

PARALLEL COMPUTING		Semester	VII
Course Code	BCS702	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab slots	Total Marks	100
Credits	04	Exam Hours	03
Examination nature (SEE)	Theory/Practical		

Course outcomes (Course Skill Set):

At the end of the course, the student will be able to:

- Explain the need for parallel programming
- Demonstrate parallelism in MIMD system.
- Apply MPI library to parallelize the code to solve the given problem.
- Apply OpenMP pragma and directives to parallelize the code to solve the given problem
- Design a CUDA program for the given problem.

MODULE-5

GPU programming with CUDA - GPUs and GPGPU, GPU architectures, Heterogeneous computing, Threads, blocks, and grids Nvidia compute capabilities and device architectures, Vector addition, Returning results from CUDA kernels, CUDA trapezoidal rule I, CUDA trapezoidal rule II: improving performance, CUDA trapezoidal rule III: blocks with more than one warp.

GPUs and GPGPU

In the late 1990s and early 2000s, the computer industry responded to the demand for highly realistic computer video games and video animations by developing extremely powerful **graphics processing units** or **GPUs**.

These processors, as their name suggests, are designed to improve the performance of programs that need to render many detailed images.

One of the biggest difficulties faced by the early developers of GPGPU was that the GPUs of the time could only be programmed using computer graphics APIs, such as Direct3D and OpenGL.

GPU architectures

if (x [i] >= 0)

x [i] += 1;

else |

x [i] -= 2;

In a typical SIMD system, each datapath carries out the test $x[i] \geq 0$. Then the datapaths for which the test is true execute $x[i] += 1$, while those for which $x[i] < 0$ are *idle*. Then the roles of the datapaths are reversed: those for which $x[i] \geq 0$ are idle while the other datapaths execute $x[i] = 2$.

