

Introduction to parallel programming

Why parallel computing?

- From 1986-2002
 - Performance of microprocessor increased on an average by 50%
 - But later, the performance gain was reduced by 20%
 - Because computer designers started focusing on designing parallel computers
 - Rather than designing complex single core processors
 - Multi-core processors
- But software developers used to develop **serial programs**
- *Aren't single processor systems fast enough?*
- *Why build parallel systems?*
- *Why we need parallel programs?*

Why we need ever-lasting increase in performance?

- Past improvements in performance of microprocessor resulted in quicker web searcher, accurate and quick medical diagnosis, realistic computer games, etc
- Higher computation power means we can solve larger problems:
 - Climate modelling
 - Protein folding
 - Drug discovery
 - Energy research
 - Data analysis

Why we are building parallel systems?

- Increase in single processor performance has been due the ever-increasing density of transistors
- As the size of transistors decreases, their speed can be increased
 - Their power consumption also increases
 - Dissipates heats
 - Highly unreliable
- Hence, it was impossible to increase the speed of integrated circuits
- But, increasing transistor density can continue
- Rather than building ever-faster, more complex, monolithic processors
- They started bringing out multiple, relatively simple, complete processors on a single chip

Why we need to write parallel programs?

- Most serial programs are designed to run on single core
- They are unaware of multiple processors
- We can at max run multiple instances of same program on multiple cores
 - This is not what we want. Why?

Why parallelism?

- Transistor to FLOPs
 - It is possible to fabricate devices with very large transistor counts
 - How we use these transistors to achieve increasing rates of computation?
- Memory and Disk speed
 - Overall speed of computation is determined not just by the speed of the processor, but also by the ability of the memory system to feed data to it
 - Bottleneck: Gap between processor speed and memory
- Data communication
 - Data mining: mining of large data distributed over relatively low bandwidth data
 - Without parallelism its not possible to collect the data at a central location

Applications of parallel computing

- Applications in Engineering and Design
 - Optimization problems
 - Internal combustion engines
 - Airfoils designs in aircraft
- Scientific Applications
 - Sequencing of the human genome
 - Weather modeling, mineral prospecting, flood prediction, etc.
- Commercial Application
 - Web and database servers
 - Data mining
 - Analysis for optimizing business and marketing decisions
- Applications in Computer Systems

Introduction

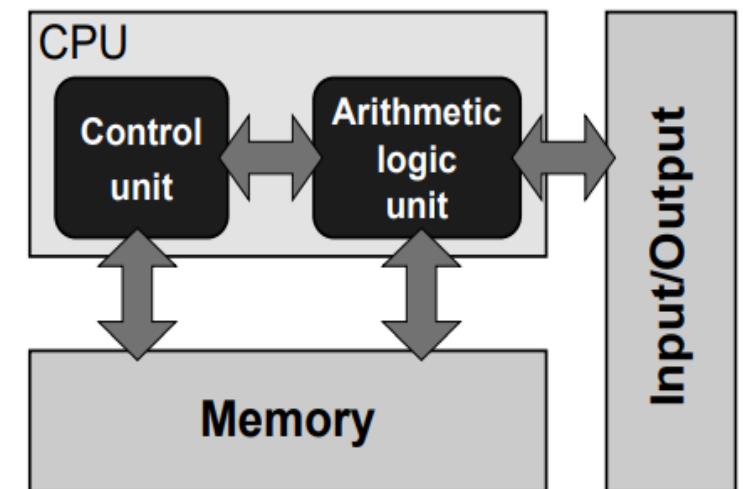
- **Parallel Computing:** It is the use of parallel computer to reduce the time needed to solve computational problem
- **Parallel Computers:** It is a multi-processor computer system that supports parallel programming
 - Two types of parallel computers:
 - **Multi-computer:** Parallel computer constructed out of multiple computers and an inter-connection network
 - **Centralized multi-processors:** An integrated system in which all CPUs share access to a single global memory

Introduction

- **Parallel Programming:** is programming in a language that allows you to explicitly indicate how different portion of the computation may be executed concurrently

Stored-program computer architecture

- Instructions are numbers that are stored as data in memory
- Instructions are read and executed by a control unit
- Arithmetic logic unit is responsible for actual computation. It manipulates the data along with the instructions
- I/O facilities allows for the communication with the users
- Control unit and ALU along with appropriate interfaces to memory and I/O is called as CPU

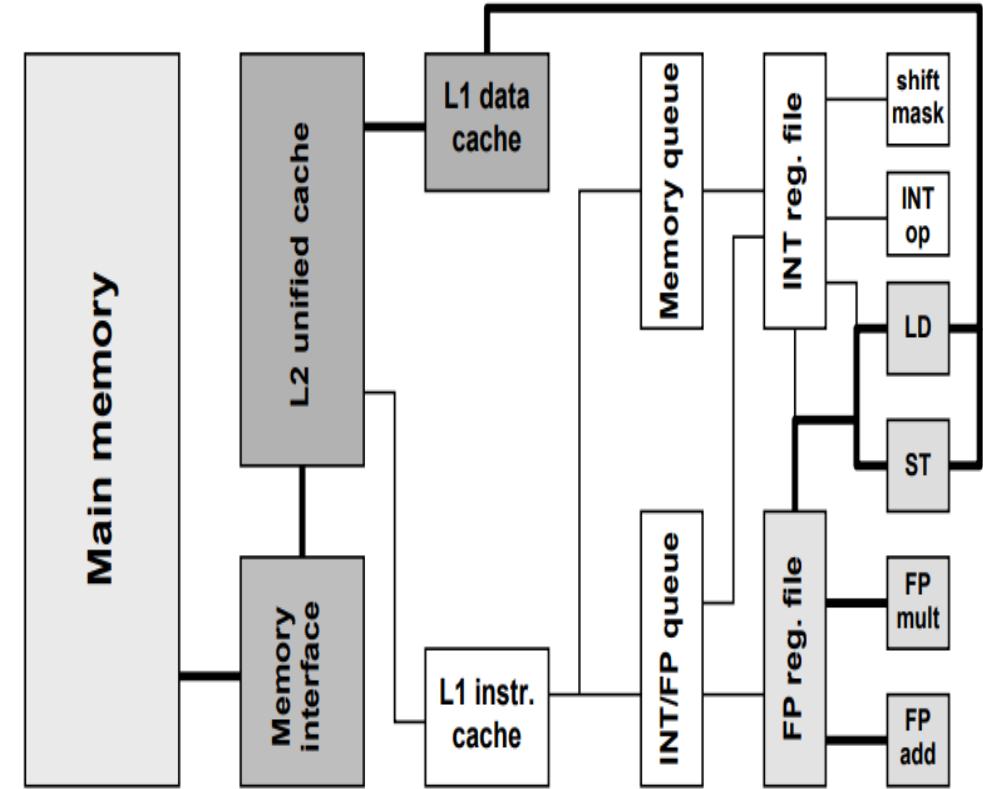


Stored-program computer architecture

- Programming a stored program computer requires us to modify the instructions stored in memory
- This is generally done by another program called as **compiler**
- **This is the general blueprint for all mainstream computers**
- Has few drawbacks:
 - Instructions and data must be continuously fed to the control and arithmetic units: This is known as von Neumann bottleneck
 - The architecture is sequential, processing a single instruction with (possibly) a single operand or a group of operands from memory

General purpose cache-based microprocessor architecture

- Arithmetic units are responsible for running the applications
 - FP (Floating point) and INT (Integer)
- CPU registers hold operands to be accessed by instructions
 - INT reg. file and FP reg. file
 - 16-128 such registers are generally available
- LD and ST units handle instructions that transfer data to and from registers
- Instructions are stored in queues to be executed
- Cache holds the data for re-use



Performance metrics

- Let Π be an arbitrary computational problem which is to be solved by a computer
 - Sequential algorithm performs one operation in each step
 - **Parallel algorithm** may perform multiple operations in a single step
- Let P be a parallel algorithm that has parallelism
- Let $C(p)$ be a parallel computer of the kind C which contains p processing units

Performance metrics

- The performance of P depends on both C and p
- We must consider two things:
 - Potential parallelism in P
 - Ability of $C(p)$ to execute, in parallel, multiple operations of P
- So, the performance of the algorithm P on the parallel computer $C(p)$ depends on $C(p)$'s capability to exploit P 's potential parallelism
- The “***performance***” means the time required to execute P on $C(p)$
 - This is called the ***parallel execution time*** (or, ***parallel runtime***) of P on $C(p)$
 - Denoted by T_{par}

Performance metrics

- **Speedup:** How many times is the parallel execution of P in $C(p)$ faster than the sequential execution of P

$$S = \frac{T_{seq}}{T_{par}}$$

- Parallel execution of P on $C(p)$ is S times faster than sequential execution

Performance metrics and enhancement

- **Efficiency:** Average contribution of each of the p processing units of $C(p)$ to the speedup

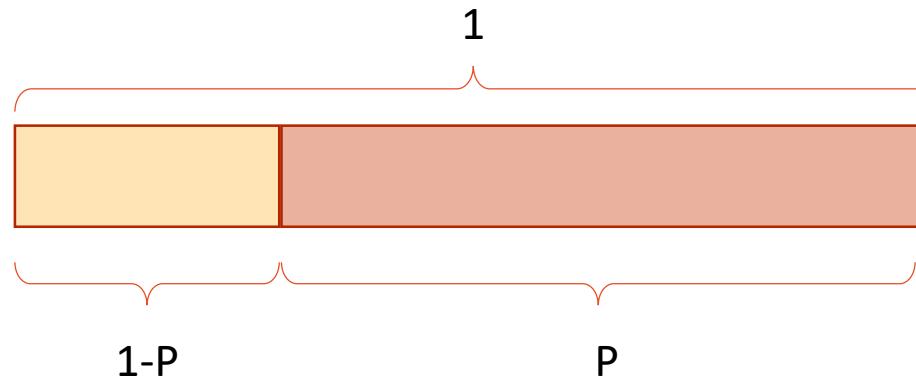
$$E = \frac{S}{p}$$

- Since, $T_{par} \leq T_{seq} \leq p \cdot T_{par}$, Speedup is bounded by p and efficiency is bounded by 1

$$E \leq 1$$

- “For any C and p , the parallel execution of P on $C(p)$ can be at most p times faster than the execution of P on a single processor”

Example:



- Time taken to execute the given parallel program= $T_{par} + T_{seq}$
- Time taken to execute parallel part = P / n
 - Where n =number of processors
- Then the time of running the parallel program will be $1-P+P/n$

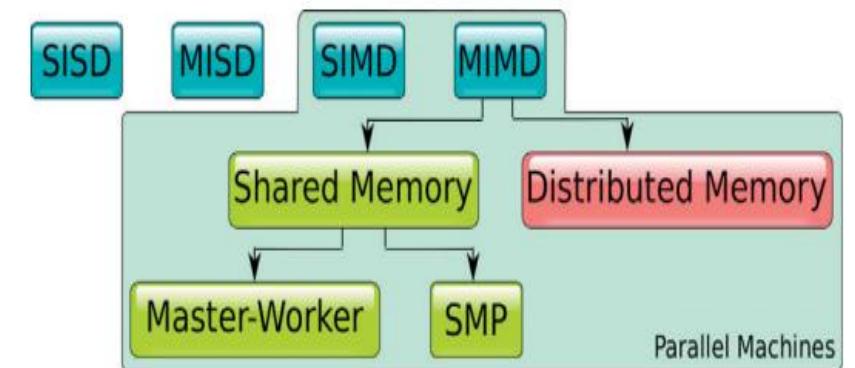
Example:

- Assume, 80% of the program can be parallelized
- Then, 20% cannot be parallelized
- Assume n=4
- Then, time taken to run the parallel program is : $1 - 0.8 + (0.8/4) = 0.4$

- Speedup (S) = T_{seq} / T_{par}
= $1 / 0.4$
= 2.5
- Efficiency = $S/n = 2.5/4$

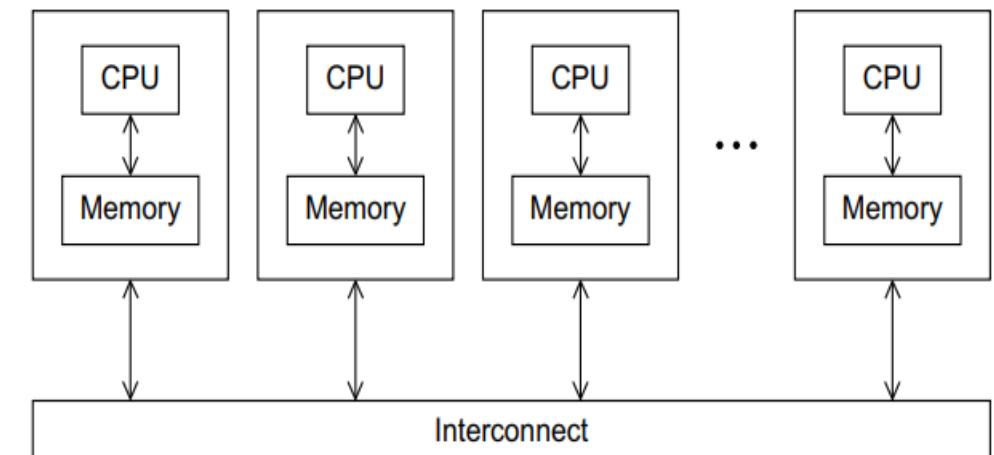
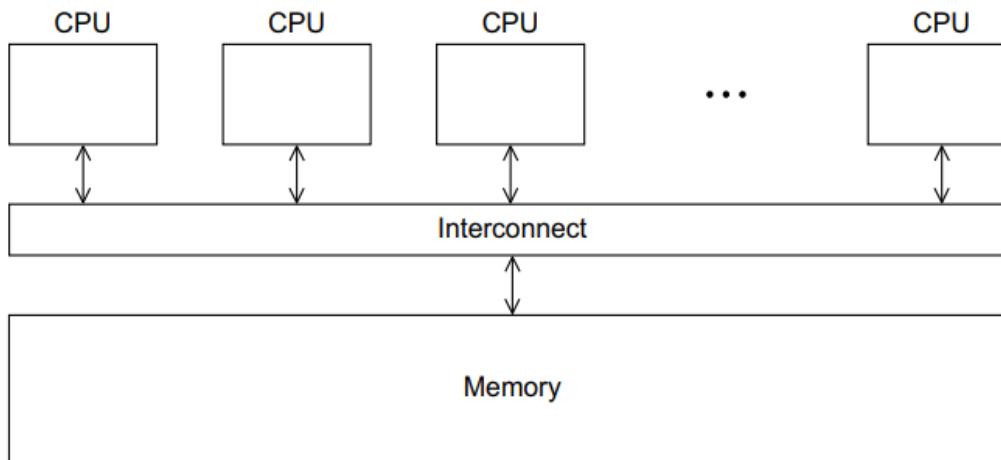
Taxonomy of parallel computers

- In 1966, Michael Flynn proposed a taxonomy of computer architectures
- Based on how many instructions and data items they can process concurrently
 - **SISD:** A sequential machine that can execute one instruction at a time on a single data item (Conventional non-parallel systems)
 - **SIMD:** Single instruction is applied on a collection of items (Ex: GPUs)
 - **MISD:** Multiple instructions are applied on a single data
 - **MIMD:** Multiple instructions are applied on multiple data



Taxonomy of parallel computers

- MIMD can be further divided into two categories:
 - Shared-memory MIMD
 - Distributed-memory MIMD

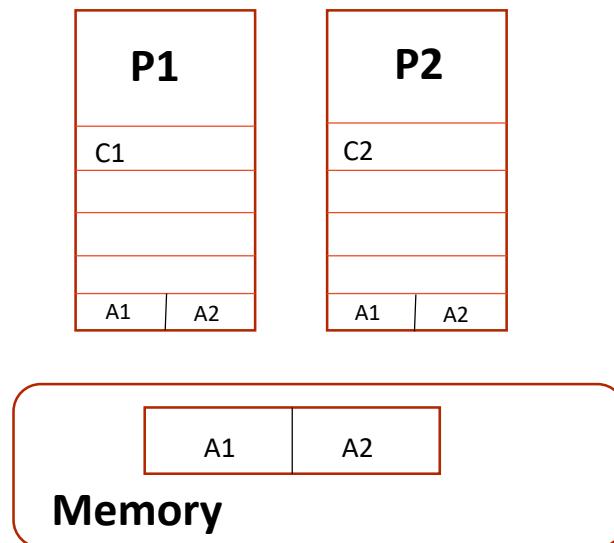


Shared-memory systems

- A number of CPUs work on a common, shared physical address space
- Two varieties of shared-memory systems
 - Uniform Memory Access
 - Latency and bandwidth are same for all processors and memory locations
 - Also called as Symmetric Multi-Processing (SMP)
 - On cache-coherent Nonuniform Memory Access
 - Memory is physically distributed but logically shared
 - The physical layout of such systems are similar to distributed systems
 - The network logic makes the aggregated memory of the whole system appear as one single address space
- In both these cases, copy of same information may reside in different caches, probably in modified state

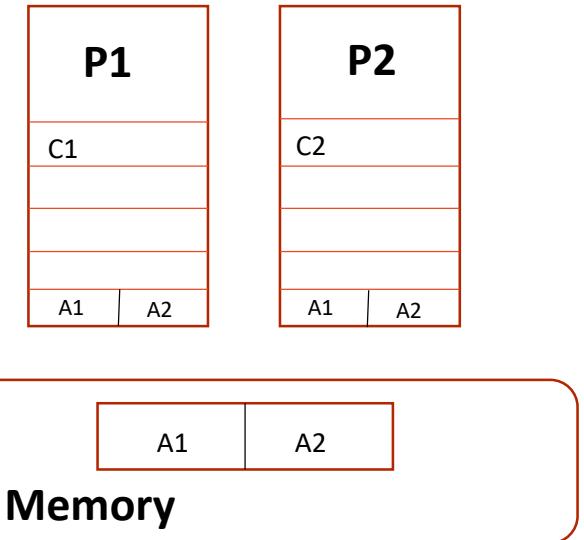
Cache-coherence

- Copies of the same cache line could potentially reside in several CPU caches
- One of those gets modified and evicted to memory
- The other caches' contents reflect outdated data
- **Cache coherence** protocols ensure a consistent view of memory under all circumstances



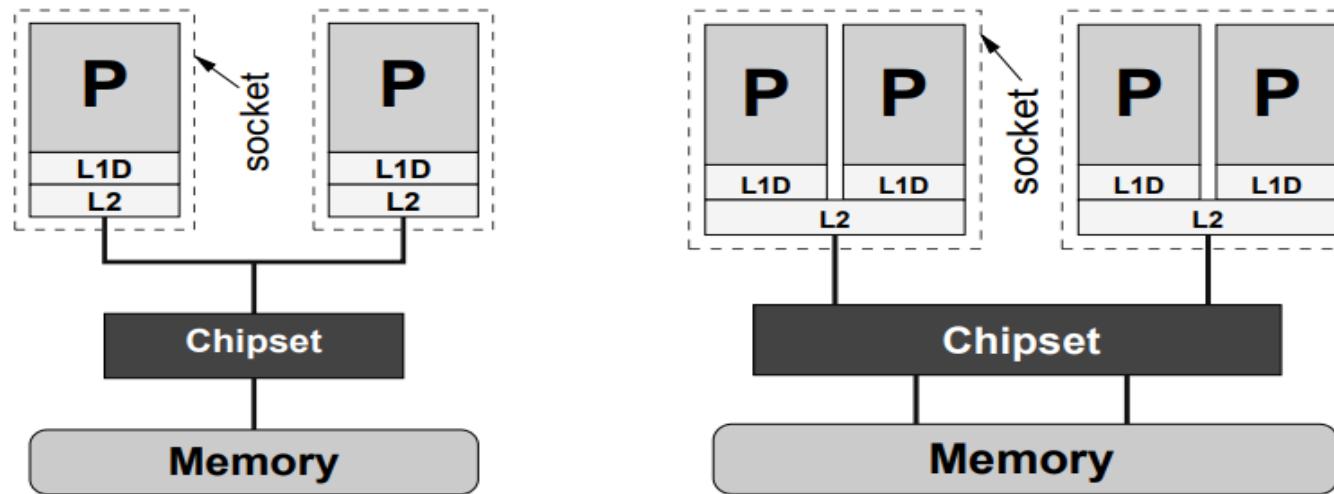
Cache-coherence (MESI protocol)

- Under control of cache coherence logic discrepancy can be avoided
- **M modified:** The cache line has been modified in this cache, and it resides in no other cache than this one. Only upon eviction, memory reflect the most current state.
- **E exclusive:** The cache line has been read from memory but not (yet) modified. However, it resides in no other cache.
- **S shared:** The cache line has been read from memory but not (yet) modified. There may be other copies in other caches of the machine.
- **I invalid:** The cache line does not reflect any sensible data. Under normal circumstances this happens if the cache line was in the shared state and another processor has requested exclusive ownership.



