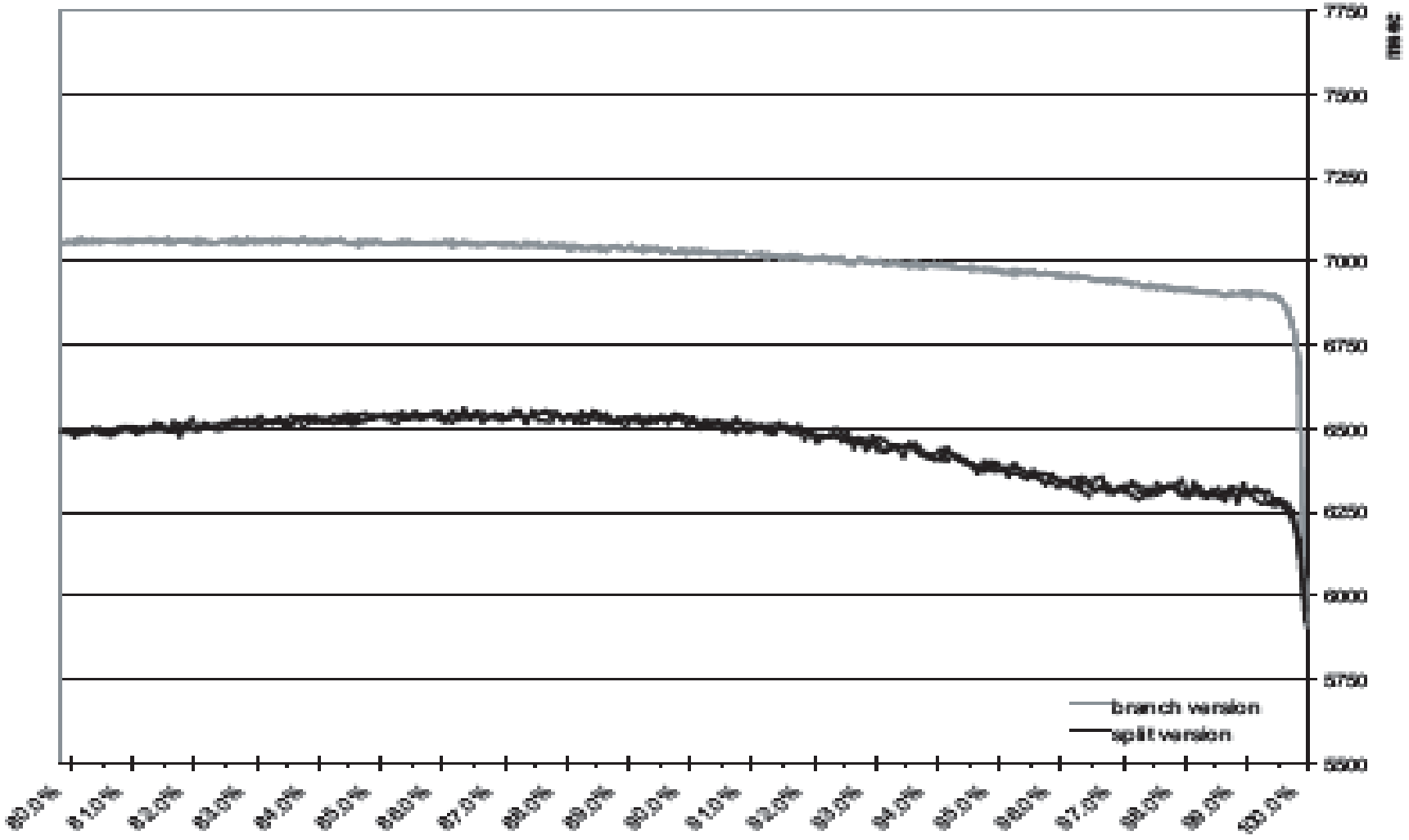
Linear Decision Mask

- For the first iteration, the branch-version executes only the “if” branch but with 67% occupancy.
- Whereas the split version drops out the else kernel and hence runs at 100% occupancy.
- For the last iteration the overhead of additional kernel invocations and loads leads to lower performance of the split version.

Random Decision Mask



Random Decision Mask



Random Decision Mask

- The drop in the performance after the extreme cases is due to the serialization of the branch statement.

- The split version performs better because of two reasons.
  - Reduction in the serialization of branches.
  - Lowering the resource usage per thread.