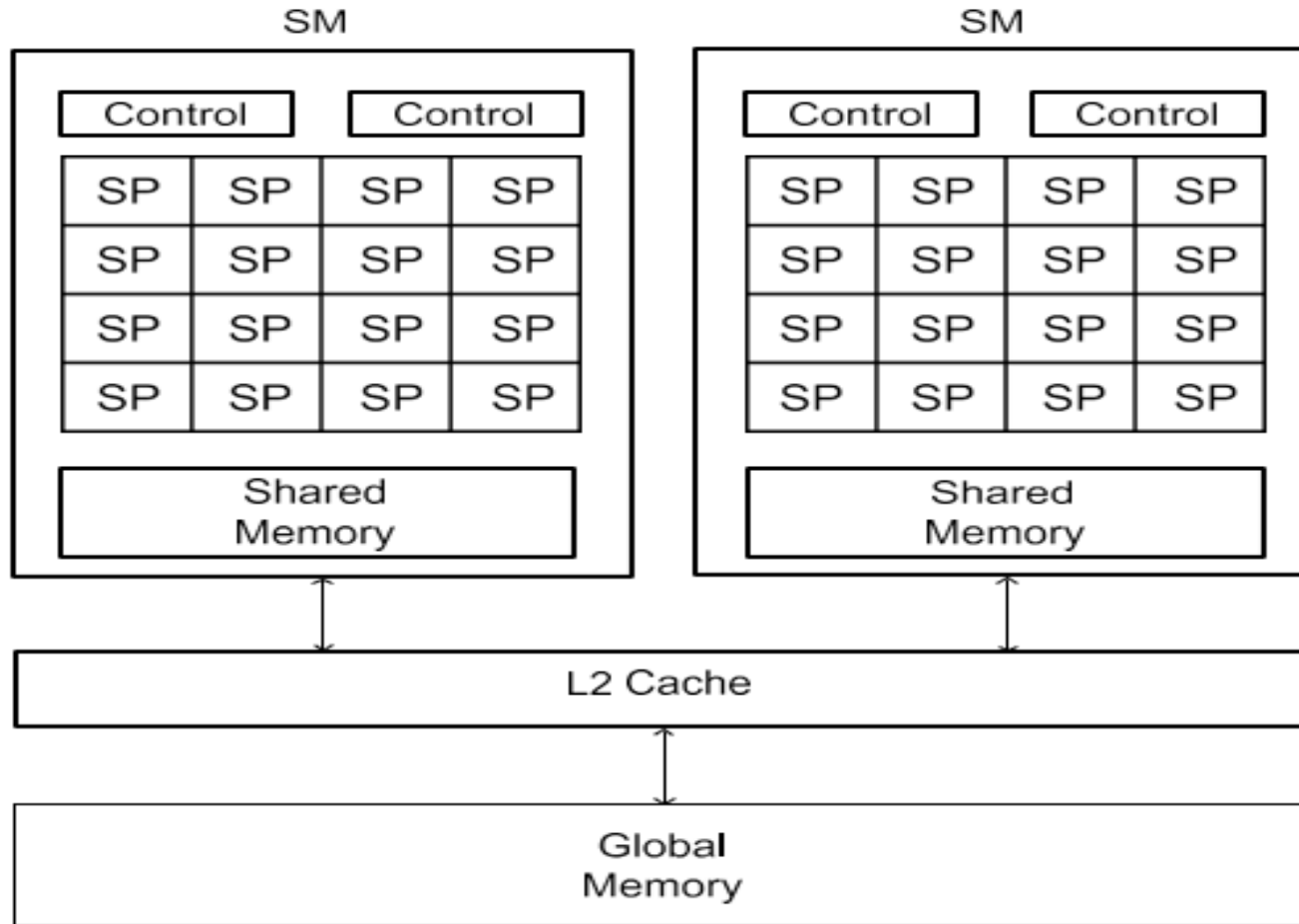


FIGURE 6.1

Simplified block diagram of a GPU.