Uniform Memory Access (UMA)

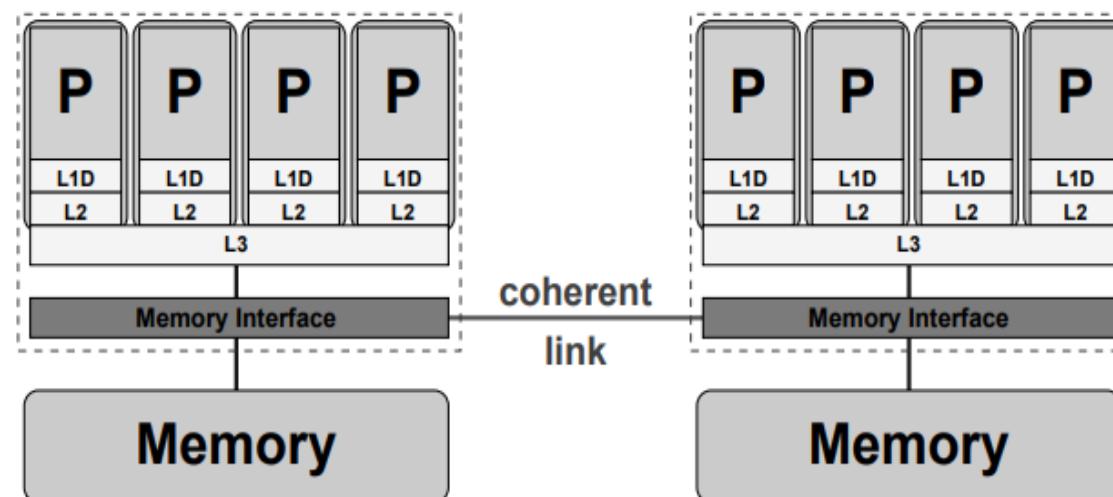
- Simplest implementation of a UMA system is a dual-core processor, in which two CPUs on one chip share a single path to memory



- Problem of UMA systems is that bandwidth bottlenecks are bound to occur

ccNUMA

- Locality domain (LD) is a set of processor cores together with locally connected memory
- Multiple LDs are linked via a coherent interconnect
 - Provides transparent access from any processor to any other processor's memory

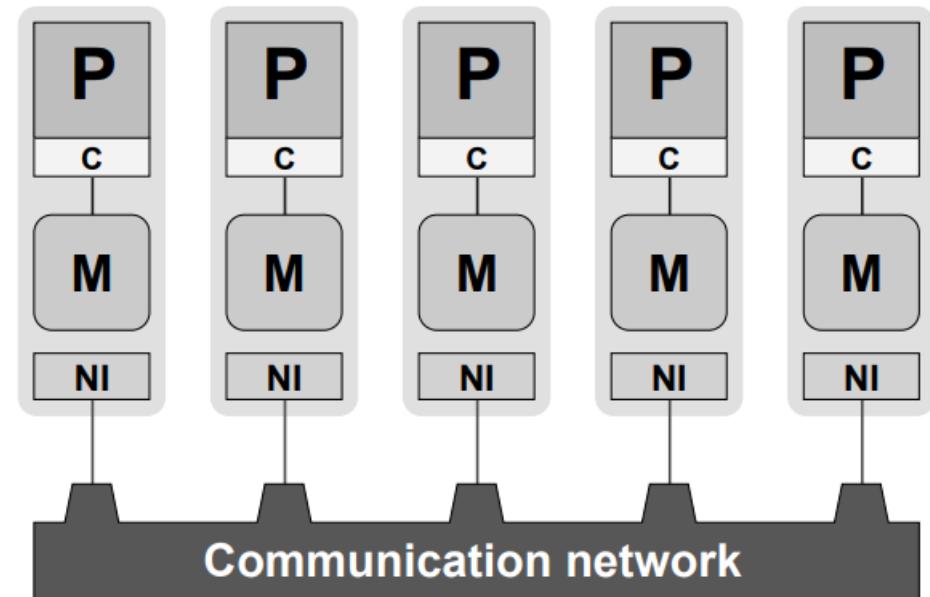


Shared-memory systems

- Advantages:
 - Global address space provides a user-friendly programming perspective to memory
 - Fast and uniform data sharing due to proximity of memory to CPUs
- Disadvantages
 - Lack of scalability between memory and CPUs. Adding more CPUs increases traffic on the shared memory-CPU path
 - Programmers responsibility for correct access to global memory

Distributed-memory systems

- Each processor P is connected to exclusive local memory
- No other CPU has direct access to it
- Each node comprises at least one network interface
- A serial process runs on each CPU that can communicate with other processes on other CPUs by means of the network

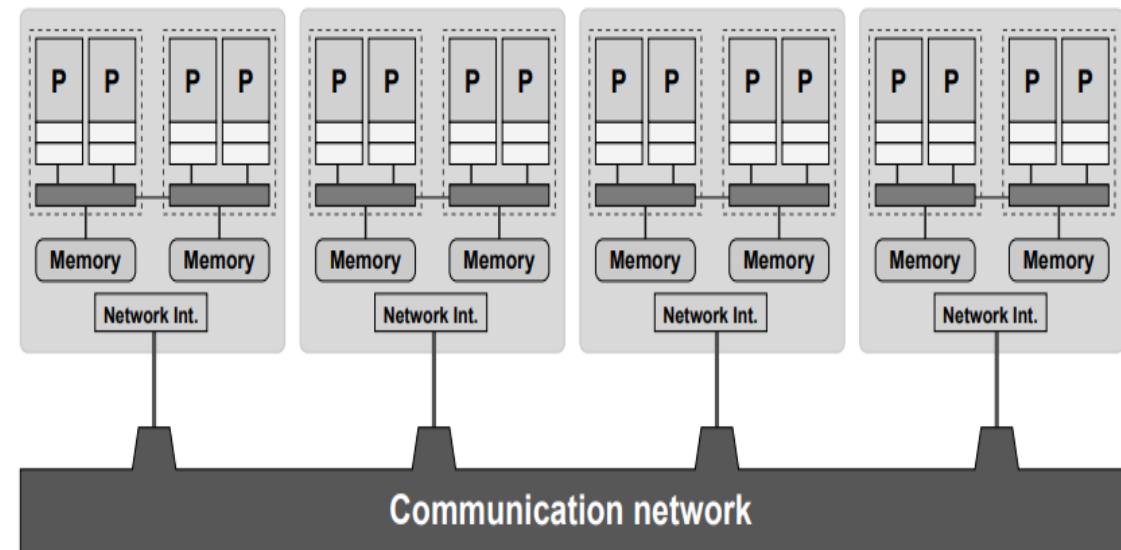


Distributed-memory systems

- Advantages:
 - Memory is scalable with number of CPUs
 - Each CPU can rapidly access its own memory without overhead incurred with trying to maintain global cache coherence
- Disadvantages
 - Programmer is responsible for many of the details associated with data communication between processors
 - It is usually difficult to map existing data structures to this memory organization, based on global memory

Hybrid systems

- Large-scale parallel computers are neither of the purely shared-memory nor of the purely distributed-memory
- Shared-memory building blocks connected via a fast network
- Advantages:
 - Increased scalability
- Disadvantages:
 - Increased programming complexity



END

Introduction to Parallel Programming with MPI

Dr Savitha and Dr Girish's PPLecture slides

Difference between shared memory and distributed memory computer architectures.

- The price of communication: the time needed to exchange a certain amount of data between two or more processors is much faster in shared memory computers
- The second difference, is in the number of processors that can cooperate efficiently, is in favor of distributed memory computers.
- Usually, our primary choice when computing complex tasks will be to engage a large number of fastest available processors, but the communication among them poses additional limitations.
- Cooperation among processors implies communication or data exchange among them.
- When the number of processors must be high (e.g., more than eight) to reduce the execution time, the speed of communication becomes a crucial performance factor.

- There is a difference in the speed of data movement between two computing cores within a single multi-core computer, depending on the location of data to be communicated.
- This is because the data can be stored in registers, cache memory, or system memory, which can differ by up to two orders of magnitude if their access times are considered

- The differences in the communication speed get even more pronounced in the interconnected computers, again by orders of magnitude, but this now depends on the technology and topology of the interconnection networks and on the geographical distance of the cooperating computers.
- Complex tasks can be executed efficiently either
 - (i) on a small number of extremely fast computers or
 - (ii) on a large number of potentially slower interconnected computers.

Message Passing Interface (MPI)

- Enables **system independent** parallel programming.
- The MPI standard includes process creation and management, language bindings for C, point-to-point and collective communications, group and communicator concepts.
- Programmers have to be aware that the cooperation among processes implies the data exchange.
 - The total execution time is consequently a sum of computation and communication time.

- Algorithms with only local communication between neighboring processors are faster and more scalable than the algorithms with the global communication among all processors.
- Therefore, the programmer's view of a problem that will be parallelized has to incorporate a wide number of aspects
 - e.g., data independency, communication type and frequency, balancing the load among processors, balancing between communication and computation, overlapping communication and computation, synchronous or asynchronous program flow, stopping criteria, and others.

Message Passing Interface (MPI)

- The MPI is not a language
- All MPI “**operations**” are expressed as functions, subroutines, or methods
- The MPI standard defines the syntax and semantics of library operations that support the message passing model, independently of program language or compiler specification.
- An MPI program consists of autonomous processes that are able to execute their own code in the sense of multiple instruction multiple data (MIMD) paradigm.
- An MPI “**process**” can be interpreted in this sense as a program counter that addresses their program instructions in the system memory, which implies that the program codes executed by each process need not to be the same.

- The processes communicate via calls to MPI communication operations, independently of operating system.
- Based on the MPI library specifications, several efficient MPI library implementations have been developed, either in open-source or in a proprietary domain.
- Based on the MPI library specifications, several efficient MPI library implementations have been developed, either in open-source or in a proprietary domain.
- The basic MPI communication is characterized by two fundamental MPI operations
 - `MPI_SEND` and `MPI_RECV` that provide sends and receives of process data
 - They are represented by numerous data types.
 - Besides the data transfer these two operations synchronize the cooperating processes in time instants where communication has to be established
 - e.g., a process cannot proceed if the expected data has not arrived.

Message Passing Interface (MPI)

```
1 #include "stdafx.h"
2 #include <stdio.h>
3 #include "mpi.h"
4
5 int main(int argc, char **argv)
6 //int main(argc, argv)
7 //int argc;
8 //char **argv;
9 {
10     int rank, size;
11     MPI_Init(&argc, &argv);
12     MPI_Comm_size(MPI_COMM_WORLD, &size);
13     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14     printf("Hello world from process %d of %d processes.\n", rank, size);
15     MPI_Finalize();
16     return 0;
17 }
```

Listing 4.1 “Hello world” MPI program MSMPIHello.ccp in C programming syntax.

Message Passing Interface (MPI)

- The program has to be compiled only once to be executed on all active processes. Such a methodology could simplify the development of parallel programs.
- `#include "stdafx.h"` is needed because the MS Visual Studio compiler has been used
- `#include <stdio.h>` is needed because of `printf`, which is used later in the program
- `#include "mpi.h"` provides basic MPI definition of named constants, types, and function prototypes, and must be included in any MPI program.

- The **number of processes** will be determined by parameter **-n** of the MPI execution utility ***mpiexec***, usually provided by the MPI library implementation.
- **MPI_Init** initializes the MPI execution environment and
- **MPI_Finalize** exits the MPI.
- **MPI_Comm_size(MPI_COMM_WORLD, & size)** returns size, which is the number of started processes.
- **MPI_Comm_rank(MPI_COMM_WORLD, & rank)** returns rank, i.e., an ID of each process.
- MPI operations return a status of the execution success; in C routines as the value of the function, which is not considered in the above C program

Message Passing Interface (MPI)

```
1 #include "stdafx.h"
2 #include <stdio.h>
3 #include "mpi.h"
4
5 int main(int argc, char **argv)
6 //int main(argc, argv)
7 //int argc;
8 //char **argv;
9 {
10     int rank, size;
11     MPI_Init(&argc, &argv);
12     MPI_Comm_size(MPI_COMM_WORLD, &size);
13     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14     printf("Hello world from process %d of %d processes.\n", rank, size);
15     MPI_Finalize();
16     return 0;
17 }
```

Listing 4.1 “Hello world” MPI program MSMPIHello.ccp in C programming syntax.

Message Passing Interface (MPI)

- The “Hello World” code is the same for all processes.
- It has to be compiled only once to be executed on all active processes.
- Run the program with:

`$ mpiexec -n 3 MSMPIHello`

from Command prompt of the host process, at the path of directory where **MSMPIHello.exe** is located.

The program should output three “Hello World” messages, each with a process identification data.

- All non-MPI procedures are local, e.g., `printf` in the above example.
- It runs on each process and prints separate “Hello World” notice.
- If one would prefer to have only a notice from a specific process, e.g., 0, an extra `if(rank == 0)` statement should be inserted.
- ***Note also that in this simple example no communication between processes has been required.***

- Depending on the number of processes, the printf function will run on each process, which will print a separate “Hello World” notice.
- If all processes will print the output, we expect size lines with “HelloWorld” notice, one from each process.
- Note that the order of the printed notices is not known in advance, because there is no guaranty about the ordering of the MPI processes.

MPI Operation Syntax

- The MPI standard is independent of specific programming languages.
- Capitalized MPI operation names will be used in the definition of MPI operations.

MPI operation arguments, in a language-independent notation, are marked as:

- IN—for input values that may be used by the operation, but not updated;
 - OUT—for output values that may be updated by the operation, but not used as input value;
 - INOUT—for arguments that may be used and/or updated by the MPI operation.
-
- IN arguments are in normal text, e.g., buf, sendbuf, MPI_COMM_WORLD, etc.
 - OUT arguments are in underlined text, e.g., rank, recbuf, etc.
 - INOUT arguments are in underlined italic text, e.g., *inbuf*, request, etc.

Some terms and conventions that are implemented with C program language binding:

- Function names are equal to the MPI definitions but with the MPI_ prefix and the first letter of the function name in uppercase, e.g., **MPI_Finalize()**.
- The status of execution success of MPI operations is returned as integer return codes, e.g., **ierr = MPI_Finalize()**.
 - The return code can be an error code or **MPI_SUCCESS** for successful competition, defined in the file mpi.h.
 - Note that all predefined constants and types are fully capitalized.
- Operation arguments IN are usually passed by value with an exception of the send buffer, which is determined by its initial address. All OUT and INOUT arguments are passed by reference (as pointers)
 - e.g., **MPI_Comm_size (MPI_COMM_WORLD, & size)**.

MPI Data Types

- MPI standard defines its own basic data types that can be used for the specification of message data values, and correspond to the basic data types of the host language.
- As MPI does not require that communicating processes use the same representation of data, i.e., it needs to keep track of possible data types through the build-in basic MPI data types
- For more specific applications, MPI offers operations to construct custom data types, e.g., array of (int, float) pairs, and many other options

- A value of type MPI_BYTE consists of a byte, i.e., 8 binary digits.
- A byte is uninterpreted and is different from a character.
- Different machines may have different representations for characters or may use more than one byte to represent characters.
- On the other hand, a byte has the same binary value on all machines. If the size and representation of data are known, the fastest way is the transmission of raw data, for example, by using an elementary MPI data type MPI_BYTE.

MPI data type	C data type	MPI data type
MPI_INT	int	MPI_INTEGER
MPI_SHORT	short int	MPI_REAL
MPI_LONG	long int	MPI_DOUBLE_PRECISION
MPI_FLOAT	float	MPI_COMPLEX
MPI_DOUBLE	double	MPI_LOGICAL
MPI_CHAR	char	MPI_CHARACTER
MPI_BYTE	/	MPI_BYTE
MPI_PACKED	/	MPI_PACKED

- The MPI communication operations have involved only buffers containing a continuous sequence of identical basic data types.
- Often, one wants to pass messages that contain values with different data types,
 - e.g., a number of integers followed by a sequence of real numbers;
 - or one wants to send noncontiguous data, e.g., a subblock of a matrix.
- The type **MPI_PACKED** is maintained by **MPI_PACK** or **MPI_UNPACK** operations, which enable to pack different types of data into a contiguous send buffer and to unpack it from a contiguous receive buffer

- A user specifies in advance the layout of data types to be sent or received and the communication library can directly access a noncontinuous data.
 - The simplest noncontiguous data type is the vector type, constructed with **MPI_Type_vector**.
 - For example, a sender process has to communicate the main diagonal of an $N \times N$ array of integers, declared as: **int matrix[N][N]**; which is stored in a row-major layout.
- A continuous derived data type **diagonal** can be constructed:
 - **MPI_Datatype MPI_diagonal**; specifies the main diagonal as a set of integers: **MPI_Type_vector (N, 1, N+1, MPI_INT, & diagonal)**; where their count is N, block length is 1, and stride is N+1.
 - The receiver process receives the data as a contiguous block.

Advantages:

- If all data of an MPI program is specified by MPI types it will support data transfer between processes on computers with different memory organization and different interpretations of elementary data items,
 - e.g., in heterogeneous platforms.
- The parallel programs, designed with MPI data types, can be easily ported even between computers with unknown representations of data.
- Further, the custom application oriented data types can reduce the number of memory-to-memory copies or can be tailored to a dedicated hardware for global communication.

MPI Environment Management Routines:

MPI_Init: Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.

MPI_Init (&argc,&argv);

MPI_INIT (int *argc, char *argv)**

MPI_Finalize: Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program. No other MPI routines may be called after it.

MPI_Finalize ();

MPI_FINALIZE ()

Note: The arguments argc and argv are required in C language binding only, where they are parameters of the main C program.

No MPI routine can be called before MPI_INIT or after MPI_FINALIZE, with one exception MPI_INITIALIZED (flag), which queries if MPI_INIT has been called.

MPI Environment Management Routines:

MPI_Comm_rank: Returns the rank of the calling MPI process within the specified communicator. Each process will be assigned a unique integer rank between 0 and size - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a process ID.

```
MPI_Comm_rank(Comm,&rank);
```

MPI_COMM_RANK (comm, rank)

MPI_Comm_size: Returns the total number of MPI processes to the variable size in the specified communicator, such as MPI_COMM_WORLD. **Returns the number of processes in the current communicator**

```
MPI_Comm_size(Comm,&size);
```

MPI_COMM_SIZE (comm, size)

- The input argument comm is the handle of communicator; the output argument size returned by the operation MPI_COMM_SIZE is the number of processes in the group of comm.
- If comm is MPI_COMM_WORLD, then it represents the number of all active MPI processes.

MPI Error Handling

- The MPI standard assumes a reliable and error-free underlying communication platform; therefore, it does not provide mechanisms for dealing with failures in the communication system.
 - For example, a message sent is always received correctly, and the user need not check for transmission errors, time-outs, or similar.
- MPI does not provide mechanisms for handling processor failures. A program error can follow an MPI operation call with incorrect arguments,
 - e.g., non-existing destination in a send operation, exceeding available system resources, or similar
- Most of MPI operation calls return an error code that indicates the completion status of the operation.
- Before the error value is returned, the current MPI error handler is called, which, by default, aborts all MPI processes.
- One can specify that no MPI error is fatal, and handle the returned error codes by custom error-handling routines.