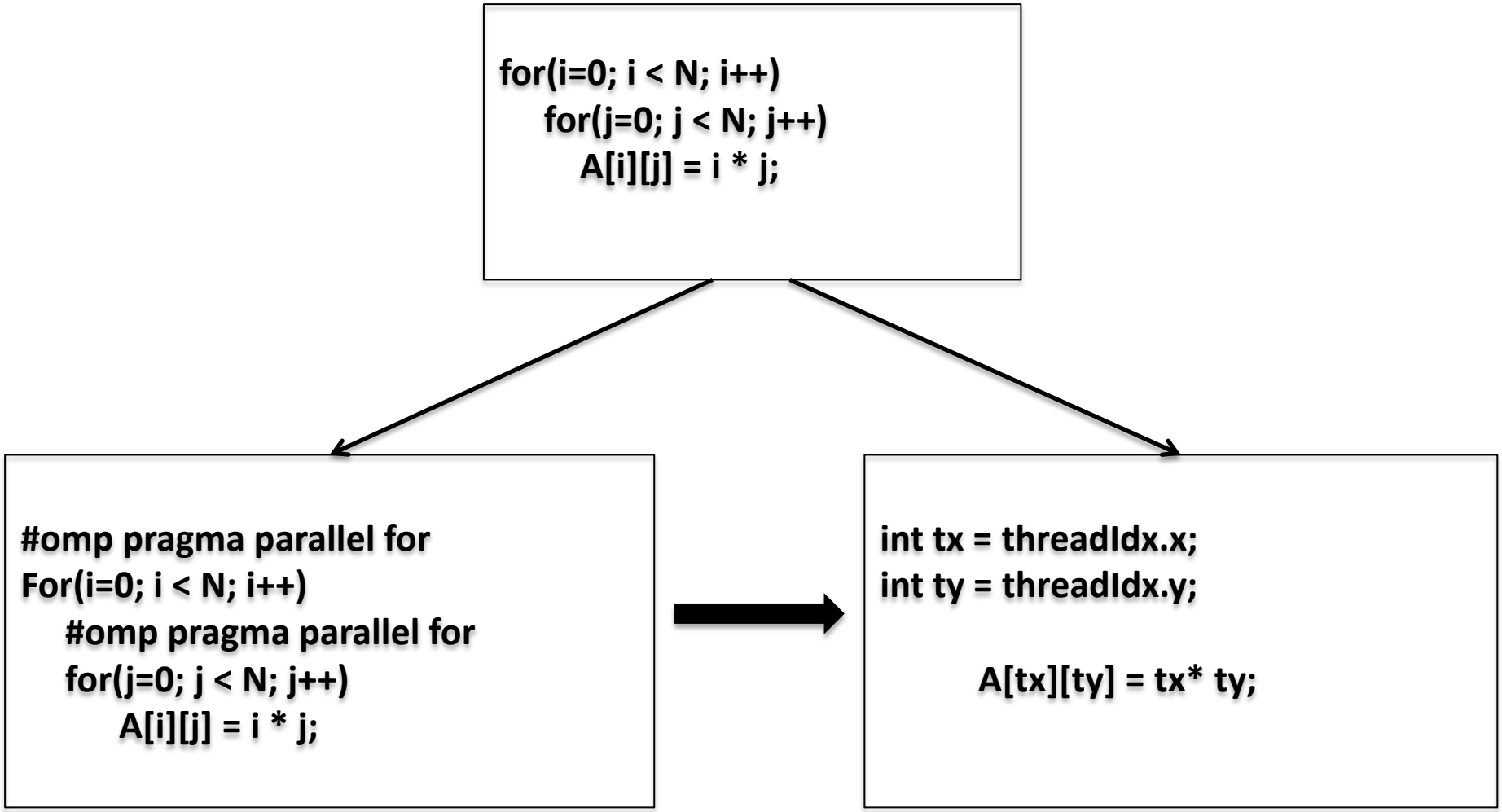
Compiler Framework for Automatic Translation and Optimization from OpenMP to GPGPU

OpenMP to GPU compilation

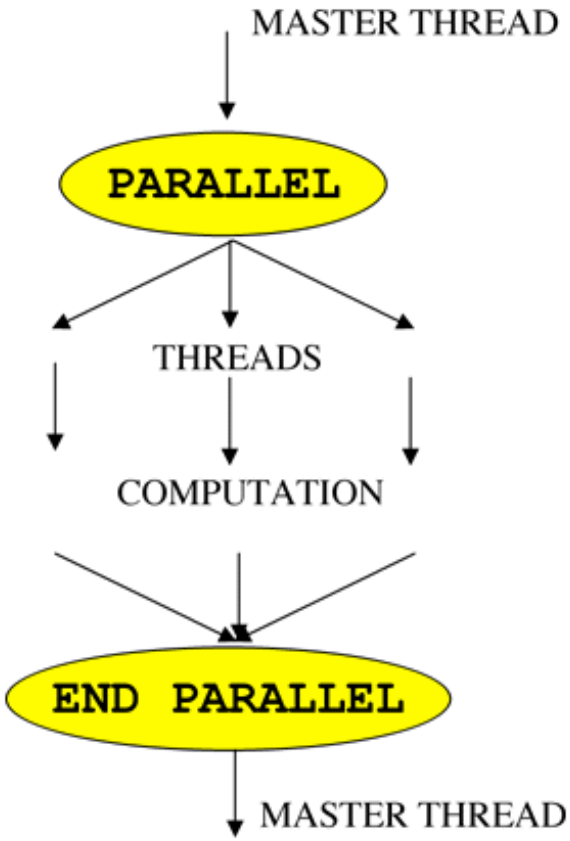
- Improve programmability.
- OpenMP to GPU translator.
- Automatic Source to Source translation of standard OpenMP applications to CUDA based GPGPU applications.
- Significant similarities between the working of OpenMP and CUDA.
- Popular extension for Shared Memory programming.