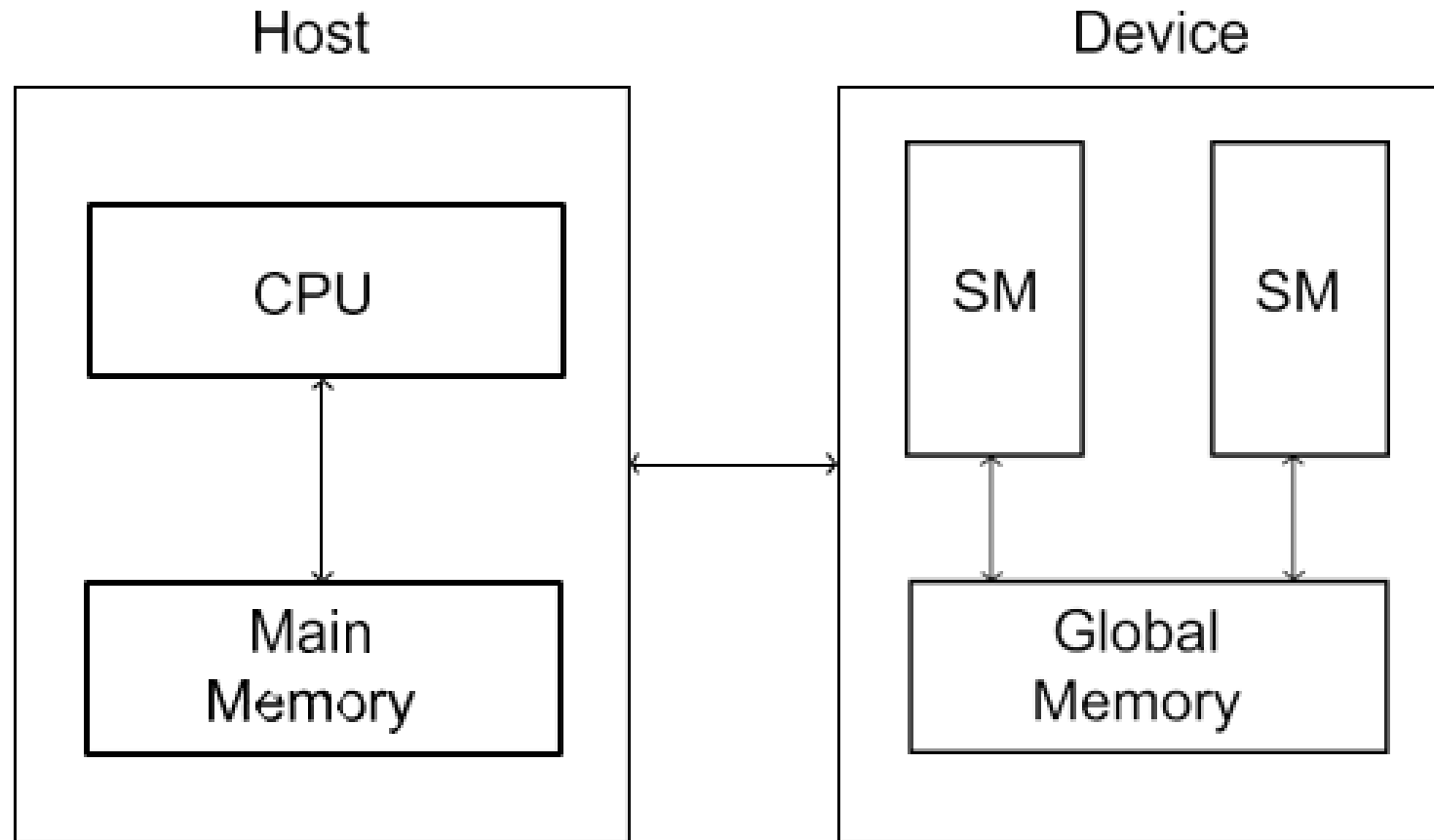


FIGURE 6.2

Simplified block diagram of a CPU and a GPU.

Heterogeneous computing

Writing a program that runs on a GPU is an example of **heterogeneous** computing.

Field Programmable Gate Arrays or FPGAs, and Digital Signal Processors or DSPs. FPGAs contain programmable logic blocks and interconnects that can be configured prior to program execution.

DSPs contain special circuitry for manipulating (e.g., compressing, filter- ing) signals, especially “real-world” analog signals.

CUDA hello

CUDA is a software platform that can be used to write GPGPU programs for heterogeneous systems equipped with an Nvidia GPU.

```
float *x = malloc (n * sizeof ( float ));
```

However, in C++ a cast is required

```
float *x = (float *) malloc (n * sizeof ( float ));
```

```
1 #include <stdio.h>
2 #include <cuda.h>    /* Header file for CUDA */
3
4 /* Device code: runs on GPU */
5 __global__ void Hello(void) {
6
7     printf("Hello from thread %d!\n", threadIdx.x);
8 } /* Hello */
9
10
11 /* Host code: Runs on CPU */
12 int main(int argc, char* argv[]) {
13     int thread_count;    /* Number of threads to run on GPU */
14
15     thread_count = strtol(argv[1], NULL, 10);
16
17     /* Get thread_count from command line */
18
19     Hello <<<1, thread_count>>>();
20
21     /* Start thread_count threads on GPU, */
22
23     cudaDeviceSynchronize();    /* Wait for GPU to finish */
24
25     return 0;
26 } /* main */
```

Program 6.1: CUDA program that prints greetings from the threads.

```
$ nvcc -o cuda_hello cuda_hello.cu
```

If we want to run one thread on the GPU, we can type

```
$ ./cuda_hello 1
```

and the output will be

```
Hello from thread 0!
```

If we want to run ten threads on the GPU, we can type

```
$ ./cuda_hello 10
```

and the output of will be

```
Hello from thread 0! Hello  
from thread 1! Hello from  
thread 2! Hello from thread  
3! Hello from thread 4! Hello  
from thread 5! Hello from  
thread 6! Hello from thread  
7! Hello from thread 8! Hello  
from thread 9!
```

Compiling and running the program

Threads, blocks, and grids

Hello:

Hello <<<1, thread_count > > >());

Hello <<<2, thread_count / 2 > > >());

If thread_count is even, this kernel call will start a total of thread_count threads, and the threads will be divided between the two SMs: thread_count/2 threads will run on each SM.
(What happens if thread_count is odd?)

```
1  #include <stdio.h>
2  #include <cuda.h>    /* Header file for CUDA */
3
4  /* Device code: runs on GPU */
5  __global__ void Hello(void) {
6
7      printf("Hello from thread %d in block %d\n",
8             threadIdx.x, blockIdx.x);
9  } /* Hello */
10
11
12 /* Host code: Runs on CPU */
13 int main(int argc, char* argv[]) {
14     int blk_ct;                /* Number of thread blocks */
15     int th_per_blk;           /* Number of threads in each block */
16
17     blk_ct = strtol(argv[1], NULL, 10);
18     /* Get number of blocks from command line */
19     th_per_blk = strtol(argv[2], NULL, 10);
20     /* Get number of threads per block from command line */
21
22     Hello <<<blk_ct, th_per_blk>>>();
23     /* Start blk_ct*th_per_blk threads on GPU, */
24
25     cudaDeviceSynchronize();    /* Wait for GPU to finish */
26
27     return 0;
28 } /* main */
```

Program 6.2: CUDA program that prints greetings from threads in multiple blocks.