MPI Environment Management Routines:

Solved Example:

Write a program in MPI to print total number of process and rank of each process.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank,size;

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("My rank is %d in total %d processes", rank, size);
    MPI_Finalize();
    return 0;
}
```

Process-to-Process Communication or Point to point Communication in MPI

Objectives:

1. Understand the different APIs used for point to point communication in MPI
 2. Learn the different modes available in case of blocking send operation
-
- The process-to-process communication has to implement two essential tasks:
 - data movement and
 - Synchronization of processes;Therefore, it requires cooperation of sender and receiver processes.

Process-to-Process Communication or Point to point Communication in MPI

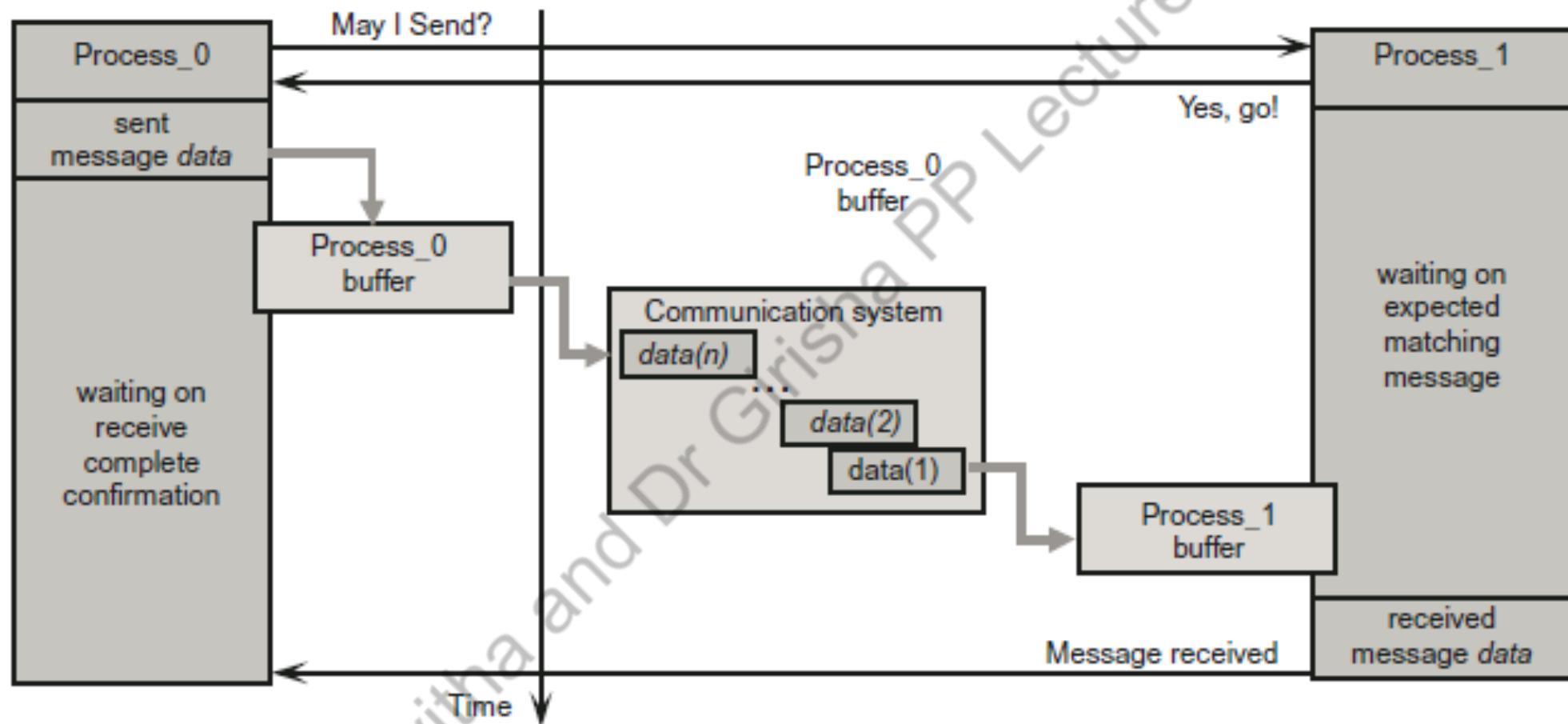


Fig.4.1 Communication between two processes awakes both of them while transferring data from sender Process_0 to receiver Process_1, possibly with a set of shorter sub-messages

Point to Point Communication in MPI

- Optional **intermediate message buffers** are used in order to enable sender Process_0 to continue immediately after it initiates the send operation.
- However, Process_0 will have to wait on the return from the previous call, before it can send a new message.
- On the receiver side, Process_1 can do some useful work instead of idling while waiting on the matching message reception.
- It is a communication system that must ensure that the message will be reliably transferred between both processes.
- If the processes have been created on a single computer, the actual communication will be probably implemented through a shared memory.

- If the processes reside on two distant computers, then the actual communication might be performed through an existing interconnection network using, e.g., TCP/IP communication protocol.
- Although that blocking send/receive operations enable a simple way for **synchronization of processes**, they could introduce **unnecessary delays** in cases where sender and receiver do not reach communication point at the same real time.
- For example, if Process_0 issues a send call significantly before the matching receives call in Process_1, Process_0 will start waiting to the actual message data transfer.
- In the same way, processes' idling can happen if a process that produces many messages is much faster than the consumer process. **Message buffering may alleviate the idling** to some extent, but if the amount of data exceeds the capacity of the message buffer, which can always happen, Process_0 will be blocked again.

Point to Point Communication in MPI

- The next concern of the blocking communication are **deadlocks**.
- For example, if Process_0 and Process_1 initiate their send calls in the same time, they will be blocked forever by waiting matching receive calls.

Point to Point Communication in MPI

Point to Point communication in MPI

- MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.
- MPI provides both blocking and non-blocking send and receive operations.

MPI_SEND (buf, count, datatype, dest, tag, comm)

- The send buffer is specified by the following arguments
 - buf - pointer to the send buffer,
 - count - number of data items,
 - datatype - type of data items.
- The receiver process is addressed by an envelope that consists of arguments
 - dest, which is the rank of receiver process within all processes in the communicator comm, and of a message tag.
 - tag provide a mechanism for distinguishing between different messages for the same receiver process identified by destination rank

Point to Point Communication in MPI

MPI_RECV (buf, count, datatype, source, tag, comm, status)

This operation waits until the communication system delivers a message with matching datatype, source, tag, and comm.

- The entire set of arguments: count, datatype, source, tag and comm, must match between the sender process and the receiver process to initiate actual message passing.
- When a message, posted by a sender process, has been collected by a receiver process, the message is said to be completed, and the program flows of the receiver and the sender processes may continue.

Point to Point Communication in MPI

Sending message in MPI

- **Blocked Send** sends a message to another processor and waits until the receiver has received it before continuing the process. Also called as **Synchronous send**.
- **Send** sends a message and continues without waiting. Also called as **Asynchronous send**.

There are multiple communication modes used in blocking send operation:

- **Standard mode**
- **Synchronous mode**
- **Buffered mode**

Point to Point Communication in MPI

Standard mode

This mode blocks until the message is buffered.

```
MPI_Send(&Msg, Count, Datatype, Destination, Tag, Comm);
```

- First 3 parameters together constitute message buffer. The **Msg** could be any address in sender's address space. The **Count** indicates the number of data elements of a particular type to be sent. The **Datatype** specifies the message type. Some Data types available in MPI are: MPI_INT, MPI_FLOAT, MPI_CHAR, MPI_DOUBLE, MPI_LONG
- Next 3 parameters specify message envelope. The **Destination** specifies the rank of the process to which the message is to be sent.
- **Tag:** The **tag** is an integer used by the programmer to label different types of messages and to restrict message reception.

Point to Point Communication in MPI

- **Communicator:** Major problem with tags is that they are specified by users who can make mistakes. **Context** are allocated at run time by the system in response to user request and are used for matching messages. The notions of **context and group** are combined in a single object called a communicator (**Comm**).
- The default process group is **MPI_COMM_WORLD**.

Point to Point Communication in MPI

Synchronous mode

This mode requires a send to block until the corresponding receive has occurred.

```
MPI_Ssend(&Msg, Count, Datatype, Destination, Tag, Comm);
```

Buffered mode

```
MPI_Bsend(&Msg, Count, Datatype, Destination, Tag, Comm);
```

In this mode a send assumes availability of a certain amount of buffer space, which must be previously specified by the user program through a routine call that allocates a user buffer.

Point to Point Communication in MPI

```
MPI-Buffer_attach(buffer, size);
```

This buffer can be released by

```
MPI_Buffer_detach(*buffer, *size);
```

Point to Point Communication in MPI

Receiving message in MPI

```
MPI_Recv(&Msg, Count, Datatype, Source, Tag, Comm, &status);
```

- Receive a message and block until the requested data is available in the application buffer in the receiving task.
- The **Msg** could be any address in receiver's address space. The **Count** specifies number of data items. The **Datatype** specifies the message type. The **Source** specifies the rank of the process which has sent the message. The **Tag** and **Comm** should be same as that is used in corresponding send operation. The status is a structure of type status which contains following information: Sender's rank, Sender's tag and number of items received

Point to Point Communication in MPI

Finding execution time in MPI

MPI_Wtime: Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

- **MPI_Wtime ()**

```
double start, finish;
start = MPI_Wtime ();
... //MPI program segment to be clocked
finish = MPI_Wtime ();
printf ("Elapsed time is %f\n", finish - start);
```

Solved Example:

Write a MPI program using standard send. The sender process sends a number to the receiver. The second process receives the number and prints it.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank,size,x;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Status status;
    if(rank==0)
    {
        printf("Enter a value in master process:");
        scanf("%d",&x);
        MPI_Send(&x,1,MPI_INT,1,1,MPI_COMM_WORLD);
        fprintf(stdout,"I have sent %d from process 0\n",x);
        fflush(stdout);
    }
    else
    {
        MPI_Recv(&x,1,MPI_INT,0,1,MPI_COMM_WORLD,&status);
        fprintf(stdout,"I have received %d in process 1\n",x);
        fflush(stdout);
    }
    MPI_Finalize();
    return 0;
}
```

Point to Point Communication in MPI

Seven basic MPI operations

Many parallel programs can be written and evaluated just by using the following seven MPI operations that have been overviewed in the previous sections:

```
MPI_INIT,  
MPI_FINALIZE,  
MPI_COMM_SIZE,  
MPI_COMM_RANK,  
MPI_SEND,  
MPI_RECV,  
MPI_WTIME.
```

Enable MPI in Visual Studio

- Download MPI for Windows(Microsoft MPI)
- <https://www.microsoft.com/en-s/download/details.aspx?id=57467>
- Run both **.exe** and **.msi** file, they will install Microsoft MPI under C:\Program Files\Microsoft MPI by default.(But if you have changed the register manually, the path might be changed)

Configure MPI in Visual Studio 2019

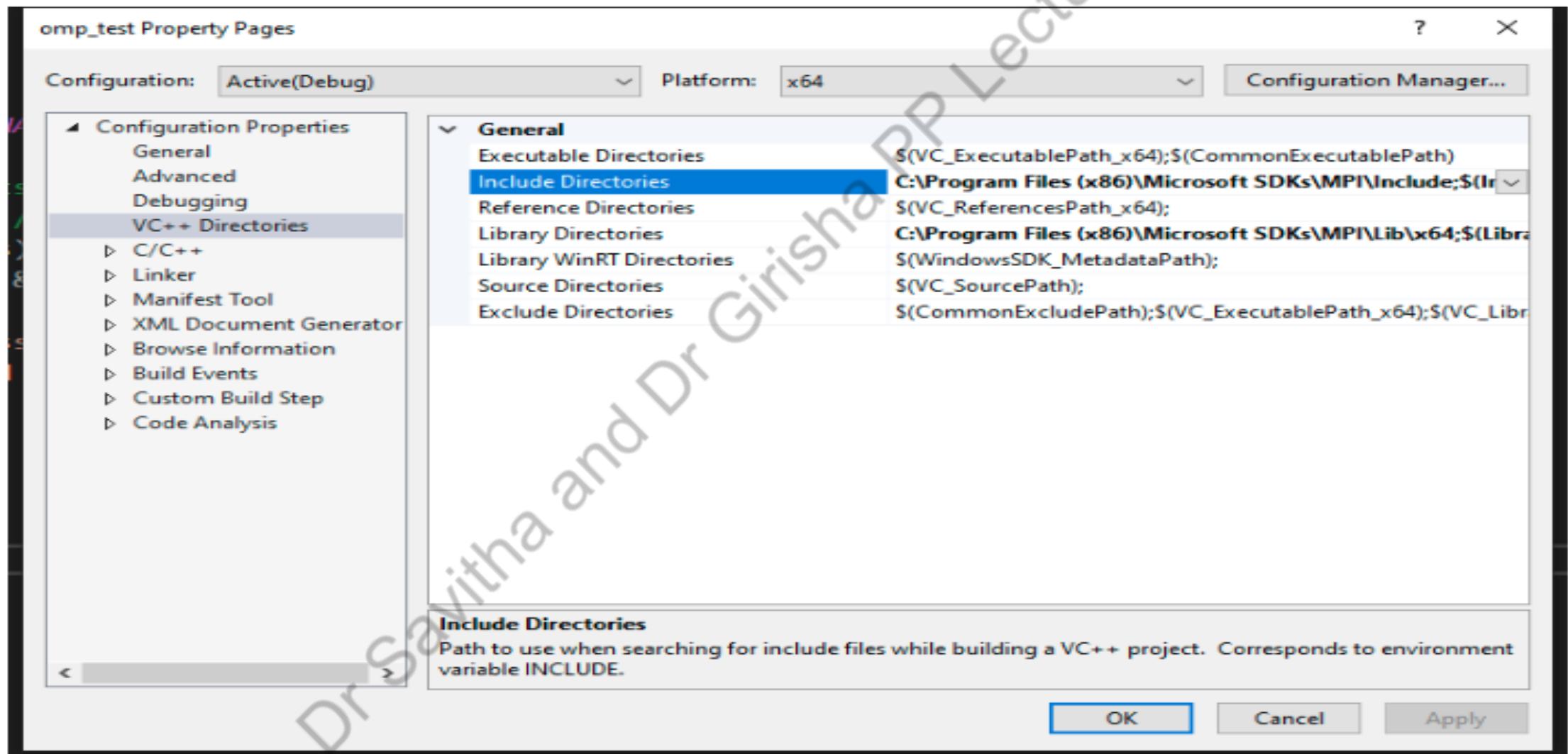
Open Project -> Project_name Properties

Under VC++ Directories

Add C:\Program Files (x86)\Microsoft SDKs\MPI\Include in Include Directories

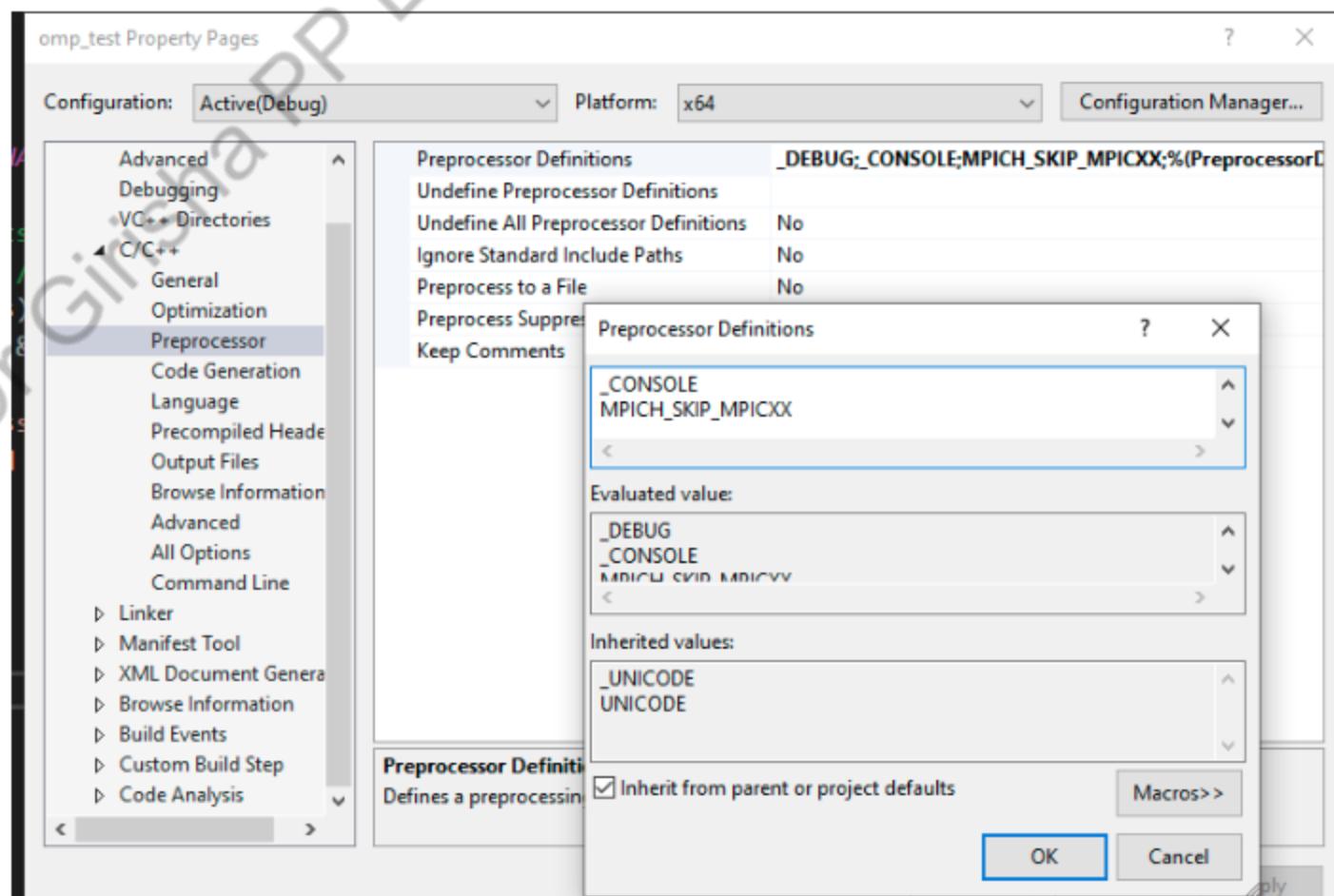
Add C:\Program Files(x86)\Microsoft SDKs\MPI\Lib\x86 in Library Directories