Why choose OpenMP?

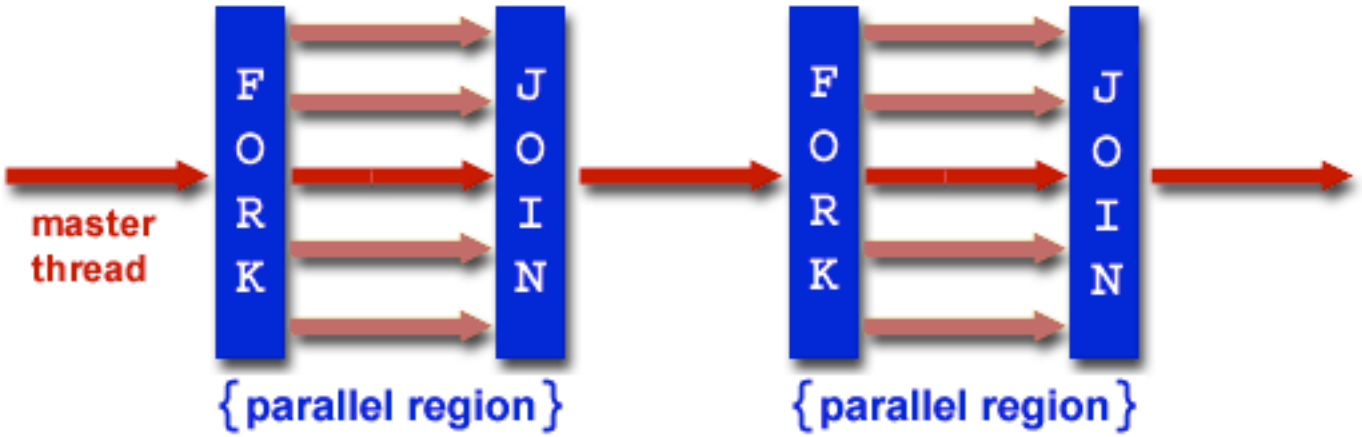
- Efficient in expressing loop level parallelism.



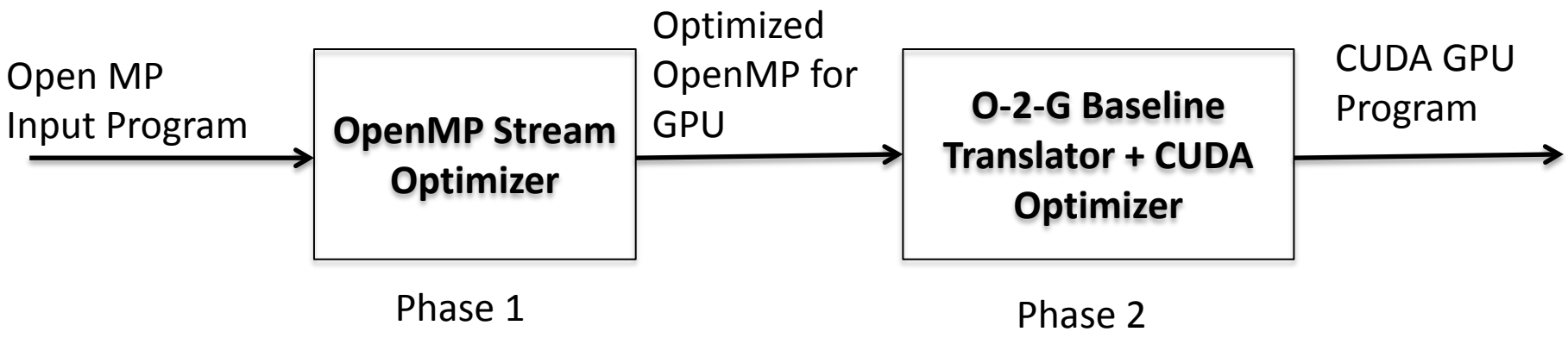
- OpenMP's fork-join model is similar to Master threads running in host CPU and a pool of threads in the GPU device.