Nvidia compute capabilities and device architectures

There are a number of versions of the CUDA API, and they do *not* correspond to the compute capabilities of the different GPUs.

Name	Ampere	Tesla	Fermi	Kepler	Maxwell	Pascal	Volta	Turing
Compute capability	8.0	1.b	2.b	3.b	5.b	6.b	7.0	7.5

GPU architectures and compute capabilities.

Vector addition

We'll initialize x and y on the host. Then a kernel can start at least n threads, and the i th thread will add

$$z[i] = x[i] + y[i];$$

```
float *x , *y , *z ;
```

After allocating and initializing the arrays,
we'll call the kernel, and after the kernel completes execution,
the program checks the result, frees memory, and quits.

```
1 void Serial_vec_add(  
2     const float  x[]    /* in */,  
3     const float  y[]    /* in */,  
4     float        cz[]   /* out */,  
5     const int    n      /* in */) {  
6  
7     for (int i = 0; i < n; i++)  
8         cz[i] = x[i] + y[i];  
9 } /* Serial_vec_add */
```

Program 6.4: Serial vector addition function.

```
1 void Get_args(  
2     const int argc          /* in */,  
3     char* argv[]           /* in */,  
4     int* n_p                /* out */,  
5     int* blk_ct_p           /* out */,  
6     int* th_per_blk_p       /* out */,  
7     char* i_g               /* out */) {  
8     if (argc != 5) {  
9         /* Print an error message and exit */  
10        ...  
11    }  
12  
13    *n_p = strtol(argv[1], NULL, 10);  
14    *blk_ct_p = strtol(argv[2], NULL, 10);  
15    *th_per_blk_p = strtol(argv[3], NULL, 10);  
16    *i_g = argv[4][0];  
17  
18    /* Is n > total thread count = blk_ct*th_per_blk? */  
19    if (*n_p > (*blk_ct_p)*(*th_per_blk_p)) {  
20        /* Print an error message and exit */  
21        ...  
22    }  
23 } /* Get_args */
```

Program 6.5: Get_args function from CUDA program that adds two vectors.

```
1 void Allocate_vectors(  
2     float ** x_p      /* out */,  
3     float ** y_p      /* out */,  
4     float ** z_p      /* out */,  
5     float ** cz_p     /* out */,  
6     int      n        /* in  */) {  
7  
8     /* x, y, and z are used on host and device */  
9     cudaMallocManaged(x_p, n*sizeof(float));  
10    cudaMallocManaged(y_p, n*sizeof(float));  
11    cudaMallocManaged(z_p, n*sizeof(float));  
12  
13    /* cz is only used on host */  
14    *cz_p = (float *) malloc(n*sizeof(float));  
15 } /* Allocate_vectors */
```

Program 6.6: Array allocation function of CUDA program that adds two vectors.

```
1  double Two_norm_diff(  
2      const float  z[]      /* in */,  
3      const float  cz[]     /* in */,  
4      const int     n       /* in */) {  
5      double diff,  sum = 0.0;  
6  
7      for (int i = 0; i < n; i++) {  
8          diff = z[i] - cz[i];  
9          sum += diff*diff;  
10     }  
11     return sqrt(sum);  
12 } /* Two_norm_diff */
```

Program 6.7: C function that finds the distance between two vectors.

```
1 void Free_vectors(  
2     float * x    /* in/out */,  
3     float * y    /* in/out */,  
4     float * z    /* in/out */,  
5     float * cz   /* in/out */) {  
6  
7     /* Allocated with cudaMallocManaged */  
8     cudaFree(x);  
9     cudaFree(y);  
10    cudaFree(z);  
11  
12    /* Allocated with malloc */  
13    free(cz);  
14 } /* Free_vectors */
```

Program 6.8: CUDA function that frees four arrays.

```
1 void Allocate_vectors (  
2     float **  hx_p      /* out */,  
3     float **  hy_p      /* out */,  
4     float **  hz_p      /* out */,  
5     float **  cz_p      /* out */,  
6     float **  dx_p      /* out */,  
7     float **  dy_p      /* out */,  
8     float **  dz_p      /* out */,  
9     int       n          /* in  */) {  
10  
11     /* dx, dy, and dz are used on device */  
12     cudaMalloc(dx_p, n*sizeof(float));  
13     cudaMalloc(dy_p, n*sizeof(float));  
14     cudaMalloc(dz_p, n*sizeof(float));  
15  
16     /* hx, hy, hz, cz are used on host */  
17     *hx_p = (float *) malloc(n*sizeof(float));  
18     *hy_p = (float *) malloc(n*sizeof(float));  
19     *hz_p = (float *) malloc(n*sizeof(float));  
20     *cz_p = (float *) malloc(n*sizeof(float));  
21 } /* Allocate_vectors */
```

Program 6.10: Allocate_vectors function for CUDA vector addition program that doesn't use unified memory.