Configure MPI in Visual Studio 2019



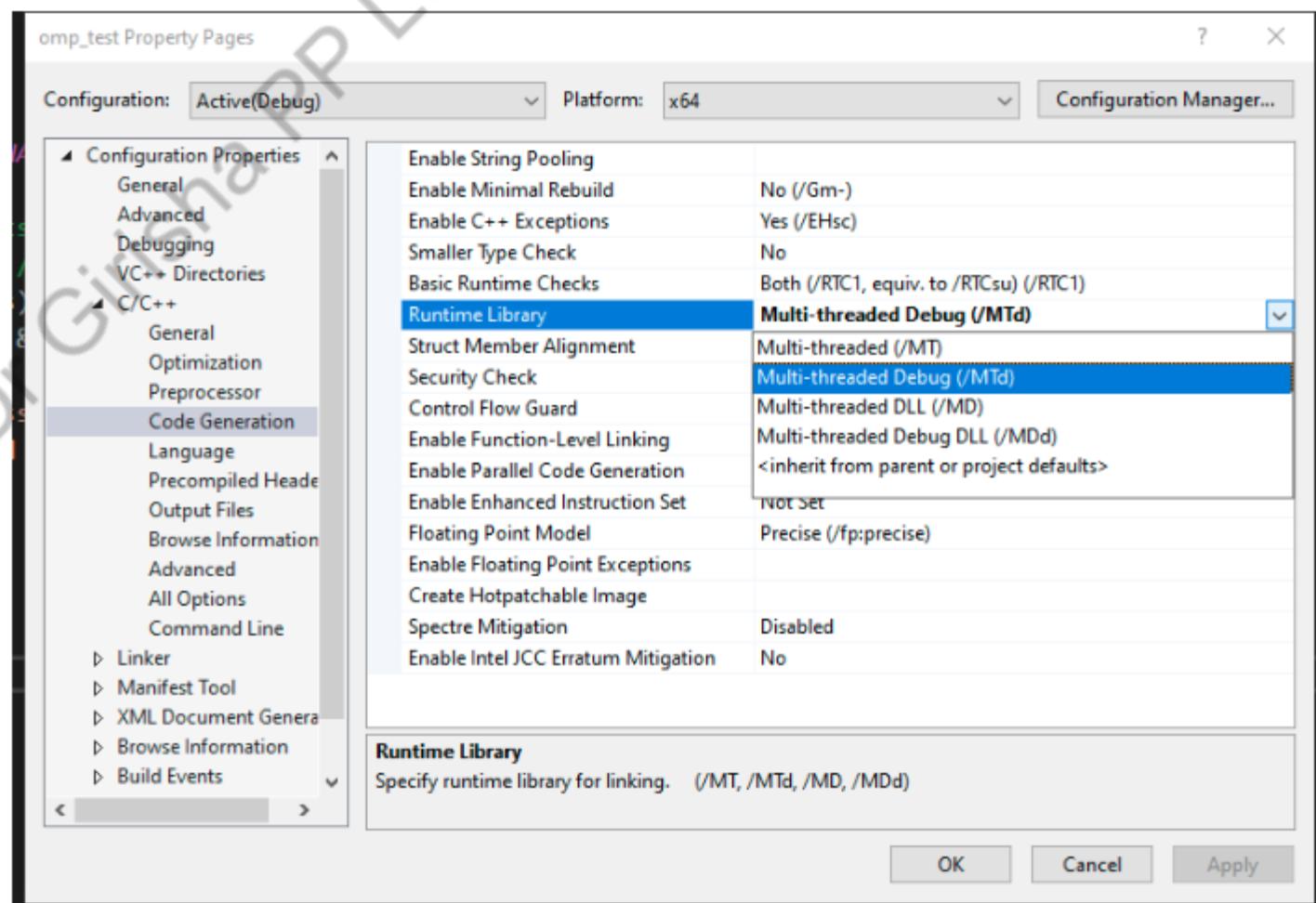
Configure MPI in Visual Studio 2019

- In C/C++ -> Preprocessor -> Preprocessor Definitions
- Add MPICH_SKIP_MPICXX



Configure MPI in Visual Studio 2019

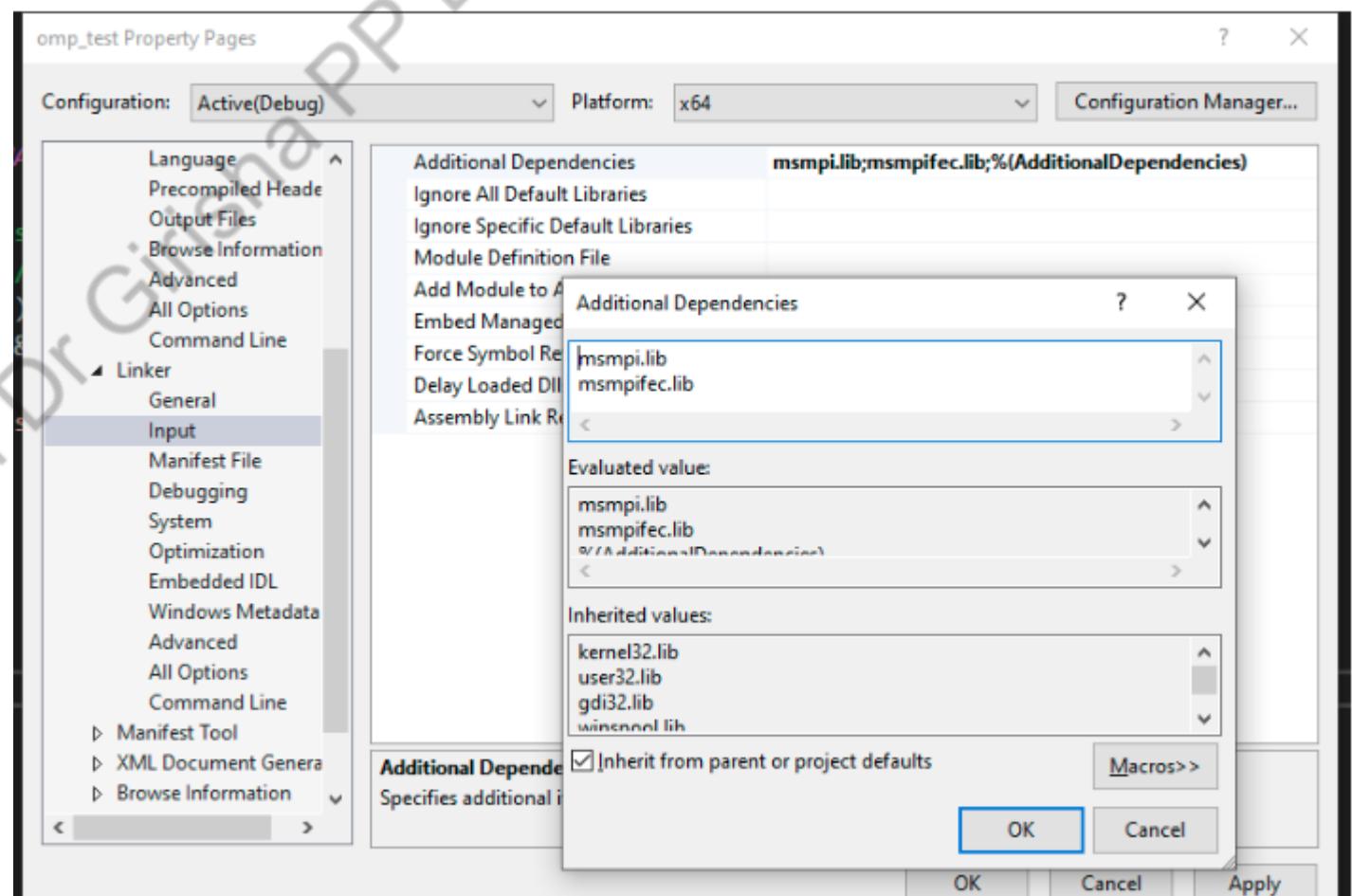
- In C/C++ -> Code Generation
- Change Runtime Library to MTd



Configure MPI in Visual Studio 2019

In Linker -> Input -> Additional Dependencies

Add `msmpi.lib` and `msmpifec.lib`



Testing

```
#include<stdio.h> #include<mpi.h> #include<stdlib.h>
int main(int argc, char* argv[])
{
    int myid, numprocs, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);          // starts MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &myid); // get current process id
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs); // get number of processes
    MPI_Get_processor_name(processor_name, &namelen);

    if (myid == 0) printf("number of processes: %d\n...", numprocs);
    printf("%s: Hello world from process %d \n", processor_name, myid);

    MPI_Finalize();

    return 0;
}
```

Click Build -> Build Solution

And your terminal will look like the following screenshot.

Please make sure you can build .exe file successfully without error.

You can see the .exe file path inside the terminal, for me, it is

E:\TestingPrograms\omp_test\x64\Debug\omp_test.exe

Testing

The screenshot shows the Microsoft Visual Studio IDE interface. In the top-left corner, there's a context menu with options like 'Rebuild', 'Clean', 'Run Code Analysis', 'Project Only', etc. Below the menu is a toolbar with icons for 'Compile' and 'Run Code Analysis'. The main area is a code editor displaying MPI C code. The code includes MPI initialization, communication, and finalization. The output window at the bottom shows a successful build process.

```
Rebuild omp_test
Clean omp_test
Run Code Analysis on omp_test
Project Only
Profile Guided Optimization
Batch Build...
Configuration Manager...

Compile Ctrl+F7
Run Code Analysis on File Ctrl+Shift+Alt+F7

12
13
14
15
16
17
18
19
20
21
22

MPI_Comm_size(MPI_COMM_WORLD, &numprocs); // get number of processes
MPI_Get_processor_name(processor_name, &namelen);
if (myid == 0) printf("number of processes: %d\n...", numprocs);
printf("%s: Hello world from process %d \n", processor_name, myid);
MPI_Finalize();

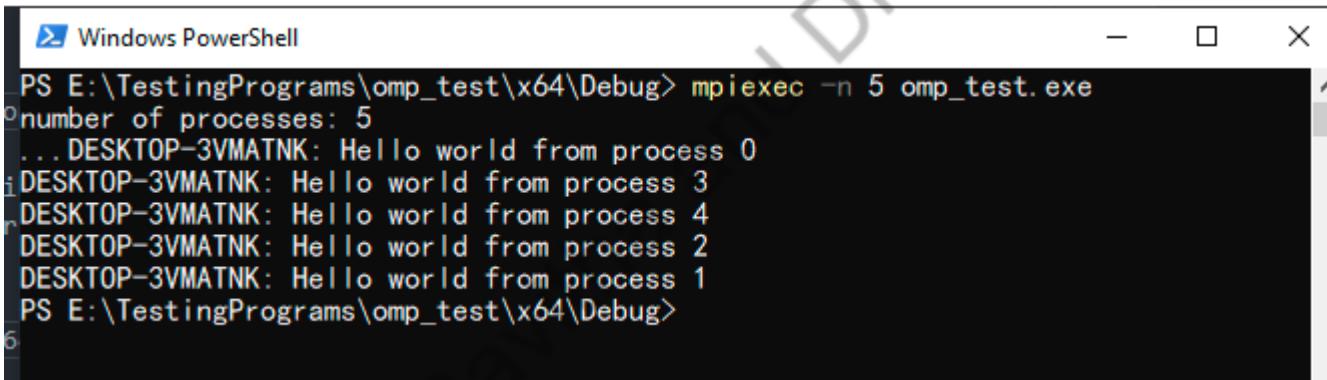
return 0;
}

Output
Show output from: Build
1>----- Build started: Project: omp_test, Configuration: Debug x64 -----
1>Source.cpp
1>omp_test.vcxproj -> E:\TestingPrograms\omp_test\x64\Debug\omp_test.exe
Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped
```

Open file explorer, E:\TestingPrograms\omp_test\x64\Debug folder (Your path will be different!)
Do right-click while pressing Shift button

Testing

- Enter `mpiexec -n 5 omp_test.exe` in powershell or command-line window(depending on your Windows version, for Win10 you will see powershell, but for early version you will see command-line)
- You can replace `5` with number of process you want,
- `omp_test.exe` must be replaced by the name of your builded .exe file



```
Windows PowerShell
PS E:\TestingPrograms\omp_test\x64\Debug> mpiexec -n 5 omp_test.exe
number of processes: 5
... DESKTOP-3VMATNK: Hello world from process 0
i DESKTOP-3VMATNK: Hello world from process 3
DESKTOP-3VMATNK: Hello world from process 4
DESKTOP-3VMATNK: Hello world from process 2
DESKTOP-3VMATNK: Hello world from process 1
PS E:\TestingPrograms\omp_test\x64\Debug>
```

Collective MPI Communication

Objectives:

1. Understand the usage of collective communication in MPI
2. Learn how to broadcast messages from root
3. Learn and use the APIs for distributing values from root and gathering the values in the root

Collective MPI Communication

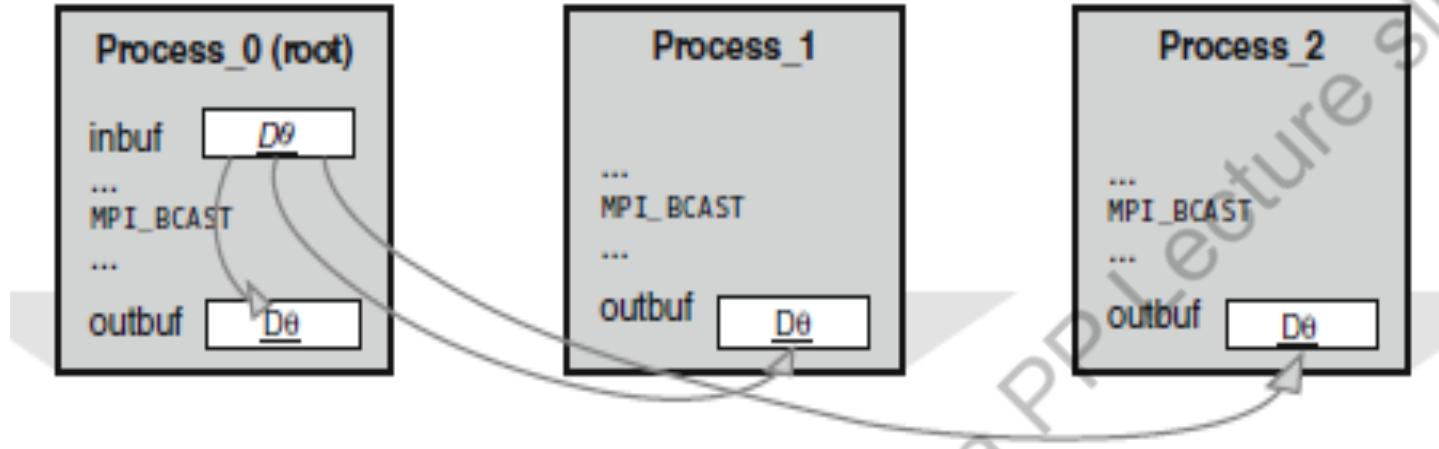
- Collective Communication routines
 - When **all processes** in a group participate in a global communication operation, the resulting communication is called a collective communication.
- The MPI collective operations are called by all processes in a communicator
 - MPI_BARRIER (comm)
 - MPI_BCAST (inbuf, incnt, intype, root, comm)
 - MPI_GATHER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)
 - MPI_SCATTER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)
- Tasks that can be elegantly implemented in this way are as follows:
 - global synchronization
 - reception of a local data item from all cooperating processes in the communicator

MPI_BARRIER (comm)

- This operation is used to synchronize the execution of a group of processes specified within the communicator **comm**
- When a process reaches this operation, it has to wait until all other processes have reached the MPI_BARRIER.
 - No process returns from MPI_BARRIER until all processes have called it.
- Programmer is responsible that all processes from communicator comm will really call MPI_BARRIER.
- The barrier is a simple way of separating two phases of a computation to ensure that messages generated in different phases do not interfere.
- MPI_BARRIER is a global operation that invokes all processes; therefore, it could be time-consuming. In many cases, the call to MPI_BARRIER should be avoided by an appropriate use of explicit addressing options, e.g., tag, source, or comm.

MPI_BCAST (inbuf, incnt, intype, root, comm)

- The operation implements a ***one-to-all broadcast operation*** whereby a single named ***process root*** sends its data to ***all other processes*** in the communicator, including to ***itself***
- Each process receives this data from the root process, which can be of any rank
- At the time of call, the input data are located in ***inbuf*** of process root and consists of ***incnt*** data items of a specified ***intype***
- After the call, the data are replicated in ***inbuf*** as output data of all remaining processes.
- As inbuf is used as an ***input argument at the root process, but as an output argument in all remaining processes***, it is of the INOUT type.

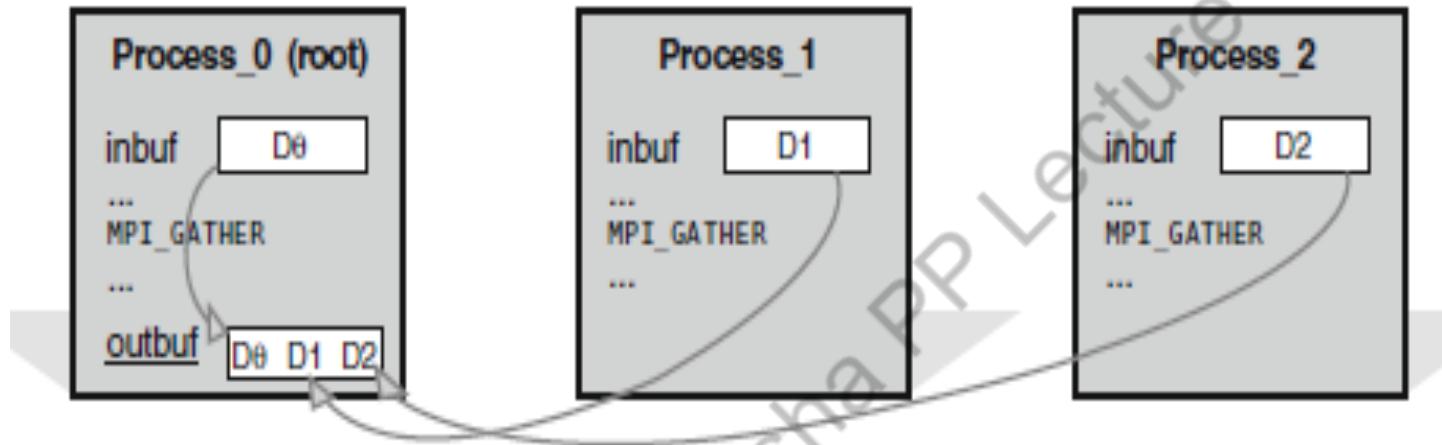


- A simple case of three processes is depicted above.
- The process with rank = 0 is the root process.
- Arrows symbolize the required message transfer.
- Note that all processes have to call `MPI_BCAST` to complete the requested data relocation

- The functionality of MPI_BCAST could be implemented, in the above example, by three calls to MPI_SEND in the root process and by a single corresponding MPI_RECV call in any remaining process.
- Usually, such an implementation will be less efficient than the original MPI_BCAST.
- All collective communications could be *time-consuming*.
- Their *efficiency* is strongly related with the *topology and performance of interconnection network*.
- The process ranked **Root** send the same message whose content is identified by the triple (*Address, Count, Datatype*) to **all** processes (including itself) in the communicator **Comm**.

MPI_GATHER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)

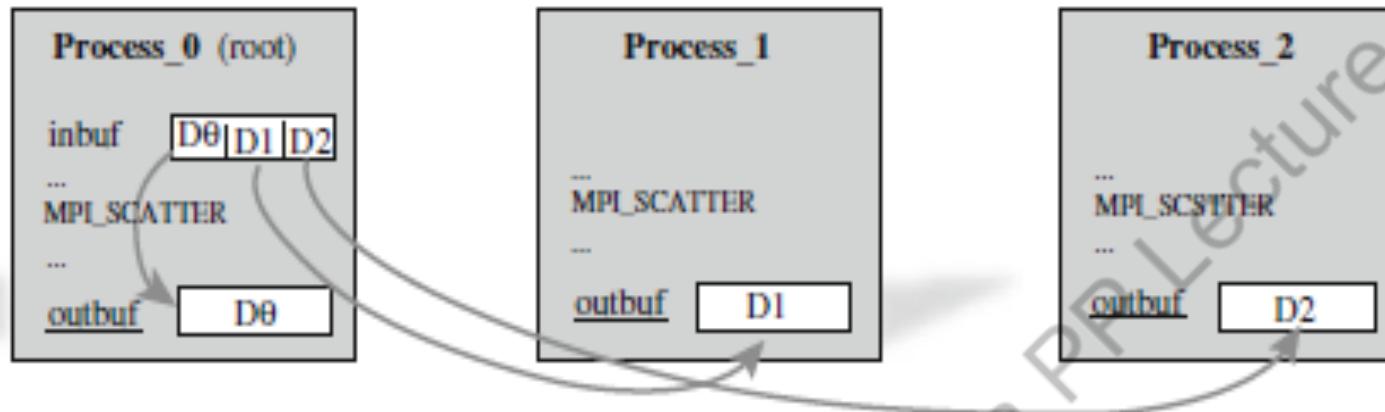
- *All-to-one collective communication* is implemented by MPI_GATHER.
- This operation is also called *by all processes* in the communicator.
- Each process, including *root process*, sends its *input data located in inbuf that consists of incnt data items of a specified intype, to the root process*, which can be of any rank.
- Note that the communication data can be different in count and type for each process.
- However, the root process has *to allocate enough space, through its output buffer, that suffices for all expected data*.
- After the return from MPI_GATHER in all processes, the data are collected in *outbuf of the root processes*.



- A schematic presentation of data relocation after the call to MPI_GATHER is shown in here.
- Example is for the case of three processes, where process with rank = 0 is the root process.
- Note again that all processes have to call MPI_GATHER to complete the requested data relocation.

MPI_SCATTER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)

- This operation works *inverse to MPI_GATHER*
 - i.e., it scatters data from *inbuf of process root to outbuf of all remaining processes, including itself.*
- Note that the count *outcnt and type outtype of the data in each of the receiver processes are the same, so, data is scattered into equal segments.*



- A schematic presentation of data relocation after the call to MPI_SCATTER is shown in Figure, for the case of three processes, where process with rank = 0 is the root process.
- Note again that all processes have to call MPI_SCATTER to complete the requested data relocation.

- There are also more complex collective operations
 - e.g., **MPI_GATHERV** and **MPI_SCATTERV** that allow a varying count of process data from each process and permit some options for process data placement on the root process.
- Such extensions are possible by changing the ***incnt*** and ***outcnt*** arguments from a single integer to an array of integers, and by providing a new array argument ***displs*** for specifying the displacement relative to *root* buffers at which to place the processes data.

Collective MPI Data Manipulations

- MPI provides a **set of operations** that perform several simple manipulations on the transferred data.
- These operations represent a combination of **collective communication** and **computational manipulation** in a single call and therefore simplify MPI programs.
- Collective MPI operations for data manipulation are based on data **reduction paradigm** that involves *reducing a set of numbers into a smaller set of numbers via a data manipulation*.

Example:

- Three pairs of numbers: $\{5, 1\}$, $\{3, 2\}$, $\{7, 6\}$, each representing the local data of a process
- Can be reduced in a pair of maximum numbers, i.e., $\{7, 6\}$
- Or in a sum of all pair numbers, i.e., $\{15, 9\}$

- Reduction operations defined by MPI:
 - ***MPI_MAX, MPI_MIN***; return either maximum or minimum data item;
 - ***MPI_SUM, MPI_PROD***; return either sum or product of aligned data items;
 - ***MPI_LAND, MPI_LOR, MPI_BAND, MPI_BOR***; return logical or bitwise AND or OR operation across the data items;
 - ***MPI_MAXLOC, MPI_MINLOC***; return the maximum or minimum value and the rank of the process that owns it;

- The MPI operation that implements all kind of data reductions is
MPI_REDUCE (inbuf, outbuf, count, type, op, root, comm).
- The ***MPI_REDUCE*** operation implements manipulation ***op*** on matching data items in input buffer ***inbuf*** from all processes in the communicator ***comm***.
- The results of the manipulation are stored in the output buffer ***outbuf*** of process root.
- The functionality of ***MPI_REDUCE*** is in fact an ***MPI_GATHER*** followed by manipulation ***op*** in process root.
- Reduce operations are implemented on a per-element basis, i.e., ***i***th elements from each process ***inbuf*** are combined into the ***i***th element in ***outbuf*** of process root.