- Incremental Parallelization of Applications.
- OpenMP allows multiple parallel regions.
- CUDA allows call to multiple kernels.



The Compilation System



Baseline Translation

1) Interpret the OpenMP semantics to CUDA.
2) Identify the “kernel regions”.
3) Transform the “kernel regions” to GPU kernel functions.
4) Insert necessary memory transfer call by analysing the shared data that would be accessed by the GPU.

Interpretation of OpenMP semantics

1) Parallel Constructs

- ***omp parallel***
  - Fundamental constructs to specify parallel regions .
  - Candidate kernel regions.
  - Transformed to GPU functions.

2) Work Sharing Constructs

- ***omp parallel, omp for***
  - Used to partition work among threads on a GPU device.
  - Each iteration of an *omp for* is assigned to a thread.
  - Each section in an *omp sections* is assigned to a thread.

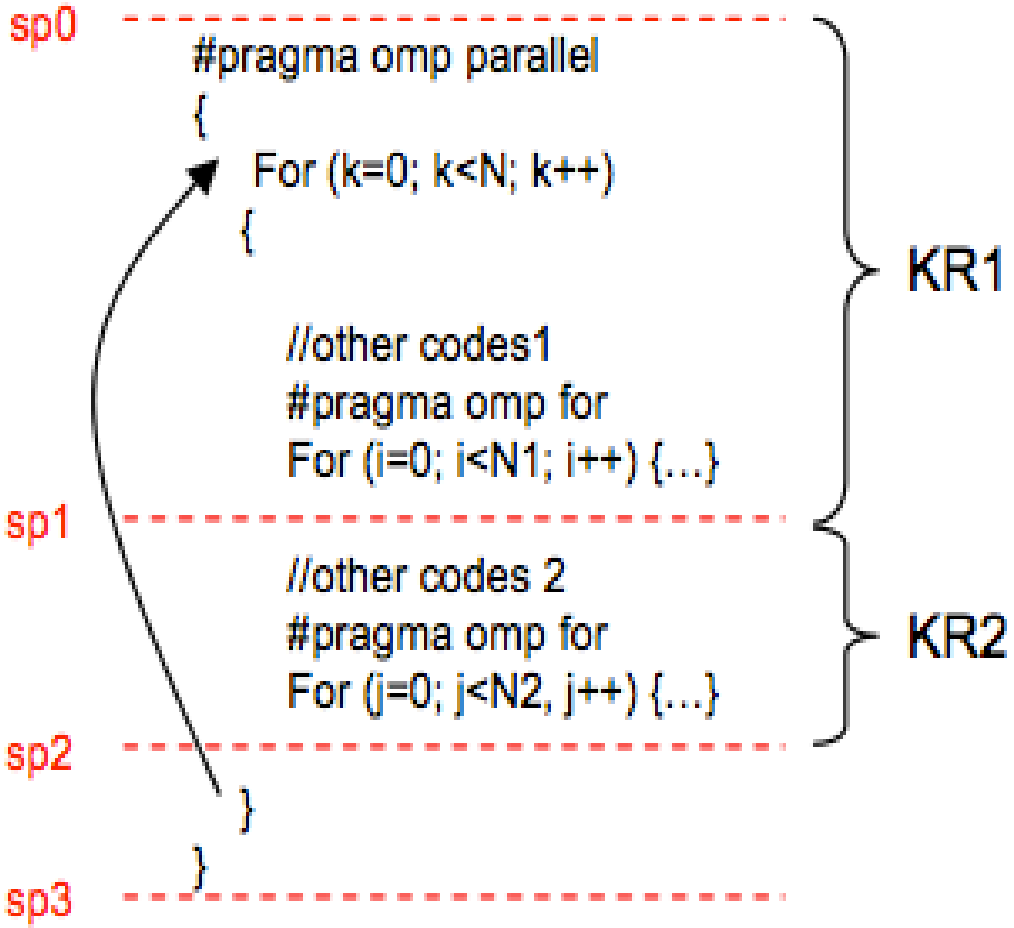
Transforming a kernel region to a CUDA Kernel

- Identified Kernel regions placed inside CUDA Kernels
- Work partitioning
  - Partition loop iterations
  - Map *omp sections*
  - Determine redundant sections
  - Calculate # of threads in a block and # threads blocks
- Data Mapping
  - Map shared data to global or private memory
  - Insert memory allocation(cudaMalloc) and transfer calls(CudaMemcpy)
  - What to copy from the host to the device, and vice versa?
  - Is it necessary to move back all data?