Returning results from CUDA kernels

```
__global__ void Add(int x, int y, int *sum_p) {
    *sum_p = x + y;
} /* Add */

int main(void) {
    int sum = -5;
    Add <<<1, 1>>> (2, 3, &sum);
    cudaDeviceSynchronize();
    printf("The sum is %d\n", sum);

    return 0;
}
```

```
int main(void) {
    int *sum_p;
    cudaMallocManaged(&sum_p, sizeof(int));
    *sum_p = -5;
    Add <<<1, 1>>> (2, 3, sum_p);
    cudaDeviceSynchronize();
    printf("The sum is %d\n", *sum_p);
    cudaFree(sum_p);

    return 0;
}
```

```
__managed__ int sum;
```

```
__global__ void Add(int x, int y) {  
    sum = x + y;  
} /* Add */
```

```
int main(void) {  
    sum = -5;  
    Add <<<1, 1>>> (2, 3);  
}
```

CUDA trapezoidal rule I

This gives us a total approximation of the area between the graph and the x-axis as

$$\frac{h}{2} [f(x_0) + f(x_1)] + \frac{h}{2} [f(x_1) + f(x_2)] + \cdots + \frac{h}{2} [f(x_{n-1}) + f(x_n)],$$

and we can rewrite this as

$$h \left[\frac{1}{2} (f(a) + f(b)) + (f(x_1) + f(x_2) + \cdots + f(x_{n-1})) \right].$$

We can implement this with the serial function shown in Program 6.11.

```

1  float Serial_trap(
2      const float  a    /* in */,
3      const float  b    /* in */,
4      const int    n    /* in */) {
5      float x, h = (b-a)/n;
6      float trap = 0.5 * (f(a) + f(b));
7
8      for (int i = 1; i <= n-1; i++) {
9          x = a + i*h;
10         trap += f(x);
11     }
12     trap = trap*h;
13
14     return trap;
15 } /* Serial_trap */

```

Program 6.11: A serial function implementing the trapezoidal rule for a single CPU.

Initialization, return value, and final update

To deal with the initialization and the final update (Items 1 and 5), we could try to select a single thread—say, thread 0 in block 0—to carry out the operations:

```
int my_i = blockDim.x * blockIdx.x + threadIdx.x ;  
if ( my_i == 0 ) {  
    h = ( b-a )/ n ;  
  
    trap = 0.5 * ( f ( a ) + f ( b ) );  
}  
...  
if ( my_i == 0 )  
    trap = trap * h ;
```

CUDA trapezoidal rule II: improving performance

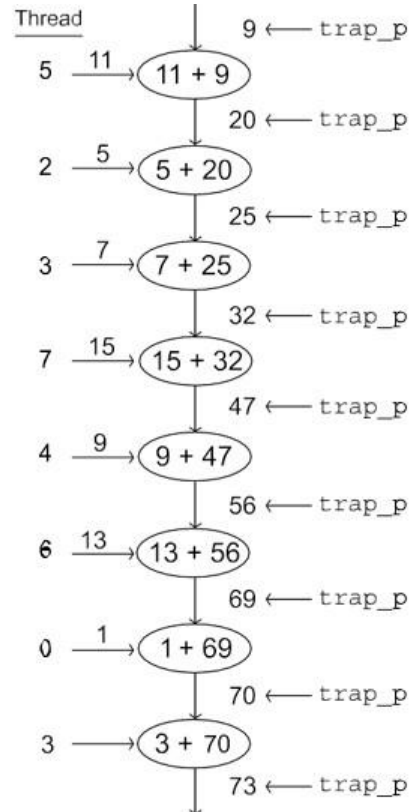


FIGURE 6.3

Basic sum.