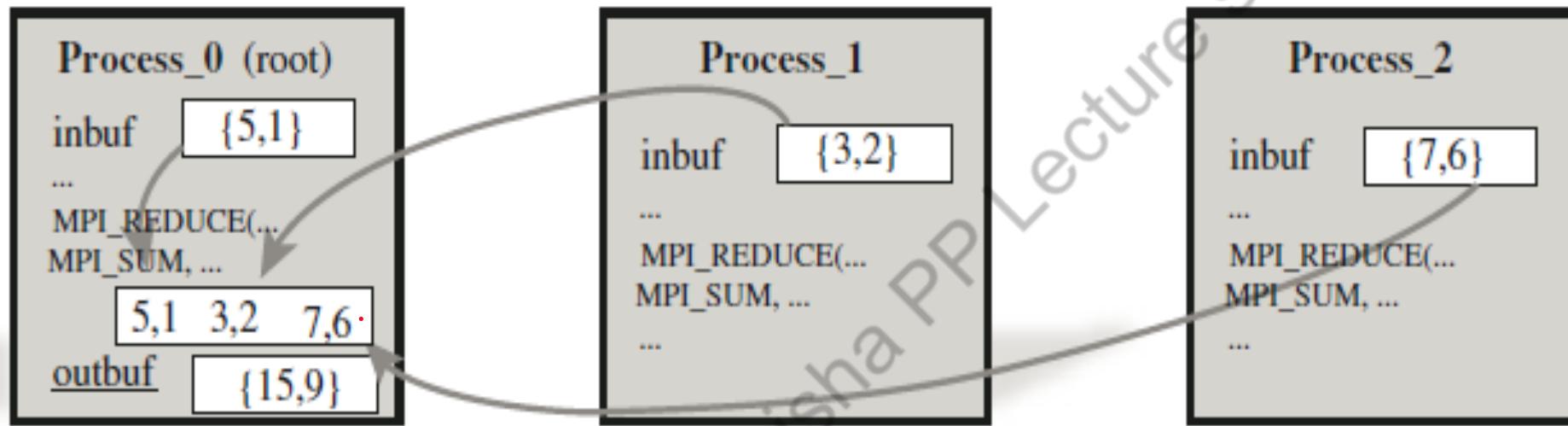


Fig. 4.7 Root process collects the data from input buffers of all processes, performs per-element MPI_SUM manipulation, and saves the result in its output buffer

- In many parallel calculations, a global problem domain is divided into subdomains that are assigned to corresponding processes.
- Often, an algorithm requires that all processes take a decision based on the global data.
 - For example, an iterative calculation can stop when the maximal solution error reaches a specified value.
 - An approach to the implementation of the stopping criteria could be the calculation of maximal sub-domain errors and collection of them in a root process, which will evaluate stopping criteria and broadcast the final result/decision to all processes

- MPI provides a specialized operation for this task, i.e.:

MPI_ALLREDUCE (*inbuf*, *outbuf*, *count*, *type*, *op*, *comm*)

- It improves simplicity and efficiency of MPI programs.
- It works as MPI_REDUCE followed by MPI_BCAST.
- Note that the argument root is not needed anymore because the final result has to be available to all processes in the communicator.
- For the same *inbuf* data as in Fig. 4.7 and with ***MPI_SUM*** manipulation, a call to ***MPI_ALLREDUCE*** will produce the result {15, 9}, in output buffers of all processes in the communicator.

- MPI_REDUCE (inbuf,outbuf,2,MPI_INT, MPI_SUM,0,MPI_COMM_WORLD)
 - Before the call, inbuf of three processes with ranks 0, 1, and 2 were: {5, 1}, {3, 2}, and {7, 6}, respectively.
 - After the call to the MPI_REDUCE the value in outbuf of root process is {15, 9}

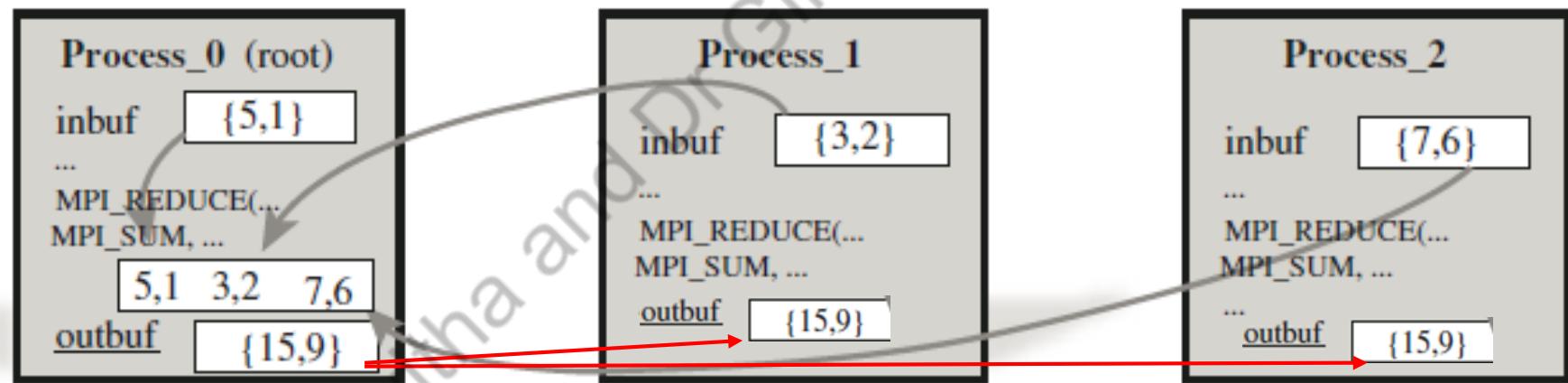


Fig. 4.7 Root process collects the data from input buffers of all processes, performs per-element MPI_SUM manipulation, and saves the result in its output buffer

List of MPI Operations :

Basic MPI operations:

```
MPI_INIT, MPI_FINALIZE,  
MPI_COMM_SIZE, MPI_COMM_RANK,  
MPI_SEND, MPI_RECV,
```

MPI operations for collective communication:

```
MPI_BARRIER,  
MPI_BCAST, MPI_GATHER, MPI_SCATTER,  
MPI_REDUCE, MPI_ALLREDUCE,
```

Control MPI operations:

```
MPI_WTIME, MPI_STATUS,  
MPI_INITIALIZED.
```

END

Dr Savitha and Dr Chisha PP Lecture slides

OpenMP

Dr Savitha and Dr Girisha PP Lecture Slides

Introduction

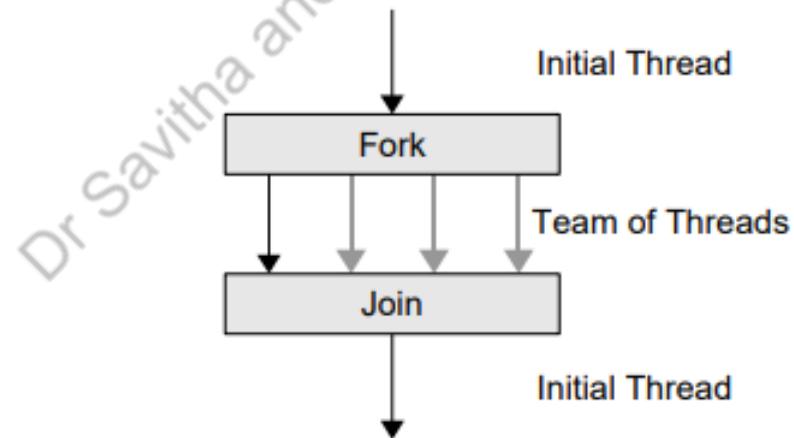
- The OpenMP **Application Programming Interface** (API) was developed to enable portable shared memory parallel programming
- The API is designed to permit an ***incremental*** approach to parallelizing an existing code, in which portions of a program are parallelized, possibly in successive steps
 - Contrast to the ***all-or-nothing*** conversion of an entire program in a single step that is typically required by other parallel programming paradigms

The Idea of OpenMP

- A thread is a **runtime entity** that is able to independently execute a stream of instructions
- The operating system creates a **process** to execute a program
 - It will allocate some resources to that process, including pages of memory and registers for holding values of objects
- If **multiple threads collaborate** to execute a program, they will share the resources, including the address space, of the corresponding process
- The individual threads need just a few resources of their own:
 - A **program counter** and an **area in memory** to save variables that are specific to it (including registers and a stack)

The Idea of OpenMP

- Multiple threads may be executed on a single processor or core via context switches;
 - They may be interleaved via simultaneous multithreading
- Threads running simultaneously on multiple processors or cores may work concurrently to execute a parallel program
- It supports the so-called **fork-join programming model**



The Idea of OpenMP

- The program starts as a single thread of execution, just like a sequential program
- The thread that executes this code is referred to as the *initial thread*
- Whenever an OpenMP parallel construct is encountered by a thread while it is executing the program, it *creates a team of threads* (this is the fork)
- It becomes the master of the team, and collaborates with the other members of the team to execute the code dynamically enclosed by the construct
- At the end of the construct, only the original thread, or master of the team, continues; all others terminate (this is the join)
- Each portion of code enclosed by a parallel construct is called a *parallel region*

The Feature Set

- The OpenMP API comprises a set of *compiler directives*, *runtime library routines*, and *environment variables* to specify shared-memory parallelism
 - **Directive:** A statement written in the source code of a program that lets the programmer instruct the compiler to perform a specific operation within the compilation phase.
 - **Environmental variables** define the characteristics of a specific environment
 - **Library routines** (API) that helps to do some task
- An OpenMP directive is a specially formatted comment or pragma that generally applies to the executable code immediately following it in the program
 - *A directive or OpenMP routine generally affects only those threads that encounter it*
- Many of the directives are applied to a *structured block of code*
 - A sequence of executable statements with a single entry at the top and a single exit at the bottom

The Feature Set

- OpenMP provides means for the user to:
 - Create teams of threads for parallel execution
 - Specify how to share work among the members of a team
 - Declare both shared and private variables
 - Synchronize threads and enable them to perform certain operations exclusively (i.e., without interference by other threads)

Creating Teams of Threads

- A team of threads is created to execute the code in a parallel region of an OpenMP program
 - To accomplish this, the programmer simply specifies the parallel region by inserting a parallel directive immediately before the code that is to be executed in parallel to mark its start
- At the end of a parallel region is an **implicit barrier synchronization**
 - This means that no thread can progress until all other threads in the team have reached that point in the program
- If a team of threads executing a parallel region encounters another parallel directive, each thread in the current team creates a new team of threads and becomes its master
- Nesting enables the realization of **multilevel parallel programs**

Sharing Work among Threads

- If the programmer does not specify how the work in a parallel region is to be shared among the executing threads, they will each **redundantly** execute all of the code
 - *This approach does not speed up the program*
- The OpenMP work-sharing directives are provided for the programmer to state how the computation in a structured block of code is to be distributed among the threads
- Unless explicitly overridden by the programmer, an implicit barrier synchronization also exists at the end of a work-sharing construct

Sharing Work among Threads

- Probably the most common work-sharing approach is to distribute the work **for (C/C++) loop** among the threads in a team
- To accomplish this, the programmer inserts the appropriate directive immediately before each loop within a parallel region that is to be shared among threads
- *Work-sharing directives cannot be applied to all kinds of loops that occur in C/C++ code*
- Many programs, especially scientific applications, spend a large fraction of their time in loops performing calculations on array elements and so this strategy is widely applicable and often very effective

Sharing Work among Threads

- All OpenMP strategies for sharing the work in loops assign one or more disjoint sets of iterations to each thread
- The programmer may specify the method used to **partition the iteration set**
- *The most straightforward strategy assigns one contiguous chunk of iterations to each thread*
- *More complicated strategies include dynamically computing the next chunk of iterations for a thread*
- If the programmer does not provide a strategy, then an implementation-defined default will be used.

Sharing Work among Threads

- *It must be possible to determine the number of iterations in the loop upon entry, and this number may not change while the loop is executing*
 - **While construct**, for example, may not satisfy this condition
- Furthermore, a loop is suitable for sharing among threads only if its **iterations are independent**
- By this, we mean that the order in which the iterations are performed has no bearing on the outcome

Sharing Work among Threads

- If giving distinct pieces of work to the individual threads
- This approach is suitable when independent computations are to be performed and the order in which they are carried out is irrelevant
- It is straightforward to specify this by using the corresponding OpenMP directive
- The programmer must ensure that the computations can truly be executed in parallel
- *It is also possible to specify that just one thread should execute a block of code in a parallel region*

The OpenMP Memory Model

- OpenMP is based on the shared-memory model; hence, by default, data is shared among the threads and is visible to all of them
- Sometimes, however, one needs variables that have thread-specific values
- When each thread has its own copy of a variable, so that it may potentially have a different value for each of them, we say that the variable is private
 - For example, when a team of threads executes a parallel loop, each thread needs its own value of the iteration variable.
- The use of private variables can be beneficial in several ways:
 - They can reduce the frequency of updates to shared memory
 - Thus, they may help avoid hot spots, or competition for access to certain memory locations, which can be expensive if large numbers of threads are involved.

The OpenMP Memory Model

- Threads need a place to store their private data at run time
- For this, each thread has its own special region in memory known as ***the thread stack***
- A ***flush operation*** makes sure that the thread calling it has the same values for shared data objects as does main memory
- Hence, new values of any shared objects updated by that thread are written back to shared memory, and the thread gets any new values produced by other threads for the shared data it reads

OpenMP Code Structure

```
#include <stdlib.h>
#include<stdio.h>
#include "omp.h"
int main()
{
    #pragma omp parallel
    {
        int ID = omp_get_thread_num();
        printf("Hello (%d)\n", ID);
        printf(" world (%d)\n", ID);
    }
}
```

A sample OpenMP program

```
main( )
{
    omp_set_num_threads( 8 );
    #pragma omp parallel default(none)
    {
        printf( "Hello, World, from thread #%d ! \n" , omp_get_thread_num( ) );
    }
    return 0;
}
```

A sample OpenMP program

First Run

```
Hello, World, from thread #6 !  
Hello, World, from thread #1 !  
Hello, World, from thread #7 !  
Hello, World, from thread #5 !  
Hello, World, from thread #4 !  
Hello, World, from thread #3 !  
Hello, World, from thread #2 !  
Hello, World, from thread #0 !
```

Second Run

```
Hello, World, from thread #0 !  
Hello, World, from thread #7 !  
Hello, World, from thread #4 !  
Hello, World, from thread #6 !  
Hello, World, from thread #1 !  
Hello, World, from thread #3 !  
Hello, World, from thread #5 !  
Hello, World, from thread #2 !
```

Third Run

```
Hello, World, from thread #2 !  
Hello, World, from thread #5 !  
Hello, World, from thread #0 !  
Hello, World, from thread #7 !  
Hello, World, from thread #1 !  
Hello, World, from thread #3 !  
Hello, World, from thread #4 !  
Hello, World, from thread #6 !
```

Fourth Run

```
Hello, World, from thread #1 !  
Hello, World, from thread #3 !  
Hello, World, from thread #5 !  
Hello, World, from thread #2 !  
Hello, World, from thread #4 !  
Hello, World, from thread #7 !  
Hello, World, from thread #6 !  
Hello, World, from thread #0 !
```

Thread Synchronization

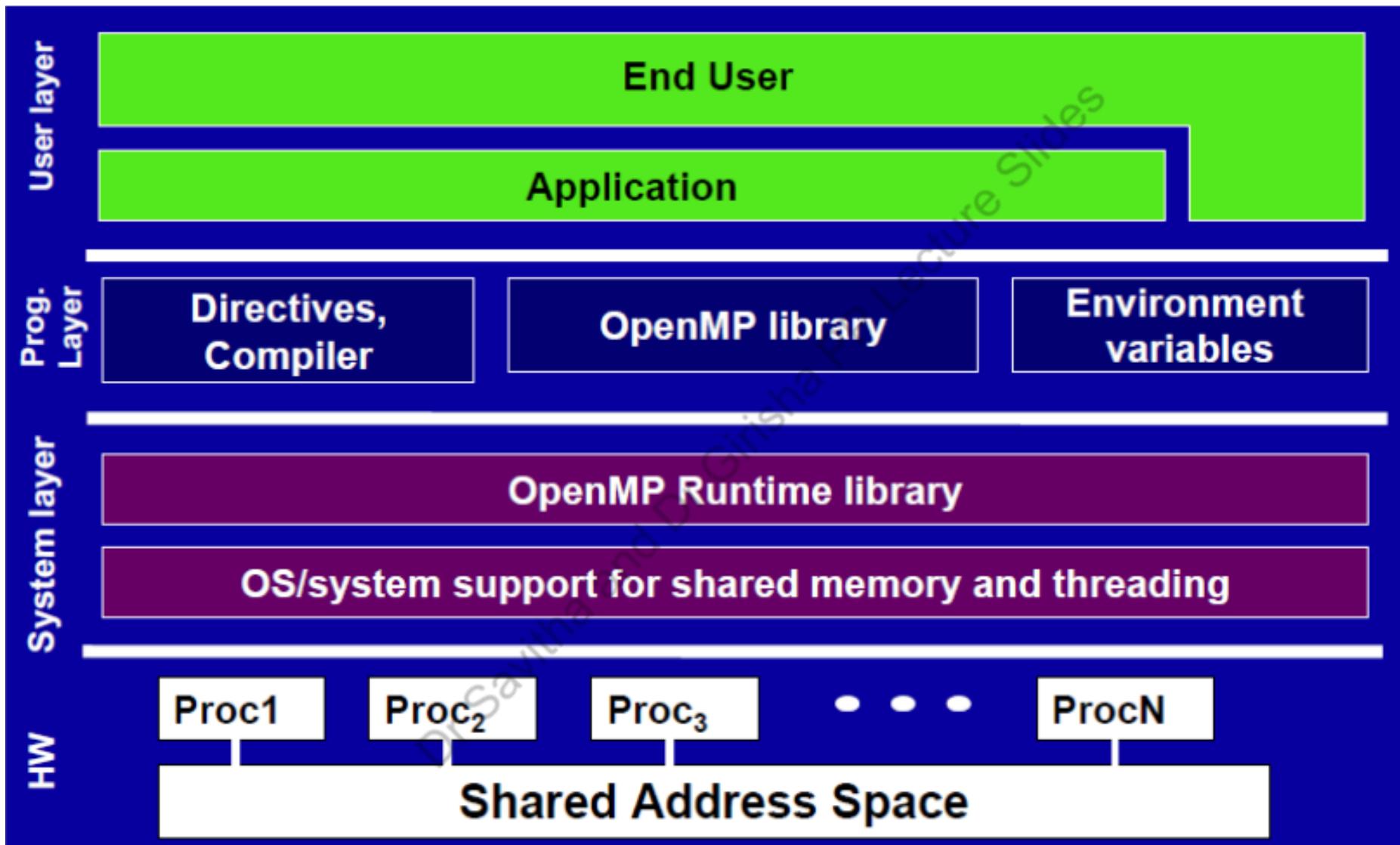
- *Why synchronization?:*
 - *Synchronizing, or coordinating the actions of, threads is sometimes necessary in order to ensure the proper ordering of their accesses to shared data and to prevent data corruption*
- By default, OpenMP gets threads to wait at the end of a *worksharing construct or parallel region* until all threads in the team executing it have finished their portion of the work. Only then can they proceed.
This is known as a barrier
- Sometimes a programmer may need to ensure that only one thread at a time works on a piece of code
- *Synchronization points are those places in the code where synchronization has been specified, either explicitly or implicitly*

OpenMP Programming Styles

- OpenMP encourages structured parallel programming and relies heavily on *distributing the work in loops among threads*. But sometimes the amount of loop-level parallelism in an application is limited
- *One can also write OpenMP programs that do not rely on work-sharing directives but rather assign work explicitly to different threads using their thread numbers.*
- This approach can lead to highly efficient code
 - However, the programmer must then insert synchronization manually to ensure that accesses to shared data are correctly controlled
 - Those approaches that require manual assignment of work to threads and that need explicit synchronization are often called “*low-level programming*”

What is OpenMP?

- **Open specifications for Multi Processing**
- An **Application Program Interface** (API) that is used to explicitly direct multi-threaded, shared memory parallelism
- API components:
 - Compiler directives
 - Runtime library routines
 - Environment variables
- Portability
 - API is specified for C/C++ and Fortran
 - Implementations on almost all platforms including Unix/Linux and Windows
- Standardization
 - Jointly defined and endorsed by major computer hardware and software vendors
 - Possibility to become ANSI standard



Threads & Process

- A process is an instance of a computer program that is being executed. It contains the program code and its current activity.
- A thread of execution is the smallest unit of processing that can be scheduled by an operating system
- Differences between threads and processes:
 - **A thread is contained inside a process.** Multiple threads can exist within the same process and share resources such as memory. The threads of a process share the latter's instructions (code) and its context (values that its variables reference at any given moment)
 - Different processes do not share these resources.

Threads & Process

- A process contains all the information needed to execute the program
 - Process ID
 - Program code
 - Data on run time stack
 - Global data
 - Data on heap
- Each process has its own address space
- In multitasking, processes are given time slices in a round robin fashion
- If computer resources are assigned to another process, the status of the present process has to be saved, in order that the execution of the suspended process can be resumed at a later time.