Identifying Kernel regions

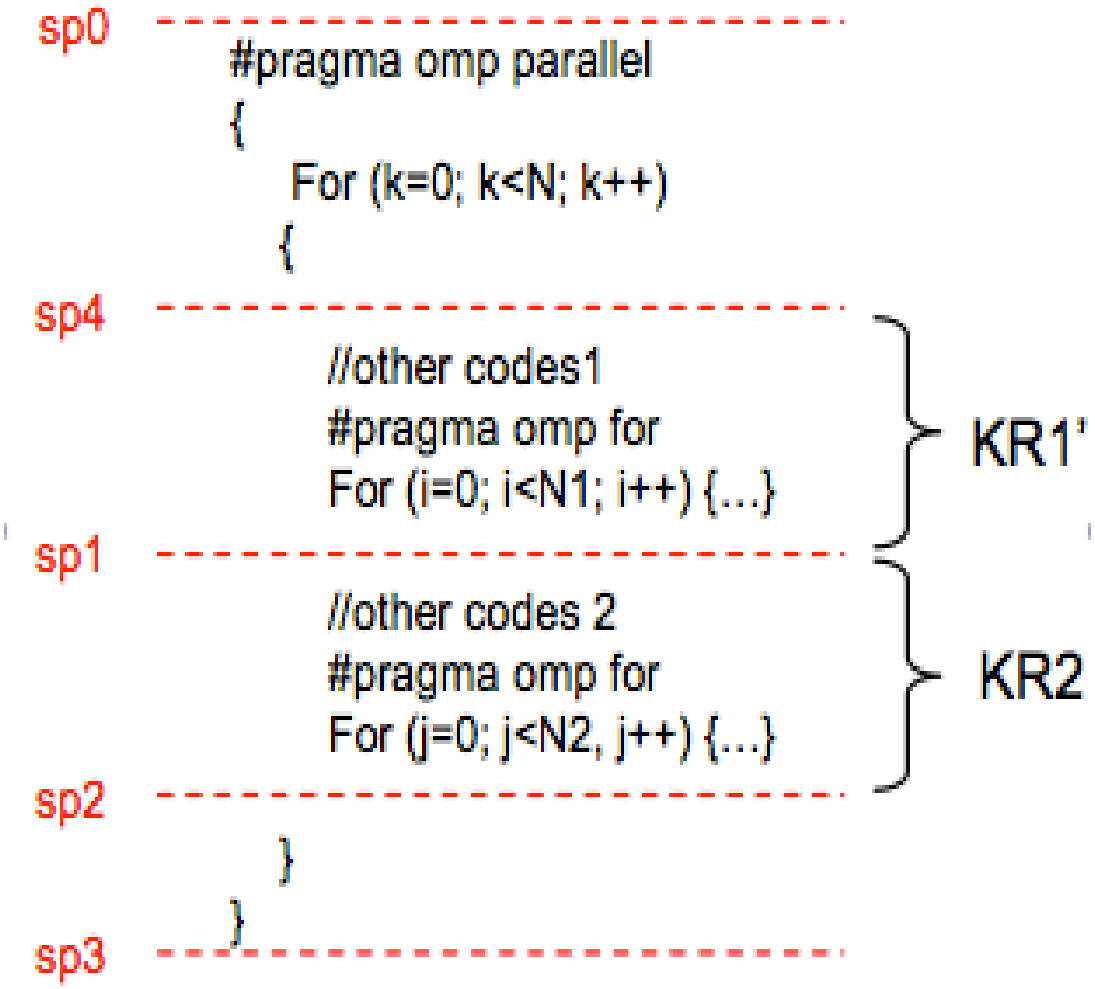
- OpenMP parallel regions are potential parallel regions.
- Split in between synchronization constructs.
- Sub-regions having atleast one work-sharing construct become kernel regions.
- Split regions decide the number of possible kernel calls.
- Chance of a split region lying within control structures.
- Results in unstructured blocks as we are going to see.

A Problem?



(a) Initial split at synchronization points

Dealing with unstructured blocks



(b) Further split at multiple entry points

Compiler Optimizations

- Mere Baseline translations from OpenMP to CUDA may not yield optimum results
- OpenMP
  - Coarse grained parallelism
- CUDA
  - Fine grained parallelism
- Compiler's OpenMP optimizer transforms CPU-style OpenMP code to suit execution on GPUs.