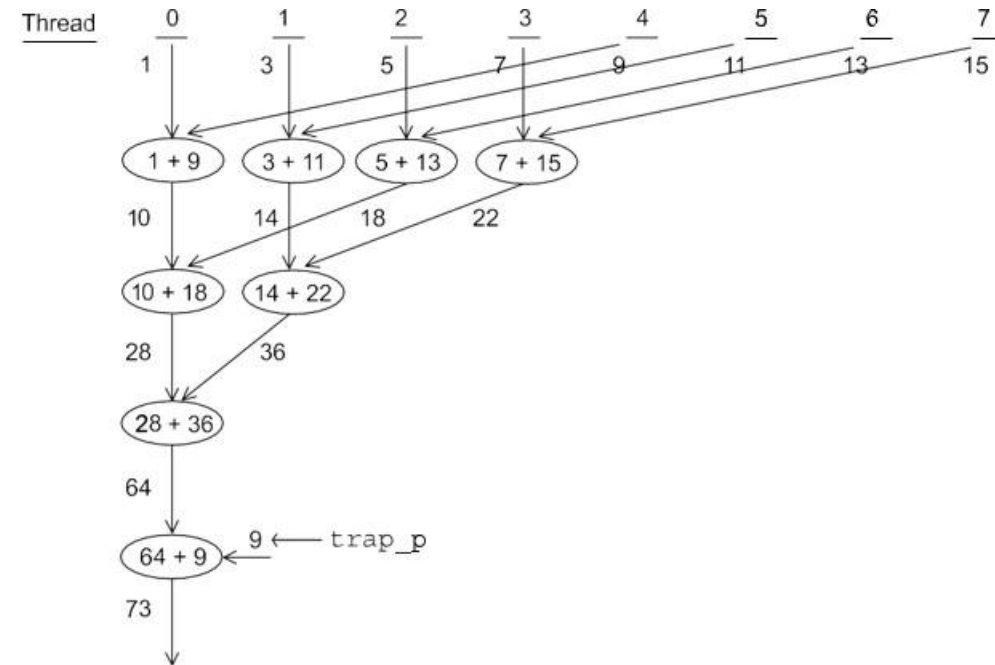


FIGURE 6.4

Tree-structured sum.