Threads & Process

- Each process consists of multiple independent instruction streams (or threads) that are assigned computer resources by some scheduling procedure
- Threads of a process share the address space of this process
 - Global variables and all dynamically allocated data objects are accessible by all threads of a process
- Each thread has its own run-time stack, register, program counter
- Threads can communicate by reading/writing variables in the common address space

OpenMP Programming Model

- Shared memory, thread-based parallelism
 - OpenMP is based on the existence of multiple threads in the shared memory programming paradigm
 - A shared memory process consists of multiple threads
- Explicit Parallelism
 - Programmer has full control over parallelization. OpenMP is not an automatic parallel programming model
- Compiler directive based
 - Most OpenMP parallelism is specified through the use of compiler directives which are embedded in the source code

What OpenMP Isn't

- OpenMP doesn't check for **data dependencies**, **data conflicts**, **deadlocks**, or **race conditions**. You are responsible for avoiding those yourself
- OpenMP doesn't check for **non-conforming** code sequences
- OpenMP doesn't **guarantee identical behavior** across vendors or hardware, or even between multiple runs on the same vendor's hardware
- OpenMP doesn't guarantee the **order in which threads execute**, just that they do execute
- OpenMP is not **overhead-free**
- OpenMP does not prevent you from writing code that triggers **cache performance problems**

OpenMP: parallel regions

- A parallel region within a program is specified as

```
#pragma omp parallel [clause [[,] clause] ...]  
    Structured-block
```

- A team of threads is formed
- Thread that encountered the omp parallel directive becomes the **master thread** within this team
- **The structured-block is executed by every thread in the team.**
- At the end, there is an implicit **barrier**
- Only after **all threads have finished, the threads created by this directive are terminated and only the master resumes execution**
- A parallel region might be refined by a list of clause

- Each thread in the team is assigned a unique thread number (also referred to as the “thread id”) to identify it.
- They range from zero (for the master thread) up to one less than the number of threads within the team, and they can be accessed by the programmer.
- Although the parallel region is executed by all threads in the team, each thread is allowed to follow a different path of execution.

- Each thread will execute all code in the parallel region, so that each thread will perform the first print statement.
- However, only one thread will actually execute the second print statement (assuming there are at least three threads in the team), since we used the thread number to control its execution.

```
#pragma omp parallel
{
    printf("The parallel region is executed by thread %d\n",
           omp_get_thread_num());

    if (omp_get_thread_num() == 2) {
        printf(" Thread %d does things differently\n",
               omp_get_thread_num());
    }
} /*-- End of parallel region --*/
```

Output:

```
The parallel region is executed by thread 0
The parallel region is executed by thread 3
The parallel region is executed by thread 2
    Thread 2 does things differently
The parallel region is executed by thread 1
```

Clauses supported by the parallel construct

if (scalar-expression)	(C/C++)
if (scalar-logical-expression)	(Fortran)
num_threads (integer-expression)	(C/C++)
num_threads (scalar-integer-expression)	(Fortran)
private (list)	
firstprivate (list)	
shared (list)	
default (none shared)	(C/C++)
default (none shared private)	(Fortran)
copyin (list)	
reduction (operator:list)	(C/C++)
reduction ({operator intrinsic_procedure_name}:list)	(Fortran)

There are several restrictions on the parallel construct and its clauses:

- A program that branches into or out of a parallel region is nonconforming.
 - In other words, if a program does so, then it is *illegal*, and the behavior is undefined.
- A program must not depend on any ordering of the evaluations of the clauses of the parallel directive or on any side effects of the evaluations of the clauses.
- At most one ***if*** clause can appear on the directive.
- At most one ***num_threads*** clause can appear on the directive. The expression for the clause must evaluate to a positive integer value.
- In C++ there is an additional constraint. A throw inside a parallel region must cause execution to resume within the same parallel region, *and* it must be caught by the same thread that threw the exception.

Active parallel region and an ***Inactive parallel region***

- A parallel region is active if it is executed by a team of threads consisting of more than one thread.
- If it is executed by one thread only, it has been serialized and is considered to be inactive.
- **Example:** - one can specify that a parallel region be conditionally executed, in order to be sure that it contains enough work for this to be worthwhile.
 - If the condition does not hold at run time, then the parallel region will be inactive.

`#pragma omp parallel if (n > 5) default(none)`

Here the parallel region is executed only if n is greater than 5.

- A parallel region may also be inactive if it is nested within another parallel region and this feature is either disabled or not provided by the implementation

Sharing the Work among Threads in an OpenMP Program

- C/C++ has three work-sharing constructs.

Functionality	Syntax in C/C++
Distribute iterations over the threads	<code>#pragma omp for</code>
Distribute independent work units	<code>#pragma omp sections</code>
Only one thread executes the code block	<code>#pragma omp single</code>

- Many applications can be parallelized by using just a parallel region and one or more of these constructs, possibly with clauses
- Work-sharing construct, along with its terminating construct where appropriate, specifies a region of code whose work is to be distributed among the executing threads; it also specifies the manner in which the work in the region is to be parceled out.
- A work-sharing region must bind to an active parallel region in order to have an effect

- Since work-sharing directives may occur in procedures that are invoked both from within a parallel region as well as outside of any parallel regions, they may be exploited during some calls and ignored during others

The two main rules regarding work-sharing constructs are as follows:

- Each work-sharing region must be encountered by all threads in a team or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.
- A work-sharing construct does not launch new threads and does not have a barrier on entry. By default, threads wait at a barrier at the end of a work-sharing region until the last thread has completed its share of the work.

Loop Construct

- The *loop construct* causes the iterations of the loop immediately following it to be executed in parallel.
- At run time, the loop iterations are distributed across the threads. This is probably the most widely used of the work-sharing features.

```
#pragma omp for [clause[, clause]...]  
    for-loop
```

Note the lack of curly braces.
These are implied with the construct.

- In C and C++ programs, the use of this construct is limited to those kinds of loops where the number of iterations can be counted;
 - that is, the loop must have an integer counter variable whose value is incremented (or decremented) by a fixed amount at each iteration until some specified upper (or lower) bound is reached

- The loop header must have the general form.

for (init-expr ; var relop b ; incr-expr)

where

- *init-expr* stands for the initialization of the loop counter *var* via an integer expression
- *b* is also an integer expression
- *relop* is one of the following: <, <=, >, >=
- The *incr-expr* is a statement that increments or decrements *var* by an integer amount using a standard operator (++, -, +=, -=). Alternatively, it may take a form such as *var = var + incr*.

- A parallel directive is used to define a parallel region and then share its work among threads via the for work sharing directive:
 - the #pragma omp for directive states that iterations of the loop following it will be distributed
 - omp get thread num(), is used obtain and print the number of the executing thread in each iteration
 - Clauses are added to the parallel construct that state which data in the region is shared and which is private.
 - Loop variable i is explicitly declared to be a private variable by the compiler, which means that each thread will have its own copy of i.
 - Unless the programmer takes special action its value is also undefined after the loop has finished

Since the total number of iterations is 9 and four threads are used, one thread has to execute the additional iteration. In this case it turns out to be thread 0, the so-called master thread

```
#pragma omp parallel shared(n) private(i)
{
    #pragma omp for
    for (i=0; i<n; i++)
        printf("Thread %d executes loop iteration %d\n",
               omp_get_thread_num(),i);
} /*-- End of parallel region --*/
```

Output : The example is executed for $n = 9$ and uses four threads

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
```

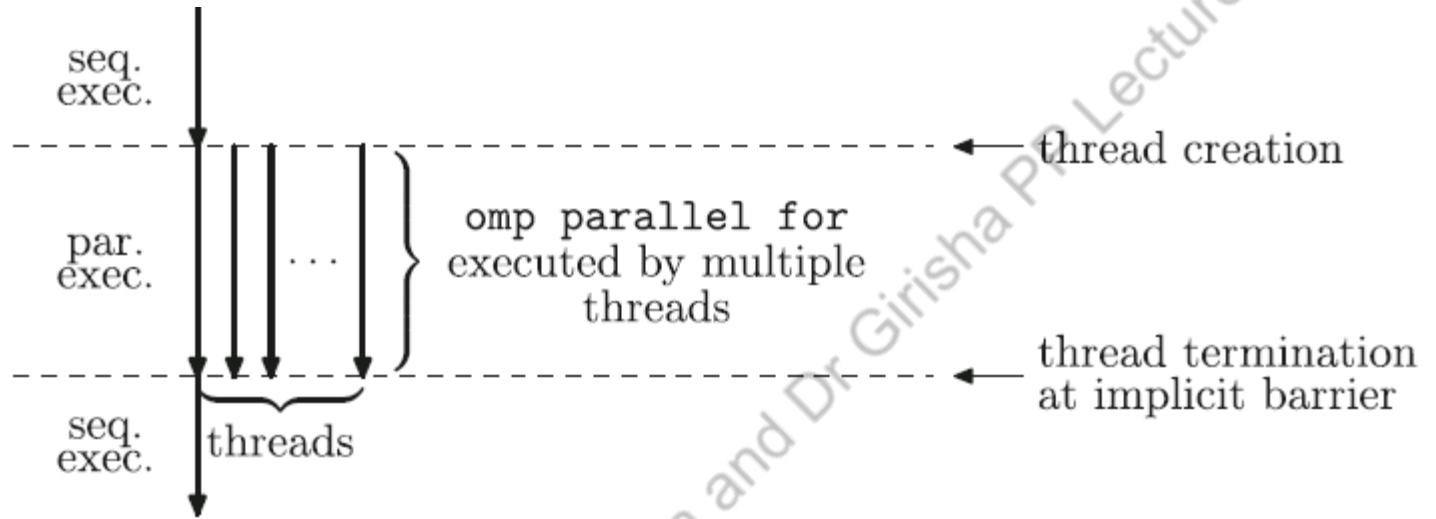


Fig. 3.5 Execution of the program for printing out integers as implemented in Listing 3.3

Divide for-loop for parallel sections

```
for (int i=0; i<8; i++) x[i]=0; //run on 4 threads
```



```
#pragma omp parallel // Assume number of threads=4
```

```
{  
    int numt=omp_get_num_thread();  
    int id = omp_get_thread_num(); //id=0, 1, 2, or 3  
    for (int i=id; i<8; i+=numt)  
        x[i]=0;
```

```
}
```

Thread 0

```
Id=0;  
x[0]=0;  
x[4]=0;
```

Thread 1

```
Id=1;  
x[1]=0;  
x[5]=0;
```

Thread 2

```
Id=2;  
x[2]=0;  
x[6]=0;
```

Thread 3

```
Id=3;  
x[3]=0;  
x[7]=0;
```

Use pragma parallel for

```
for (int i=0; i<8; i++) x[i]=0;
```



```
#pragma omp parallel for
{
    for (int i=0; i<8; i++)
        x[i]=0;
}
```

System divides loop iterations to threads

Id=0;
x[0]=0;
X[4]=0;

Id=1;
x[1]=0;
X[5]=0;

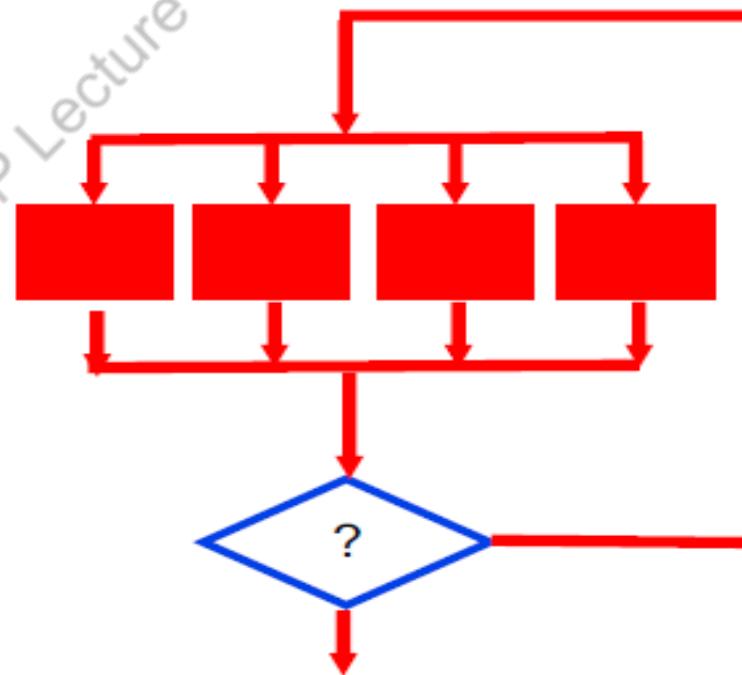
Id=2;
x[2]=0;
X[6]=0;

Id=3;
x[3]=0;
X[7]=0;

Programming Model – Parallel Loops

- Requirement for parallel loops
 - No data dependencies (reads/write or write/write pairs) between iterations!
- Preprocessor calculates loop bounds and divide iterations among parallel threads

```
#pragma omp parallel for
for( i=0; i < 25; i++ )
{
    printf("Foo");
}
```



Example

```
for (i=0 ; i<max; i++) zero[i] = 0;
```

- Breaks *for loop* into chunks, and allocate each to a separate thread
 - e.g. if max = 100 with 2 threads:
assign 0-49 to thread 0, and 50-99 to thread 1
 - Must have relatively simple “shape” for an OpenMP-aware compiler to be able to parallelize it
 - Necessary for the run-time system to be able to determine how many of the loop iterations to assign to each thread
 - No premature exits from the loop allowed
 - i.e. No break, return, exit, goto statements
- ← **In general,
don't jump
outside of
any pragma
block**

OpenMP: parallel loops

```
x[ 0 ] = 0.;  
y[ 0 ] *= 2.;  
for( int i = 1; i < N; i++ )  
{  
    x[ i ] = x[ i-1 ] + 1.;  
    y[ i ] *= 2.;  
}
```

Because of the loop dependency, this whole thing is not parallelizable:

```
x[ 0 ] = 0.;  
for( int i = 1; i < N; i++ )  
{  
    x[ i ] = x[ i-1 ] + 1.;  
}  
  
#pragma omp parallel for shared(y)  
for( int i = 0; i < N; i++ )  
{  
    y[ i ] *= 2.;  
}
```

But, it can be broken into one loop that is not parallelizable, plus one that is:

OpenMP: parallel loops

```
for( int i = 1; i < N; i++ )  
{  
    for( int j = 0; j < M; j++ )  
    {  
        ...  
    }  
}
```

Ah-ha – trick question. You put it on both!

How many for-loops to collapse into one loop

```
#pragma omp parallel for collapse(2)  
for( int i = 1; i < N; i++ )  
{  
    for( int j = 0; j < M; j++ )  
    {  
        ...  
    }  
}
```

The Sections Construct

- The *sections construct* is the easiest way to get different threads to carry out different kinds of work, since it permits us to specify several different code regions, each of which will be executed by one of the threads.

```
#pragma omp parallel shared(n,a,b) private(i)
{
    #pragma omp for
    for (i=0; i<n; i++)
        a[i] = i;

    #pragma omp for
    for (i=0; i<n; i++)
        b[i] = 2 * a[i];
} /*-- End of parallel region --*/
```

Two work-sharing loops in one parallel region

- No guarantee that the distribution of iterations to threads is identical for both loops but the implied barrier ensures that results are available when needed.

- Each section must be a structured block of code that is independent of the other sections.
- At run time, the specified code blocks are executed by the threads in the team. Each thread executes one code block at a time, and each code block will be executed exactly once.
- If there are fewer threads than code blocks, some or all of the threads execute multiple code blocks.
- If there are fewer code blocks than threads, the remaining threads will be idle.
- The assignment of code blocks to threads is implementation-dependent.
- Commonly used to execute function or subroutine calls in parallel.

```
#pragma omp sections [clause[, clause]...]
{
  [#pragma omp section ]
  structured block
  [#pragma omp section ]
  structured block
  ...
}
```

Example of parallel sections

- If two or more threads are available, one thread invokes funcA() and another thread calls funcB(). Any other threads are idle.
- This code fragment contains one sections construct, comprising two sections.
- This limits the parallelism to two threads.
- If two or more threads are available, function calls funcA and funcB are executed in parallel.
- If only one thread is available, both calls to funcA and funcB are executed, but in sequential order.
- One cannot make any assumption on the specific order in which section blocks are executed.
- Even if these calls are executed sequentially, because the directive is not in an active parallel region, funcB may be called before funcA.

```
#pragma omp parallel  
{  
    #pragma omp sections  
    {  
        #pragma omp section  
        (void) funcA();  
  
        #pragma omp section  
        (void) funcB();  
    } /*-- End of sections block --*/  
}  
/*-- End of parallel region --*/
```

Load-balancing problem

- Depending on the type of work performed in the various code blocks and the number of threads used, this construct might lead to a *load-balancing* problem.
- This occurs when threads have different amounts of work to do and thus take different amounts of time to complete.
- A result of load imbalance is that some threads may wait a long time at the next barrier in the program, which means that the hardware resources are not being efficiently exploited.
- It may sometimes be possible to eliminate the barrier at the end of the construct but that does not overcome the fundamental problem of a load imbalance *within* the sections construct.

The Single Construct

- The *single construct* is associated with the structured block of code immediately following it and specifies that this block should be executed by one thread only.
- It does not state which thread should execute the code block; indeed, the thread chosen could vary from one run to another.
- This construct should be used when we do not care which thread executes this part of the application, as long as the work gets done by exactly one thread.
- The other threads wait at a barrier until the thread executing the single code block has completed.

Syntax:

#pragma omp single [clause[,] clause]. . .] *structured block*

- Only one thread executes the structured block

Example of the single construct

- Only one thread initializes the shared variable a. This variable is then used to initialize vector b in the parallelized for-loop

```
#pragma omp parallel shared(a,b) private(i)
{
    #pragma omp single
    {
        a = 10;
        printf("Single construct executed by thread %d\n",
               omp_get_thread_num());
    }
}
```

/ A barrier is automatically inserted here */*

```
#pragma omp for
for (i=0; i<n; i++)
    b[i] = a;
} /*-- End of parallel region --*/
```

```
printf("After the parallel region:\n");
for (i=0; i<n; i++)
    printf("b[%d] = %d\n",i,b[i]);
```

- A barrier is essential before the **#pragma omp for** loop. Without such a barrier, some threads would begin to assign values to elements of b before a has been assigned a value
- Every thread would write the same value of 10 to the same variable a. However, this approach raises a hardware issue.
 - Depending on the data type, the processor details, and the compiler behavior, the write to memory might be translated into a sequence of store instructions, each store writing a subset of the variable.
 - **Example:** A variable 8 bytes long might be written to memory through 2 store instructions of 4 bytes each
 - Multiple threads could do this at the same time, resulting in an arbitrary combination of bytes in memory.
 - This issue is also related to the memory consistency model.
 - Moreover, multiple stores to the same memory address are bad for performance

Combined Parallel Work-Sharing Constructs

- They are shortcuts that can be used when a parallel region comprises precisely one work-sharing construct, that is, the work sharing region includes all the code in the parallel region.
- The combined parallel work-sharing constructs allow certain clauses that are supported by both the parallel construct and the workshare construct.

```
#pragma omp parallel  
{  
#pragma omp for  
for (.....)  
}
```

A single work-sharing loop in a parallel region – For cases like this OpenMP provides a shortcut.

```
#pragma omp parallel for  
for (.....)
```

Syntax of the combined constructs in C/C++

- The combined constructs may have a performance advantage over the more general parallel region with just one work-sharing construct embedded.

Full version	Combined construct
<pre>#pragma omp parallel { #pragma omp for for-loop }</pre>	<pre>#pragma omp parallel for for-loop</pre>
<pre>#pragma omp parallel { #pragma omp sections { [#pragma omp section] structured block [#pragma omp section] structured block ... } }</pre>	<pre>#pragma omp parallel sections { [#pragma omp section] structured block [#pragma omp section] structured block ... }</pre>

- The main advantage of using these combined constructs is readability.
- When the combined construct is used, a compiler knows what to expect and may be able to generate slightly more efficient code.

Clauses to Control Parallel and Work-Sharing Constructs

Shared Clause :

- The shared clause is used to specify which data will be shared among the threads executing the region it is associated with.
- Simply stated, there is one unique instance of these variables, and each thread can freely read or modify the values.
- The syntax for this clause is **shared(*list*)** . All items in the list are data objects that will be shared among the threads in the team.