Jacobi Iteration

Sequential Code:

```
for(i = 1; i < SIZE; i++)  {
    for(j = 1; j < SIZE; j++)  {
       x_j[i][j] = ( x[i][j-1] + x[i][j-1] + x[i-1][j] + x[i+1][j] )/4;
    }
 }
```

GPU Code:

```
__shared__ float u_sh[BLOCK_SIZE][BLOCK_SIZE];
x[tx][ty] = x_d[x+y*BLOCK_SIZE];

if(tx > 0 && ty > 0 && tx < BLOCK_SIZE && ty < COLS_IN_BLOCK-1)
   {
      x_d[tx + ty * BLOCK_SIZE] =  (u_sh[tx+1][ty] + u_sh[tx-1][ty]
                                    + u_sh[tx][ty+1] + u_sh[tx][ty-1])/4;
   }
```

Parallel Loop Swap on Jacobi

```
#pragma omp parallel for
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;
}
```

(a) input OpenMP code

```
#pragma omp parallel for
for (i=1; i<=SIZE; i++) {
#pragma cetus parallel
  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;
}
```

(b) Cetus-parallelized OpenMP code

```
#pragma omp parallel for schedule(static, 1)
for (j=1; j<=SIZE; j++)
  for (i=1; i<=SIZE; i++) {
    a[i][j] = (b[i-1][j] + b[i+1][j]
               + b[i][j-1] + b[i][j+1])/4;
}
(c) OpenMP output by OpenMP stream optimizer
```

```
// tid is a GPU thread identifier
for (tid=1; tid<=SIZE; tid++)
  for (i=1; i<=SIZE; i++) {
    a[i][tid] = (b[i-1][tid] + b[i+1][tid]
                 + b[i][tid-1] + b[i][tid+1])/4;
}
```

(d) internal representation in O2G translator

```
// Each iteration of the parallel-for loop
// is cyclic-distributed to each GPU thread
if (tid<=SIZE) {
  for (i=1; i<=SIZE; i++) {
    a[i][tid] = (b[i-1][tid] + b[i+1][tid]
              + b[i][tid-1] + b[i][tid+1])/4;
  }
}
```

(e) GPU code

O-2-G CUDA Optimizations

- Specific to CUDA architecture
- Frequent references to Global memory is slow
- A Few optimizations are possible

1) Caching of Frequently Accessed Global data

  - Access to global memory is not cached
  - To exploit temporal locality, treat shared memory or registers for faster memory access
  - Constant memory- a read only memory, faster than global

2) Memory Transfer Reductions

- Insertions of memory transfer calls for the shared and threadprivate data
- Remove unnecessary transfers of data to and from GPU
- Data-flow analysis

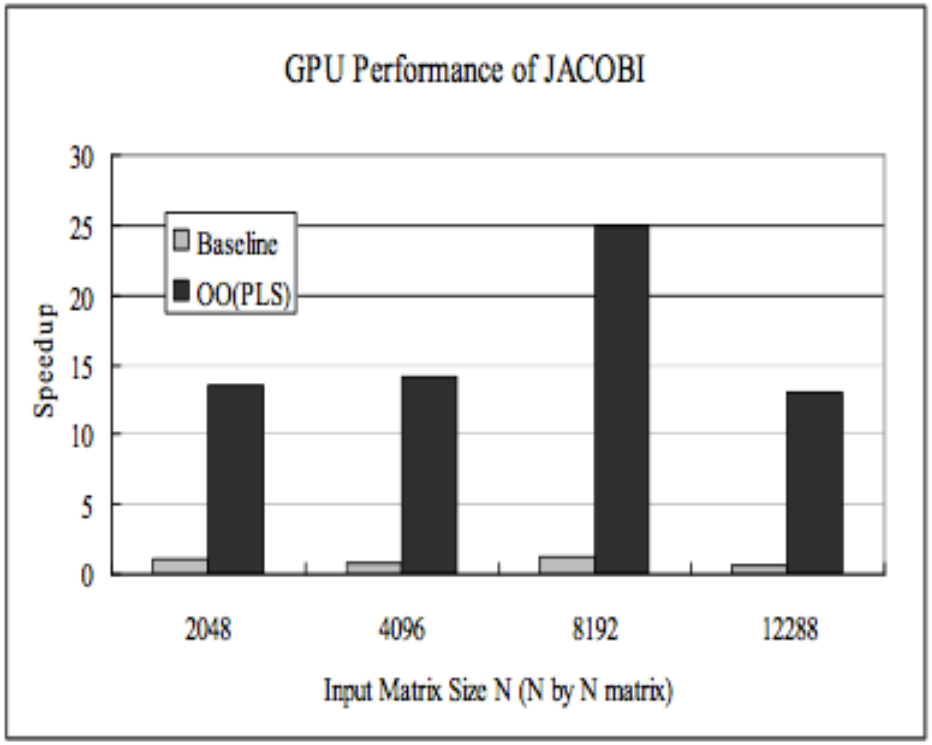
Algorithm:

For each kernel region,
Determine UseSet- set of shared data read in the kernel
Determine DefSet – set of shared data written in the kernel