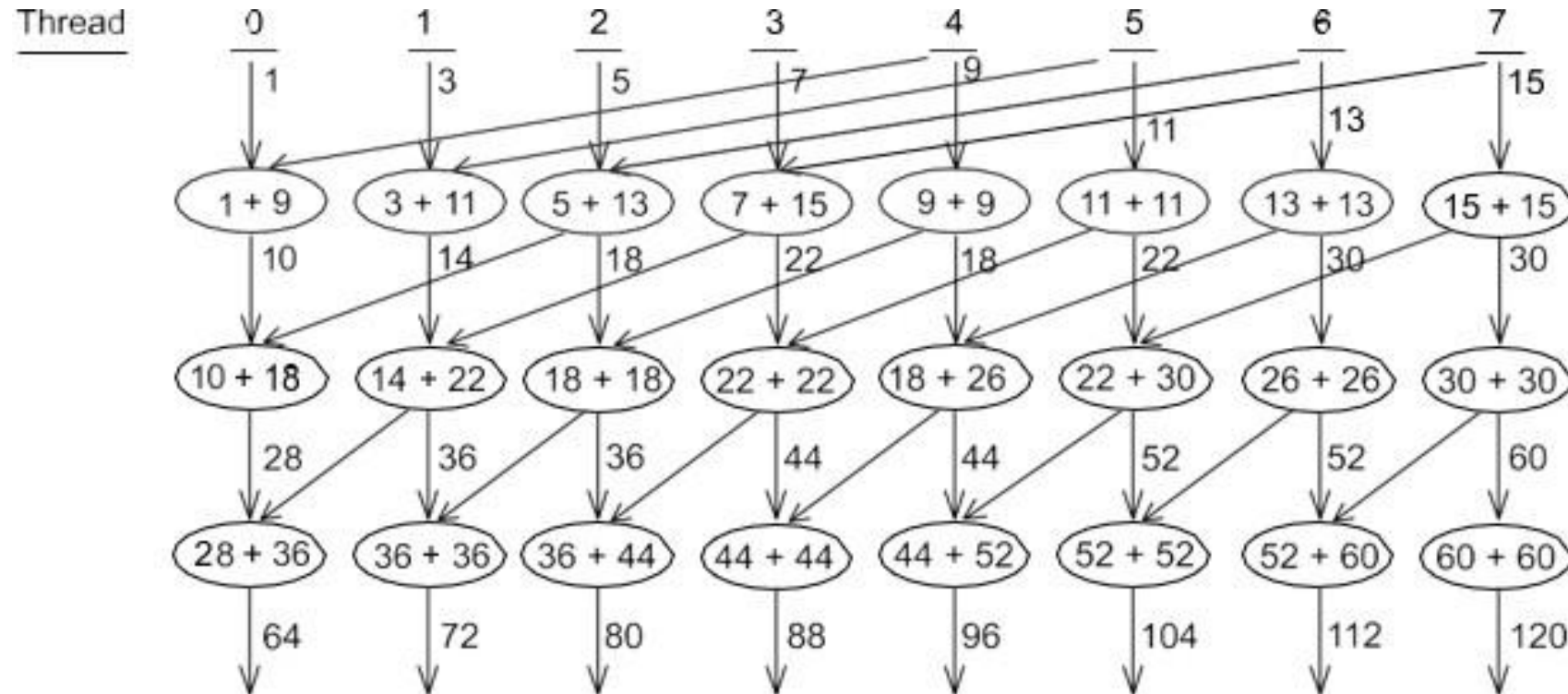


FIGURE 6.5

Tree-structured sum using warp shuffle.

```
__device__ float Shared_mem_sum(float shared_vals[]) {  
    int my_lane = threadIdx.x % warpSize;  
  
    for (int diff = warpSize/2; diff > 0; diff = diff/2) {  
        /* Make sure 0 <= source < warpSize */  
        int source = (my_lane + diff) % warpSize;  
        shared_vals[my_lane] += shared_vals[source];  
    }  
    return shared_vals[my_lane];  
}
```