```
#pragma omp parallel for
shared(a)
    for (i=0; i<n; i++)
    {
        a[i] += i;
    } /*-- End of parallel for --
*/
```

- Here, vector **a** is declared to be shared. This implies that all threads are able to read and write elements of **a**.
- Within the parallel loop, each thread will access the pre-existing values of those elements **a[i]** of **a** that it is responsible for updating and will compute their new values.
- After the parallel region is finished, all the new values for elements of **a** will be in main memory, where the master thread can access them.

- Multiple threads might attempt to simultaneously update the same memory location or that one thread might try to read from a location that another thread is updating.
- Special care has to be taken to ensure that neither of these situations occurs and that accesses to shared data are ordered as required by the algorithm.

Private Clause

- Since the loop iterations are distributed over the threads in the team, each thread must be given a unique and local copy of the loop variable *i* so that it can safely modify the value.
- Otherwise, a change made to *i* by one thread would affect the value of *i* in another thread's memory, thereby making it impossible for the thread to keep track of its own set of iterations
- private clause is used when data objects in a parallel region or work-sharing construct require which threads should be given their own copies

The syntax is **private(*list*)**

- Each variable in the list is replicated so that each thread in the team of threads has exclusive access to a local copy of this variable.
- Changes made to the data by one thread are not visible to other threads.

- If variable *a* had been specified in a shared clause, multiple threads would attempt to update the *same* variable with different values in an uncontrolled manner.
- The final value would thus depend on which thread happened to last update **a**. (This bug is a data race condition.) Therefore, the usage of *a* requires us to specify it to be a private variable, ensuring that each thread has its own copy.

```
#pragma omp parallel for private(i,a)
for (i=0; i<n; i++)
{
    a = i+1;
    printf("Thread %d has a value of a = %d for i = %d\n",
    omp_get_thread_num(),a,i);
} /*-- End of parallel for --*/
```

Thread 0 has a value of a = 1 for i = 0
Thread 0 has a value of a = 2 for i = 1
Thread 2 has a value of a = 5 for i = 4
Thread 1 has a value of a = 3 for i = 2
Thread 1 has a value of a = 4 for i = 3

Lastprivate Clause

- It ensures that the last value of a data object listed is accessible after the corresponding construct has completed execution.
- In a parallel program, we must explain what “last” means.
- In the case of its use with a work-shared loop, the object will have the value from the iteration of the loop that would be last in a sequential execution.
- If the lastprivate clause is used on a sections construct, the object gets assigned the value that it has at the end of the lexically last sections construct.
- The syntax is **lastprivate(*list*)**.

- Variable **a** now has the lastprivate data-sharing attribute
- There is a print statement after the parallel region so that we can check on the value **a** has at that point.
- According to the definition of “last,” the value of variable **a** after the parallel region, should correspond to that computed when $i = n - 1$
- **Output:** Variable **n** is set to 5, and three threads are used. The last value of variable **a** corresponds to the value for $i = 4$

```
#pragma omp parallel for private(i) lastprivate(a)
for (i=0; i<n; i++)
{
    a = i+1;
    printf("Thread %d has a value of a = %d for i = %d\n",
           omp_get_thread_num(),a,i);
} /*-- End of parallel for --*/
```

```
printf("Value of a after parallel for: a = %d\n",a);
```

Output:

```
Thread 0 has a value of a = 1 for i = 0
Thread 0 has a value of a = 2 for i = 1
Thread 2 has a value of a = 5 for i = 4
Thread 1 has a value of a = 3 for i = 2
Thread 1 has a value of a = 4 for i = 3
Value of a after parallel for: a = 5
```

Firstprivate Clause

- Variables that are declared to be “firstprivate” are private variables, but they are pre-initialized with the value of the variable with the same name before the construct.
- The initialization is carried out by the initial thread prior to the execution of the construct.
- The firstprivate clause is supported on the parallel construct, plus the work-sharing loop, sections, and single constructs.
- The syntax is **firstprivate(*list*).**

```
#include<stdio.h>
#include<omp.h>

int main()
{
    int a = 5;
#pragma omp parallel num_threads(5) firstprivate(a)
    {
        int b = 10;
        a = a + b;
        printf("Parallel region: %d\n", a);
    }

    printf("Outside parallel region: %d\n", a);
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
Parallel region: 15
Outside parallel region: 5

D:\DSCA\Parallel_Computing\Lab_Programs\Ex
ith code 0.
To automatically close the console when de
le when debugging stops.
Press any key to close this window . . .
```

Default Clause

- The default clause is used to give variables a default data-sharing attribute
- Example:
 - **default(shared)** assigns the shared attribute to all variables referenced in the construct.
 - The **default(private)** clause, which is not supported in C/C++, makes all variables private by default. It is applicable to the parallel construct only.
- This clause is used to define the data-sharing attribute of the majority of the variables in a parallel region. Only the exceptions need to be explicitly listed:
 - **#pragma omp for default(shared) private(a,b,c),** declares all variables to be shared, with the exception of a, b, and c.

- If **default(none)** is specified, the programmer is forced to specify a data-sharing attribute for each variable in the construct.
- Although variables with a predetermined data-sharing attribute need not be listed in one of the clauses
- It is recommended that the attribute be explicitly specified for *all* variables in the construct.

```
int main()
{
    int a = 5;
    #pragma omp parallel num_threads(5) default(none) shared(a)
    {
        int b = 10;
        a = a + b;
        printf("Parallel region:a= %d \n", a);
    }

    printf("Outside parallel region: %d\n", a);
    return 0;
}
```

Parallel region:a= 15

Parallel region:a= 25

Parallel region:a= 45

Parallel region:a= 35

Parallel region:a= 55

Outside parallel region: 55

D:\DSCA\Parallel_Computing\Lab_Programs\Example_Data
ith code 0.

To automatically close the console when debugging st
le when debugging stops.

Press any key to close this window . . .

Nowait Clause

- The nowait clause allows the programmer to fine-tune a program's performance.
- In the work-sharing constructs, there is an implicit barrier at the end of them. This clause overrides that feature of OpenMP;
- That is, if it is added to a construct, the barrier at the end of the associated construct will be suppressed.
- When threads reach the end of the construct, they will immediately proceed to perform other work.
- However, the barrier at the end of a parallel region cannot be suppressed.

- When a thread is finished with the work associated with the parallelized for loop, it continues and no longer waits for the other threads to finish as well.
- The clause ensures that there is no barrier at the end of the loop.

```
#pragma omp for nowait
for (i=0; i<n; i++)
{
    .....
}
```

```
int main()
{
#pragma omp parallel
{
#pragma omp for
    for (int i = 0; i < 5; i++)
    {
        printf("first loop i= %d\n", i);

        printf("outside\n");
    }

    return 0;
}
```

```
first loop i= 2
first loop i= 3
first loop i= 0
first loop i= 4
first loop i= 1
outside
outside
outside
outside
outside
outside
outside
outside
```

```
D:\DSCA\Parallel_Computing\Lab_Programs\Nowait_trial\x64\Debug\Nowait
To automatically close the console when debugging stops, enable To
le when debugging stops.
Press any key to close this window . . .
```

```
int main()
{
#pragma omp parallel
{
#pragma omp for nowait
    for (int i = 0; i < 5; i++)
    {
        printf("first loop i= %d\n", i);

        printf("outside\n");
    }

    return 0;
}
```

```
outside
first loop i= 3
outside
first loop i= 0
outside
first loop i= 2
outside
outside
first loop i= 4
outside
outside
outside
```

```
D:\DSCA\Parallel_Computing\Lab_Programs\Nowait_trial\x64\Debug\Nowait
To automatically close the console when debugging stops, enable To
le when debugging stops.
Press any key to close this window . . .
```

Schedule Clause

- The schedule clause is supported on the loop construct only.
- It is used to control the manner in which loop iterations are distributed over the threads, which can have a major impact on the performance of a program.
- The syntax is **schedule(*kind*[,*chunk_size*])**
- The schedule clause specifies how the iterations of the loop are assigned to the threads in the team.
- The granularity of this workload distribution is a **chunk**, a contiguous, nonempty subset of the iteration space.
- The *chunk_size* parameter need not be a constant; any loop invariant integer expression with a positive value is allowed

Kinds of Schedule

- **Static :**

- Iterations are divided into chunks of size ***chunk size***.
- The chunks are assigned to the threads statically in a **round-robin manner**, in the **order of the thread number**.
- The last chunk to be assigned may have a smaller number of iterations.
- When no ***chunk size*** is specified, the iteration space is divided into chunks that are approximately equal in size.
- Each thread is assigned at most one chunk.

- **Dynamic:**

- The iterations are assigned to threads as the threads request them.
- The thread executes the chunk of iterations (controlled through the ***chunk size*** parameter), then requests another chunk until there are no more chunks to work on.
- The last chunk may have fewer iterations than ***chunk size***.
- When no ***chunk size*** is specified, it defaults to 1.

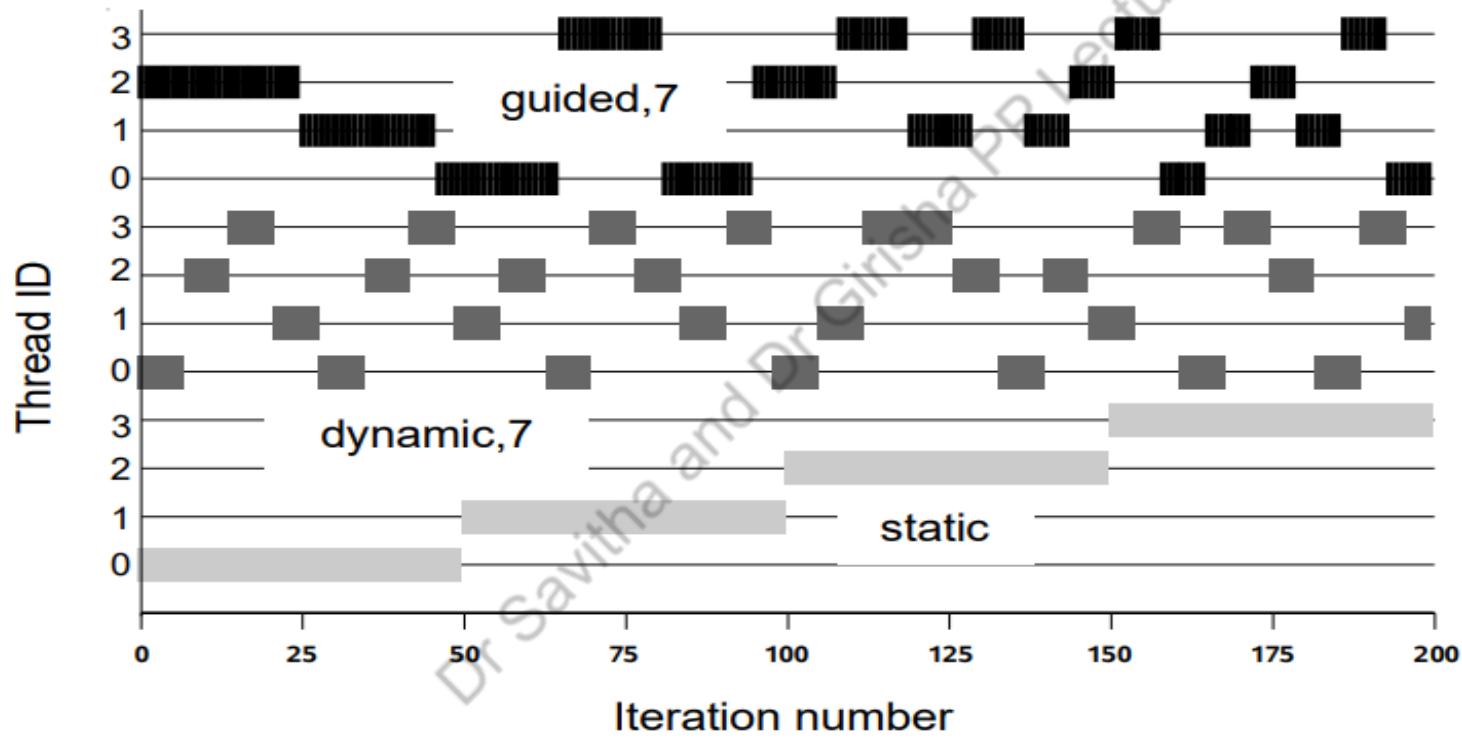
- **Guided:**

- The iterations are assigned to threads as the threads request them.
- The thread executes the chunk of iterations (controlled through the *chunk size* parameter), then requests another chunk until there are no more chunks to work on.
- For a *chunk size* of 1, the size of each chunk is proportional to the **number of unassigned iterations**, divided by **the number of threads**, decreasing to 1.
- For a *chunk size* of “ k ” ($k > 1$), the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than k iterations
- When no *chunk size* is specified, it defaults to 1.

- **Runtime:**

- If this schedule is selected, the decision regarding scheduling kind is made at run time.
- The schedule and (optional) chunk size are set through the OMP_SCHEDULE environment variable.

OpenMP: Schedule Clause



OpenMP: Schedule Clause

```
#pragma omp parallel for schedule(static,20) num_threads(5)
| for (int i = 0; i < 100; i++)
{
    printf("Loop executed by thread id=%d\n", omp_get_thread_num());
}

#pragma omp parallel for schedule(dynamic,20) num_threads(5)
| for (int i = 0; i < 100; i++)
{
    printf("Loop executed by thread id=%d\n", omp_get_thread_num());
}

#pragma omp parallel for schedule(runtime) num_threads(5)
| for (int i = 0; i < 100; i++)
{
    printf("Loop executed by thread id=%d\n", omp_get_thread_num());
```

All three workload distribution algorithms support an optional *chunk size* parameter.

- For example, a *chunk size* bigger than 1 on the static schedule may give rise to a round-robin allocation scheme in which each thread executes the iterations in a sequence of chunks whose size is given by *chunk size*.
- **It is not always easy to select the appropriate schedule and value for *chunk size* up front.**
- The choice may depend (among other things) not only on the code in the loop but also on the specific problem size and the number of threads used.
- **Therefore, the runtime clause is convenient.**
- Instead of making a compile time decision, the OpenMP OMP_SCHEDULE environment variable can be used to choose the schedule and (optional) *chunk size* at run time

Example for schedule clause

- The outer loop has been parallelized with the loop construct.
- The workload in the inner loop depends on the value of the outer loop iteration variable i.
- Therefore, the workload is not balanced, and the static schedule is probably not the best choice.

```
#pragma omp parallel for default(none) schedule(runtime) \
    private(i,j) shared(n)
    for (i=0; i<n; i++)
    {
        printf("Iteration %d executed by thread %d\n",
               i, omp_get_thread_num());
        for (j=0; j<i; j++)
            system("sleep 1");
    } /*-- End of parallel for --*/
```

OpenMP Synchronization Constructs

- Help to organize accesses to shared data by multiple threads.
- An algorithm may require us to orchestrate the actions of multiple threads to ensure that updates to a shared variable occur in a certain order, or it may simply need to ensure that two threads do not simultaneously attempt to write a shared object.
- These features can be used when the implicit barrier provided with work-sharing constructs does not suffice to specify the required interactions or would be inefficient.
- Together with the work-sharing constructs, they constitute a powerful set of features that suffice to parallelize a large number of applications.

Types of Constructs:

- **Barrier Construct**
- **Ordered Construct**
- **Critical Construct**
- **Atomic Construct**
- **Locks**
- **Master Construct**

Barrier Construct

- A barrier is a point in the execution of a program where threads wait for each other:
 - no thread in the team of threads it applies to, may proceed beyond a barrier until all threads in the team have reached that point.
- Compiler automatically inserts a barrier at the end of the construct, so that all threads wait there until all of the work associated with the construct has been completed.
- Thus, it is often **not necessary** for the programmer to explicitly add a barrier to a code.

`#pragma omp barrier`

- Two important restrictions apply to the barrier construct:
 - Each barrier **must** be encountered by all threads in a team, or by none at all.
 - The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in the team.

- Without these restrictions, one could write programs where some threads wait forever (or until somebody kills the process) for other threads to reach a barrier.
- C/C++ imposes an additional restriction regarding the placement of a barrier construct within the application
 - The barrier construct may only be placed in the program at a position where ignoring or deleting it would result in a program with correct syntax.
- The most common use for a barrier is to avoid a **data race condition**
 - Inserting a barrier between the **writes to** and **reads from a shared variable** guarantees that the accesses are appropriately ordered

```

#pragma omp parallel private(TID)
{
    TID = omp_get_thread_num();
    if (TID < omp_get_num_threads()/2 ) system("sleep 3");
    (void) print_time(TID,"before");

#pragma omp barrier

    (void) print_time(TID,"after ");
} /*-- End of parallel region --*/

```

- To ensure that some threads in the team executing the parallel region take longer than others to reach the barrier, we get half the threads to execute the sleep 3 command, causing them to idle for three seconds.
- We then get each thread to print out its the thread number (stored in variable TID), a comment string, and the time of day in the format hh:mm:ss.
- The barrier is then reached.
- After the barrier, each thread will resume execution and again print out this information.
- Four threads are used. Note that threads 2 and 3 wait for three seconds in the barrier.

```

Thread 2 before barrier at 01:12:05
Thread 3 before barrier at 01:12:05
Thread 1 before barrier at 01:12:08
Thread 0 before barrier at 01:12:08
Thread 1 after barrier at 01:12:08
Thread 3 after barrier at 01:12:08
Thread 2 after barrier at 01:12:08
Thread 0 after barrier at 01:12:08

```

Ordered Construct

- Another synchronization construct, the ordered construct, allows one to execute a structured block within a parallel loop in sequential order.
- This is used to enforce an ordering on the printing of data computed by different threads.
- It may also be used to help determine whether there are any data races in the associated code.
- The syntax of the ordered construct

```
#pragma omp ordered  
structured block
```

- An ordered construct ensures that the code within the associated structured block is executed in sequential order.
- The code outside this block runs in parallel. When the thread executing the first iteration of the loop encounters the construct, it enters the region without waiting.
- When a thread executing any subsequent iteration encounters the construct, it waits until each of the previous iterations in the sequence has completed execution of the region

Synchronization Constructs: Ordered

```
//Ordered Clause
int n = 8;
int a[8] = {};
#pragma omp parallel for default(none) ordered schedule(runtime) shared(n,a)
for (int i = 0; i < n; i++)
{
    int TID = omp_get_thread_num();
    printf("Thread %d updates a[%d]\n", TID, i);
    a[i] += i;
    #pragma omp ordered
    {
        printf("Thread %d prints value of a[%d] = %d\n", TID, i, a[i]);
    }
}
```

```
Thread 3 updates a[3]
Thread 6 updates a[6]
Thread 7 updates a[7]
Thread 0 updates a[0]
Thread 0 prints value of a[0] = 0
Thread 1 updates a[1]
Thread 1 prints value of a[1] = 1
Thread 5 updates a[5]
Thread 4 updates a[4]
Thread 2 updates a[2]
Thread 2 prints value of a[2] = 2
Thread 3 prints value of a[3] = 3
Thread 4 prints value of a[4] = 4
Thread 5 prints value of a[5] = 5
Thread 6 prints value of a[6] = 6
Thread 7 prints value of a[7] = 7
```

```
D:\DSCA\Parallel_Computing\Lab_Programs\Barr...
.
To automatically close the console when debu...
le when debugging stops.
Press any key to close this window . . .
```

Critical Construct

- The critical construct provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously. The associated code is referred to as a critical region, or a *critical section*.
- An optional *name* can be given to a critical construct. In contrast to the rules governing other language features, this name is *global* and therefore should be unique.
- When a thread encounters a critical construct, it waits until no other thread is executing a critical region with the same name.
- In other words, there is never a risk that multiple threads will execute the code contained in the same critical region at the same time.

```
#pragma omp critical [(name)]  
    structured block
```

Synchronization Constructs: Critical

```
int sum = 1;
int n = 8;
int a[8] = {};
#pragma omp parallel shared(n,a,sum)
{
    int TID = omp_get_thread_num();
    int sumLocal = 1;
    #pragma omp for
    for (int i = 0; i < n; i++)
        sumLocal += a[i];
    #pragma omp critical
    {
        sum += sumLocal;
        printf("TID=%d: sumLocal=%d sum = %d\n", TID, sumLocal, sum);
    }
}
printf("Value of sum after parallel region: %d\n", sum);
```

```
TID=5: sumLocal=1 sum = 2
TID=7: sumLocal=1 sum = 3
TID=6: sumLocal=1 sum = 4
TID=3: sumLocal=1 sum = 5
TID=2: sumLocal=1 sum = 6
TID=4: sumLocal=1 sum = 7
TID=1: sumLocal=1 sum = 8
TID=0: sumLocal=1 sum = 9
Value of sum after parallel region: 9

D:\DSCA\Parallel_Computing\Lab_Programs\Barrier_example
To automatically close the console when debugging stops
le when debugging stops.
Press any key to close this window . . .
```

Atomic Construct

- The atomic construct enables efficient updating of shared variables by multiple threads on hardware platforms which support *atomic* operations.
- The reason it is applied to just one assignment statement is that it protects updates to an individual memory location, the one on the left-hand side of the assignment.
- If a thread is atomically updating a value, then no other thread may do so simultaneously.
- This restriction applies to all threads that execute a program, not just the threads in the same team.
- Syntax is:

```
#pragma omp atomic  
    statement
```

- The atomic construct may only be used together with an expression statement in C/C++. The supported operations are: +, *, -, /, &, ^, |, <<, >>

```
int ic, i, n;  
ic = 0;  
#pragma omp parallel shared(n,ic) private(i)  
for (i=0; i++, i<n)  
{  
    #pragma omp atomic  
    ic = ic + 1;  
}  
printf("counter = %d\n", ic);
```

The atomic construct ensures that no updates are lost when multiple threads are updating a counter value.