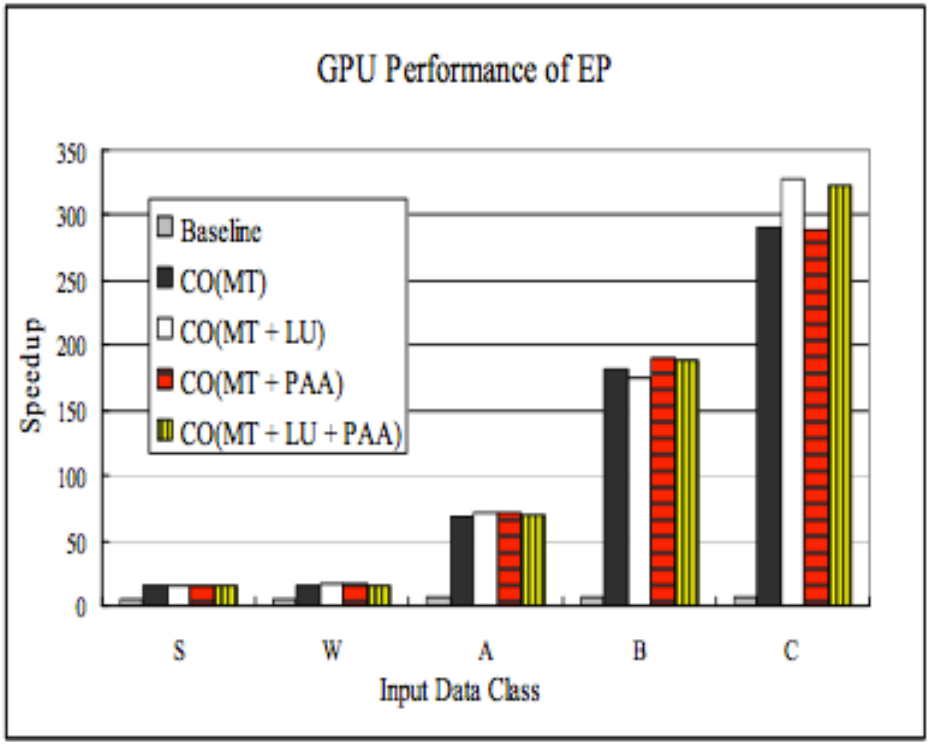
For Each variable in UseSet, if data is defined in the host code, copy data in this variable from Host to GPU

For Each variable in DefSet, if data is used in host code after the kernel call, copy it from GPU to the Host