Thread

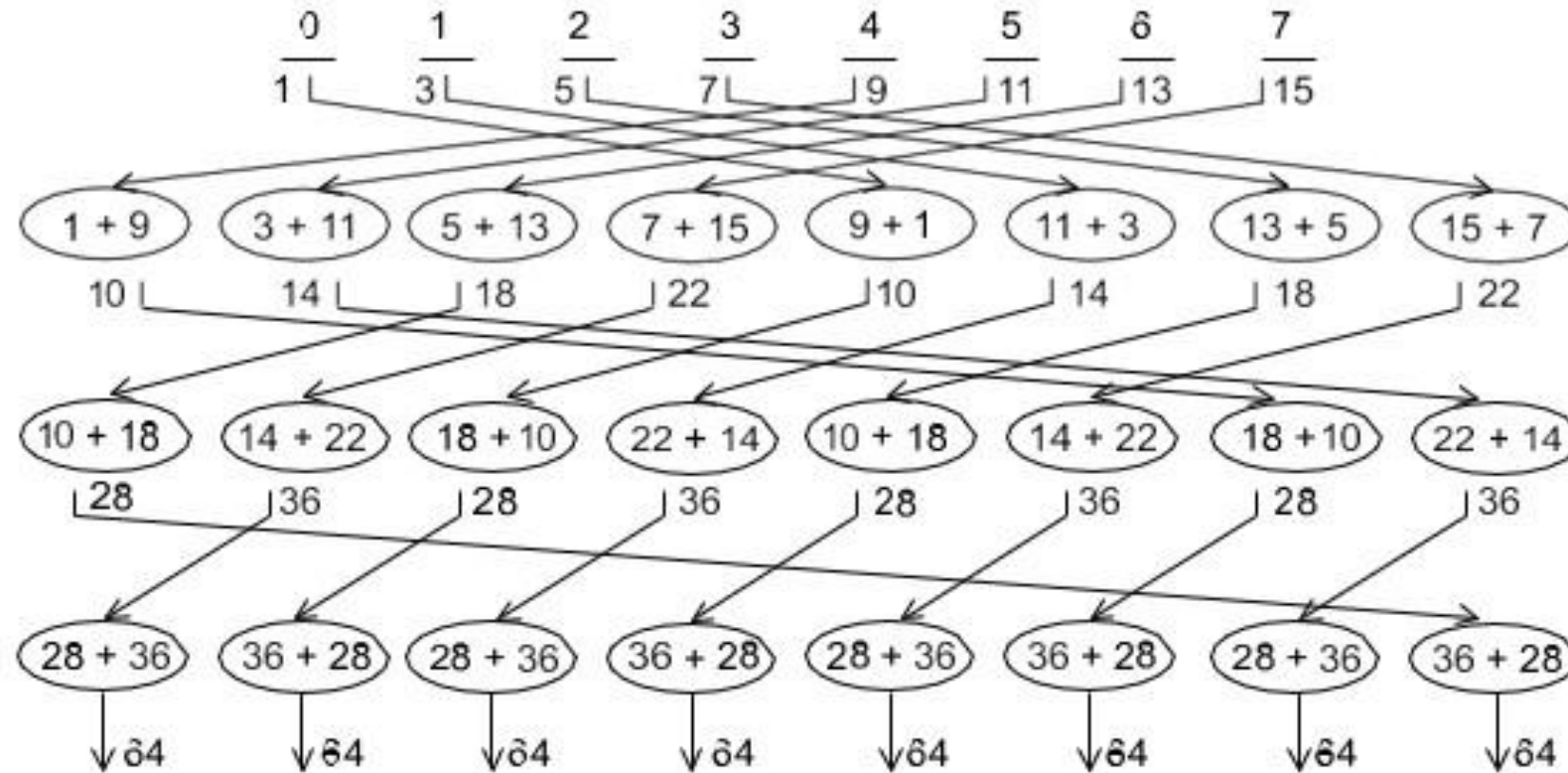


FIGURE 6.6

Dissemination sum using shared memory.

```
1  __global__ void Dev_trap(  
2      const float  a          /* in */,  
3      const float  b          /* in */,  
4      const float  h          /* in */,  
5      const int    n          /* in */,  
6      float *      trap_p     /* in/out */) {  
7      int my_i = blockDim.x * blockIdx.x + threadIdx.x;  
8  
9      float my_trap = 0.0f;  
10     if (0 < my_i && my_i < n) {  
11         float my_x = a + my_i*h;  
12         my_trap = f(my_x);  
13     }  
14  
15     float result = Warp_sum(my_trap);  
16  
17     /* result is correct only on thread 0 */  
18     if (threadIdx.x == 0) atomicAdd(trap_p, result);  
19 } /* Dev_trap */
```

Program 6.13: CUDA kernel implementing trapezoidal rule and using Warp_sum.

```
1  __global__ void Dev_trap(  
2      const float  a          /* in  */,  
3      const float  b          /* in  */,  
4      const float  h          /* in  */,  
5      const int    n          /* in  */,  
6      float *      trap_p     /* out */) {  
7      __shared__ float shared_vals[WARPSZ];  
8      int my_i = blockDim.x * blockIdx.x + threadIdx.x;  
9      int my_lane = threadIdx.x % warpSize;  
10  
11     shared_vals[my_lane] = 0.0f;  
12     if (0 < my_i && my_i < n) {  
13         float my_x = a + my_i*h;  
14         shared_vals[my_lane] = f(my_x);  
15     }  
16  
17     float result = Shared_mem_sum(shared_vals);  
18  
19     /* result is the same on all threads in a block. */  
20     if (threadIdx.x == 0) atomicAdd(trap_p, result);  
21 }
```

Program 6.14: CUDA kernel implementing trapezoidal rule and using shared memory.

Performance

Table 6.8 Mean run-times for trapezoidal rule using block size of 32 threads (times in ms).

System	ARM Cortex-A15	Nvidia GK20A	Intel Core i7	Nvidia GeForce GTX Titan X
Clock	2.3 GHz	852 MHz	3.5 GHz	1.08 GHz
SMs, SPs		1, 192		24, 3072
Original	33.6	20.7	4.48	3.08
Warp Shuffle		14.4		0.210
Shared Memory		15.0		0.206

The remaining codes for the warp sum kernel and the shared memory sum kernel are very similar. First warp 0 computes the sum of the elements in warp_sum_arr. Then thread 0 in the block adds the block sum into the total across all the threads in the grid using atomicAdd. Here's the code for the shared memory sum:

```
if ( my_warp == 0 ) {  
    if ( threadIdx . x >= blockDim . x / warpSize )  
        warp_sum_arr [ threadIdx . x ] = 0.0;  
    blk_result = Shared_mem_sum ( warp_sum_arr );  
}  
if ( threadIdx . x == 0 ) atomicAdd ( trap_p , blk_result );
```



A T M E
College of Engineering



Thank you