Performance of Regular Applications



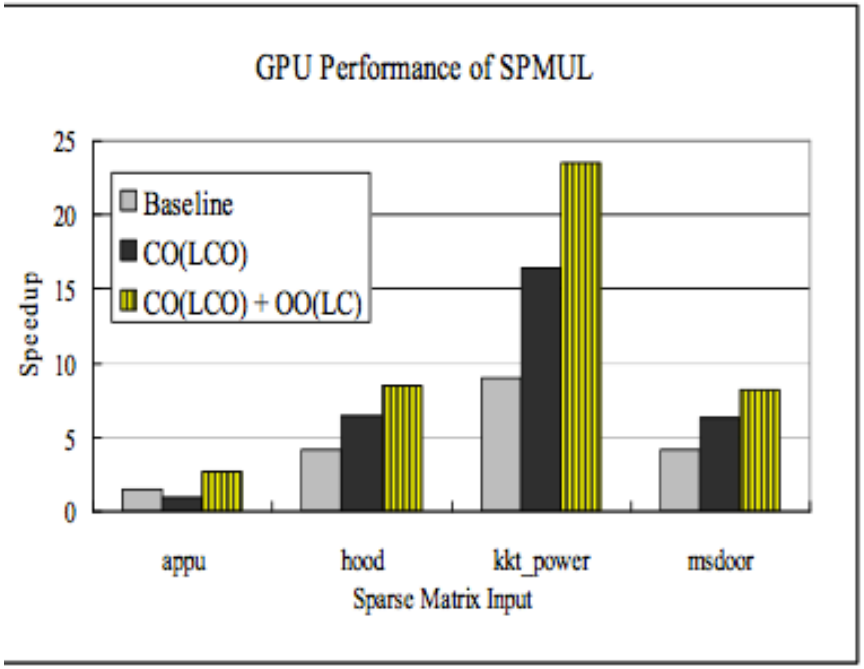
(a) JACOBI kernel



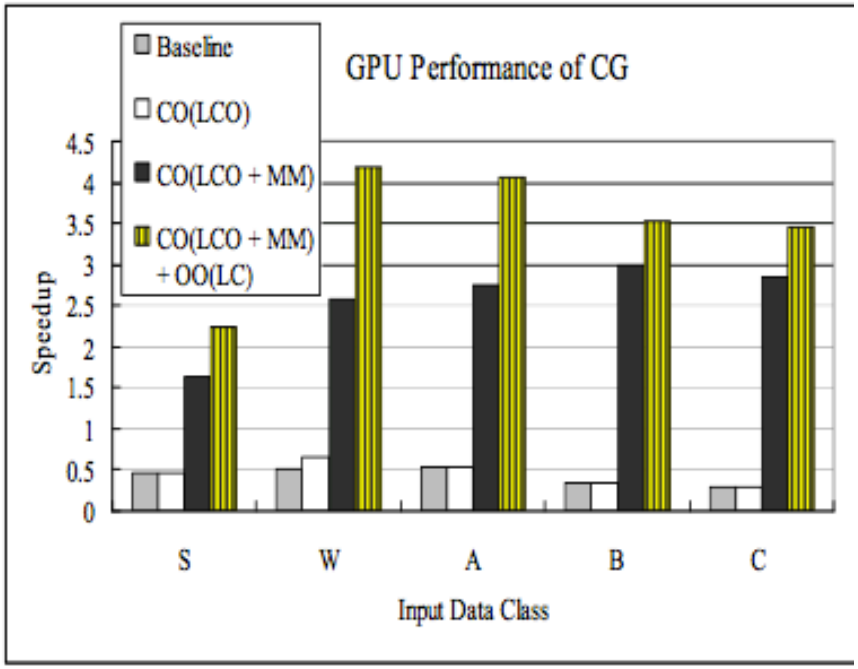
(b) NAS Parallel Benchmark EP

Figure 6. Performance of Regular Applications (speedups are over serial on the CPU). *Baseline* is the baseline translation without optimization; the other bars measure the following OpenMP optimizations (*OO*) or CUDA optimizations (*CO*): *Parallel Loop-Swap* (*PLS*), *Matrix Transpose* (*MT*), *Loop Unrolling* (*LU*), and private array allocation on *shared memory* using array expansion (*PAA*). The performance irregularity shown in the left figure is caused by reproducible, inconsistent behavior of the CPU execution; the average CPU execution time per array element is longer at $N = 8192$ than others.

Performance of Irregular Applications



(a) SPMUL kernel



(b) NAS Parallel Benchmark CG

igure 7. Performance of Irregular Applications (speedups are over serial on the CPU). *Baseline* is the baseline translation witho ptimization; the other bars represent the following OpenMP optimizations (*OO*) or CUDA optimizations (*CO*): *LCO* is a local cachi ptimization for global data, *LC* means the *loop collapsing* transformation technique, and *MM* represents a redundant memory transf inimizing optimization.

References

- OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization
  - Seyong Lee, Seung-Jai Min, Rudolf Eigenmann
- Control Structure Splitting Optimization for GPGPU
  - Snaider Carillo, Jakob Siegel, Xiaoming Li
- Nvidia Programming Guide
- Introduction to OpenMP
  - From Lawrence Livermore National Laboratory

Synchronization Constructs

- ***omp barrier, omp flush, omp critical etc..***
  - Constitute *Split points*
  - One *Split point* divides a parallel region to 2 sub-regions
  - Each sub-region now a kernel region

Why have split points in the inside the parallel region?

- Enforce a global synchronization in CUDA

- No efficient global synchronization in CUDA

- _syncthreads() is limited to a block

- One solution is to have multiple kernels

Directives for specifying data properties

- ***omp shared, omp private, etc..***
  - Map data into GPU memory spaces
  - Explicit memory transfers to specific regions in the memory
  - Shared data can be mapped to global memory
  - Private data can be seen only by a single thread(local, global)

- Why is Shared Memory not mapped to shared memory in the CUDA programming model?

- Shared data in Open MP can be seen by all threads
- CUDA -> visible only to a thread block.

Loop Collapsing

- An optimizing technique to handle irregular applications
- Irregular applications are those that have dependencies with outer loops or that have indirect memory access
- Example of an irregular application

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

- Can be applied to loops without dependencies or indirect memory accesses

Working of loop collapsing

```
int a[50][100];
#pragma omp parallel for
for(int i=0; i<50; i++)
   for(int j=0; j<100; j++)
      a[i][j] = 0;
```

(a) Input OpenMP code

```
int a[5000];
For(tid=0; tid<5000; tid++)
  a[tid] = 0;
```

(c) Internal representation in O2G translator

```
int a[50][100];
#pragma omp parallel for
#pragma omp for collapse(2)
for(int i=0; i<50; i++)
   for(int j=0; j<100; j++)
      a[i][j] = 0;
```

(b) o/p of OMP stream Optimizer

```
int a[5000];
If(tid<5000)
   a[tid] = 0;
```

(d) GPU Code

What goes into a kernel region?

- Simple translation scheme to convert all work-sharing constructs into kernel functions.
- Work-sharing constructs contain true parallel code. Constitute the kernel regions.
- Other sub-regions outside these constructs but inside the *omp parallel* are executed by a single thread (*omp master, omp single*).
- Some run serially(omp ordered, omp critical)
- Redundant execution allowed to reduce expensive memory calls.