```
int atomic_read(const int* x)  
{  
    int value;  
    #pragma omp atomic read  
    value = *x;  
    return value;  
}  
void atomic_write(int* x, int value)  
{  
    #pragma omp atomic write  
    *x = value;  
}
```

Locks

- A set of low-level, general-purpose runtime library routines, similar in function to the use of semaphores. These routines provide greater flexibility for synchronization than does the use of critical sections or atomic constructs.
- A thread lock is an object that can be held by at most one thread at a time
- An OpenMP lock can be in one of the following states:
- Uninitialized; Unlocked; or Locked
- If a lock is in the unlocked state, a thread can set the lock, which changes its state to locked
- The thread that sets the lock is then said to own the lock
- A thread that owns a lock can unset that lock, returning it to the unlocked state
- Syntax is : **void omp_func_lock (omp lock t *lck)**

- There are two types of locks:
- ***Simple locks***,
 - which may not be locked if already in a locked state
 - Simple lock variables are declared with the special type `omp_lock_t` in C/C++
- ***Nestable locks***,
 - Which may be locked multiple times by the same thread
 - Nestable lock variables are declared with the special type `omp_nest_lock_t` in C/C++

- The general procedure to use locks is as follows:
 - Define the lock variables using **omp_lock_t**
 - Initialize the lock via a call to **omp_init_lock()**
 - Set the lock using **omp_set_lock()** or **omp_test_lock()**. The latter checks whether the lock is actually available before attempting to set it. It is useful to achieve asynchronous thread execution
 - Unset a lock after the work is done via a call to **omp_unset_lock()**
 - Remove the lock association via a call to **omp_destroy_lock()**

Synchronization Constructs: Locks

```
omp_lock_t A;  
  
omp_init_lock(&A);  
int b=1;  
int c=0;  
int d=0;  
#pragma omp parallel for  
for (int i = 0; i < 10; i++)  
{  
    // some stuff  
    d=d+b;  
    printf("##### thread...%d...d=%d\n", omp_get_thread_num(),d);  
    omp_set_lock(&A);  
    c=c+b;  
    printf("Executed by thread...%d...c=%d\n", omp_get_thread_num(),c);  
    omp_unset_lock(&A);  
    // some stuff  
}  
  
omp_destroy_lock(&A);
```

```
##### thread...2...d=1  
##### thread...4...d=2  
##### thread...3...d=3  
##### thread...7...d=5  
##### thread...0...d=5  
##### thread...6...d=7  
##### thread...1...d=8  
##### thread...5...d=6  
Executed by thread...2...c=1  
Executed by thread...4...c=2  
Executed by thread...3...c=3  
Executed by thread...7...c=4  
Executed by thread...0...c=5  
##### thread...0...d=9  
Executed by thread...6...c=6  
Executed by thread...1...c=7  
##### thread...1...d=10  
Executed by thread...5...c=8  
Executed by thread...0...c=9  
Executed by thread...1...c=10  
  
-----  
Process exited after 0.4338 seconds with return value 0  
Press any key to continue . . .
```

Master Construct

- The master construct defines a block of code that is guaranteed to be executed by the master thread only
- It is thus similar to the single construct
- The master construct is technically not a work-sharing construct, however, and it does not have an implied barrier on entry or exit
- If the master construct is used to initialize data, for example, care needs to be taken that this initialization is completed before the other threads in the team use the data
- Syntax is: **#pragma omp master**
 structured block

Synchronization Constructs: Master

```
int a=0;
int b[10];
int i=0, n=10;
#pragma omp parallel shared(a,b) private(i)
{
    #pragma omp master
    {
        a = 10;
        printf("Master construct is executed by thread %d\n",
               omp_get_thread_num());
    }
    #pragma omp barrier
    #pragma omp for
    for (i=0; i<n; i++)
        b[i] = a;
} /*-- End of parallel region --*/

printf("After the parallel region:\n");
for (i=0; i<n; i++)
    printf("b[%d] = %d\n", i, b[i]);
```

Master construct is executed by thread 0
After the parallel region:

```
b[0] = 10
b[1] = 10
b[2] = 10
b[3] = 10
b[4] = 10
b[5] = 10
b[6] = 10
b[7] = 10
b[8] = 10
b[9] = 10
```

Process exited after 0.1579 seconds with return value 0
Press any key to continue . . .

OpenMP: Other Clauses

If Clause

- The **if clause** is supported on the parallel construct only, where it is used to specify conditional execution
- Since some overheads are inevitably incurred with the creation and termination of a parallel region, it is sometimes necessary to test whether there is enough work in the region to warrant its parallelization

if(scalar-logical-expression)

- If the logical expression evaluates to true, which means it is of type integer and has a non-zero value in C/C++, the parallel region will be executed by a team of threads
- If it evaluates to false, the region is executed by a single thread only

OpenMP: If Clause

```
int n=5;
int TID=0;
#pragma omp parallel if (n > 5) default(none) \
private(TID) shared(n)
{
    TID = omp_get_thread_num();
    #pragma omp single
    {
        printf("Value of n = %d\n",n);
        printf("Number of threads in parallel region: %d\n",
        omp_get_num_threads());
    }
    printf("Print statement executed by thread %d\n",TID);
} /*-- End of parallel region --/
```

```
Value of n = 5
Number of threads in parallel region: 1
Print statement executed by thread 0

-----
Process exited after 0.2263 seconds with return value 0
Press any key to continue . . .
```

```
int n=6;
int TID=0;
#pragma omp parallel if (n > 5) default(none) \
private(TID) shared(n)
{
    TID = omp_get_thread_num();
    #pragma omp single
    {
        printf("Value of n = %d\n",n);
        printf("Number of threads in parallel region: %d\n",
        omp_get_num_threads());
    }
    printf("Print statement executed by thread %d\n",TID);
} /*-- End of parallel region --/
```

```
Value of n = 6
Number of threads in parallel region: 8
Print statement executed by thread 2
Print statement executed by thread 6
Print statement executed by thread 0
Print statement executed by thread 4
Print statement executed by thread 1
Print statement executed by thread 5
Print statement executed by thread 7
Print statement executed by thread 3

-----
Process exited after 0.2341 seconds with return value 0
Press any key to continue . . .
```

Num_threads Clause

- The num threads clause is supported on the parallel construct only
- Can be used to specify how many threads should be in the team executing the parallel region
- Has higher priority over **omp_set_num_threads(4)**

Reduction Clause

- OpenMP provides the reduction clause for specifying some forms of recurrence calculations
 - They can be performed in parallel without code modification
- The programmer must identify the operations and the variables that will hold the result values

reduction(operator :list)

- The order in which thread-specific values are combined is unspecified
 - floating-point data are concerned, there may be numerical differences between the results of a sequential and parallel run

```

int a=10;
int i=0;
int sum=0;
#pragma omp parallel private(i) num_threads(6)
{
    #pragma omp for
    for(i=0;i<6;i++)
    {
        sum =sum + a;
        printf("Sum value=%d ID=%d\n",sum,omp_get_thread_num());
    }
    printf("Outside Sum value=%d ID=%d\n",sum,omp_get_thread_num());
}

```

```

Sum value=10 ID=0
Sum value=20 ID=1
Sum value=20 ID=3
Sum value=30 ID=2
Sum value=40 ID=5
Sum value=50 ID=4
Outside Sum value=50 ID=0

```

Process exited after 0.4014 seconds with return value 0
Press any key to continue . . .

```

int a=10;
int i=0;
int sum=0;
#pragma omp parallel private(i) num_threads(6)
{
    #pragma omp for reduction(:sum)
    for(i=0;i<6;i++)
    {
        sum =sum + a;
        printf("Sum value=%d ID=%d\n",sum,omp_get_thread_num());
    }
    printf("Outside Sum value=%d ID=%d\n",sum,omp_get_thread_num());
}

```

```

Sum value=10 ID=3
Sum value=10 ID=4
Sum value=10 ID=0
Sum value=10 ID=1
Sum value=10 ID=2
Sum value=10 ID=5
Outside Sum value=60 ID=0

```

Process exited after 0.4353 seconds with return value 0
Press any key to continue . . .

Operator	Initialization value
<code>+</code>	<code>0</code>
<code>*</code>	<code>1</code>
<code>-</code>	<code>0</code>
<code>&</code>	<code>~0</code>
<code> </code>	<code>0</code>
<code>~</code>	<code>0</code>
<code>&&</code>	<code>1</code>
<code> </code>	<code>0</code>

- Aggregate types (including arrays), pointer types, and reference types are not supported.
- A reduction variable must not be const-qualified.
- The operator specified on the clause can not be overloaded with respect to the variables that appear in the clause

Copyin Clause

- Allows us to copy the value of the master thread's `threadprivate` variable(s) to the corresponding `threadprivate` variables of the other threads
- `Threadprivate`:
 - Static variables are generally shared by default
 - We can change this by using **threadprivate** clause (We will discuss this later)
- The initial values of private variables are undefined
- The copy is carried out after the team of threads is formed and prior to the start of execution of the parallel region, so that it enables a straightforward initialization of this kind of data object.

Copyprivate Clause

- The **copyprivate clause** is supported on the **single directive** only
- It provides a mechanism for broadcasting the value of a private variable from one thread to the other threads in the team
- **Uses:** one thread read or initialize private data that is subsequently used by the other threads as well
- After the single construct has ended, but before the threads have left the associated barrier, the values of variables specified in the associated list are copied to the other threads
- Since the barrier is essential in this case, the standard prohibits use of this clause in combination with the **nowait clause**

```
int main()
{
    int a=10;

    #pragma omp parallel private(a)
    {
        #pragma omp single
        {
            a=a+5;
        }
        printf("Id--%d    =%d\n",omp_get_thread_num(),a);
    }
    return 0;
}
```

```
int main()
{
    int a=10;

    #pragma omp parallel private(a)
    {
        #pragma omp single copyprivate(a)
        {
            a=a+5;
        }
        printf("Id--%d    =%d\n",omp_get_thread_num(),a);
    }
    return 0;
}
```

```

int main()
{
    int a=10;

#pragma omp parallel private(a)
{
    #pragma omp single
    {
        a=a+5;
    }
    printf("Id--%d    =%d\n",omp_get_thread_num(),a);

}
return 0;
}

```

```

Id--1    =5
Id--2    =0
Id--4    =0
Id--3    =0
Id--5    =0
Id--0    =0
Id--7    =0
Id--6    =0

```

Process exited after 0.1778 seconds with return value 0
Press any key to continue . . .

```

int main()
{
    int a=10;

#pragma omp parallel private(a)
{
    #pragma omp single copyprivate(a)
    {
        a=a+5;
    }
    printf("Id--%d    =%d\n",omp_get_thread_num(),a);

}
return 0;
}

```

```

Id--4    =5
Id--3    =5
Id--6    =5
Id--5    =5
Id--0    =5
Id--7    =5
Id--1    =5
Id--2    =5

```

Process exited after 0.1717 seconds with return value 0
Press any key to continue . . .

Nested parallelism

- If a thread in a team executing a parallel region encounters another parallel construct, it creates a new team and becomes the master of that new team
 - This is generally referred to in OpenMP as “**nested parallelism**”
- If nested parallelism is not supported
 - parallel constructs that are nested within other parallel constructs will be ignored
 - parallel region serialized (executed by a single thread only)
- **Frequent starting and stopping of parallel regions may introduce a non-trivial performance penalty**
- What will happen if we call `omp_get_thread_num()` function from the nested region?
 - It returns the thread id starting from 0 to one less than the number of threads in the current thread team
 - Thread numbers are no longer unique

Nested parallelism

```
omp_set_nested(1);
printf("Nested parallelism is %s\n",omp_get_nested() ? "supported" : "not supported");
#pragma omp parallel
{
    printf("Thread %d executes the outer parallel region\n",
    omp_get_thread_num());
    #pragma omp parallel num_threads(2)
    {
        printf(" Thread %d executes inner parallel region\n",
        omp_get_thread_num());
    } /*-- End of inner parallel region --*/
} /*-- End of outer parallel region --*/
```

Nested parallelism is supported
Thread 4 executes the outer parallel region
Thread 7 executes the outer parallel region
Thread 2 executes the outer parallel region
Thread 6 executes the outer parallel region
Thread 3 executes the outer parallel region
Thread 0 executes the outer parallel region
Thread 1 executes the outer parallel region
Thread 5 executes the outer parallel region
Thread 0 executes inner parallel region
Thread 1 executes inner parallel region
Thread 0 executes inner parallel region
Thread 1 executes inner parallel region
Thread 1 executes inner parallel region
Thread 0 executes inner parallel region
Thread 0 executes inner parallel region
Thread 1 executes inner parallel region
Thread 1 executes inner parallel region
Thread 0 executes inner parallel region
Thread 0 executes inner parallel region
Thread 1 executes inner parallel region
Thread 0 executes inner parallel region
Thread 1 executes inner parallel region
Thread 0 executes inner parallel region
Thread 1 executes inner parallel region
Thread 0 executes inner parallel region
Thread 1 executes inner parallel region

Nested parallelism

```
omp_set_nested(1);
printf("Nested parallelism is %s\n",omp_get_nested() ? "supported" : "not supported");
int TID=0;
#pragma omp parallel private(TID)
{
    TID = omp_get_thread_num();
    printf("Thread %d executes the outer parallel region\n",TID);
    #pragma omp parallel num_threads(2) firstprivate(TID)
    {
        printf("TID %d: Thread %d executes inner parallel region\n",
               TID,omp_get_thread_num());
    } /*-- End of inner parallel region --*/
} /*-- End of outer parallel region --*/
```

```
Nested parallelism is supported
Thread 5 executes the outer parallel region
Thread 1 executes the outer parallel region
Thread 6 executes the outer parallel region
Thread 4 executes the outer parallel region
Thread 7 executes the outer parallel region
Thread 3 executes the outer parallel region
Thread 0 executes the outer parallel region
TID 4: Thread 1 executes inner parallel region
TID 5: Thread 0 executes inner parallel region
TID 5: Thread 1 executes inner parallel region
TID 1: Thread 0 executes inner parallel region
TID 1: Thread 1 executes inner parallel region
TID 6: Thread 0 executes inner parallel region
TID 6: Thread 1 executes inner parallel region
TID 4: Thread 0 executes inner parallel region
Thread 2 executes the outer parallel region
TID 7: Thread 0 executes inner parallel region
TID 2: Thread 0 executes inner parallel region
TID 3: Thread 0 executes inner parallel region
TID 3: Thread 1 executes inner parallel region
TID 0: Thread 0 executes inner parallel region
TID 0: Thread 1 executes inner parallel region
TID 7: Thread 1 executes inner parallel region
TID 2: Thread 1 executes inner parallel region
```

OpenMP: Flush Directive

- OpenMP memory model distinguishes between shared data and private data:
 - Which is accessible and visible to all threads (**shared data**)
 - Which is local to an individual thread (**private data**)
- If a thread updates shared data, the new values will first be saved in a **register** and then stored back to the **local cache**
 - **Other threads doesn't have access to these memories immediately**
- **Cache-Coherent machines:** Broadcasts these shared variables to other threads
- The OpenMP standard specifies that all modifications are written back to main memory
- Modifications are thus available to all threads, at synchronization points in the program

OpenMP: Flush Directive

- Between these synchronization points, threads are permitted to have new values for shared variables stored in their local memory rather than in the global shared memory
- This approach is called as **relaxed consistency model**
- Sometimes updated values of shared values must become visible to other threads in-between synchronization points
- The OpenMP API provides the **flush directive** to make this possible
- The purpose of the flush directive is to make a thread's temporary view of shared data consistent with the values in memory

```
#pragma omp flush [(list)]
```

OpenMP: Flush Directive

- The flush operation applies to all variables specified in the list
- If no list is provided, it applies to all thread-visible shared data
- If the flush operation is invoked by a thread that has updated the variables, their new values will be flushed to memory and therefore be accessible to all other threads
- If the construct is invoked by a thread that has not updated a value, it will ensure that any local copies of the data are replaced by the latest value from main memory
- This does not synchronize the actions of different threads: rather, it forces the executing thread to make its shared data values consistent **with shared memory**
- Since the compiler reorders operations to enhance program performance, one cannot assume that the flush operation will remain exactly in the position, relative to other operations, in which it was placed by the programmer

OpenMP: Flush Directive

- Implicit flush operations with no list occur at the following locations
 - All explicit and implicit barriers (e.g., at the end of a parallel region or worksharing construct)
 - Entry to and exit from critical regions
 - Entry to and exit from lock routines

Dr Savitha and Dr Girisha PP Lecture Slides

OpenMP: Flush Directive

```
#include<stdio.h>
#include<omp.h>
int main() {
    int data, flag = 0;
#pragma omp parallel num_threads(2)
{
    if (omp_get_thread_num()==0) {
        data = 42;
        #pragma omp flush(flag, data)
        /* Set flag to release thread 1 */
        flag = 1;
        #pragma omp flush(flag)
    }
    else if (omp_get_thread_num()==1) {
        #pragma omp flush(flag, data)
        while (flag < 1) {
            #pragma omp flush(flag, data)
        }
        #pragma omp flush(flag, data)
        printf("flag=%d data=%d\n", flag, data);
    }
}
return 0;
```

OpenMP: Threadprivate Directive

- By default, global data is shared
- Each thread gets a **private** or “local” copy of the specified global variables

```
#pragma omp threadprivate (list)
```

- By default, the threadprivate copies are not allocated or defined

OpenMP Code Structure

- “Pragma”: stands for “pragmatic information”
- A pragma is a way to communicate the information to the compiler
- The information is non-essential in the sense that the compiler may ignore the information and still produce correct object program.

OpenMP Core Syntax

```
#include "omp.h"
int main ()
{
    int var1, var2, var3;
    // Serial code ...
    // Beginning of parallel section.
    // Fork a team of threads. Specify variable scoping
    #pragma omp parallel private(var1, var2) shared(var3)
    {
        // Parallel section executed by all threads ...
        // All threads join master thread and disband
    }
    // Resume serial code...
}
```

Thread Creation: Parallel Region Example

```
#include <stdio.h>
#include "omp.h"
int main()
{
    int nthreads, tid;
    #pragma omp parallel num_threads(4) private(tid)
    {
        tid = omp_get_thread_num();
        printf("Hello world from (%d)\n", tid);
        if(tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("number of threads = %d\n", nthreads);
        }
    } // all threads join master thread and terminates
}
```

Thread Creation: Parallel Region Example

```
#include <stdio.h>
#include "omp.h"
int main()
{
    int nthreads, A[100] , tid;
    // fork a group of threads with each thread having a private tid variable
    omp_set_num_threads(4);
    #pragma omp parallel private (tid)
    {
        tid = omp_get_thread_num();
        foo(tid, A);
    }
    // all threads join master thread and terminates
}
```

OpenMP controlling number of threads

- Asking how many cores this program has access to:

```
num = omp_get_num_procs();
```

- Setting the number of available threads to the exact number of cores available:

```
omp_set_num_threads( omp_get_num_procs() );
```

- Asking how many OpenMP threads this program is using right now:

```
num = omp_get_num_threads();
```

- Asking which thread number this one is:

```
me = omp_get_thread_num();
```

END

Dr Savitha and Dr Gunisha PP Lecture Slides

CUDA

Dr Savitha & Dr Girish
BPLecture slides

Parallelism

- Writing a parallel program must always start by identifying the parallelism inherent in the algorithm at hand
- Different variants of parallelism induce different methods of parallelization

Dr Savitha & Dr Girish&PP Lecture slides

Parallelism

- Instruction-level parallelism (ILP) is the parallel or simultaneous execution of a sequence of instructions in a computer program.
- More specifically ILP refers to the average number of instructions run per step of this parallel execution.
- **Instruction-level parallelism (ILP)** is a measure of how many operations in a computer program can be performed "in-parallel" at the same time

Caution:

- Data dependency : Data dependence means that one instruction is dependent on another if there exists a chain of dependencies between them
- Name Dependency: A me dependency occurs when two instructions use the same register or memory location, called a name, but there is no flow of data between them.

Parallelism

- **Data Level parallelism**
 - Many problems in scientific computing involve processing of large quantities of data stored on a computer
 - If this manipulation can be performed in parallel, i.e., by multiple processors working on different parts of the data, we speak of **data parallelism**
 - This is the dominant parallelization concept in scientific computing on MIMD-type computers
 - The same code is executed on all processors, with independent instruction pointers

Parallelism

- **Functional/Task level parallelism**

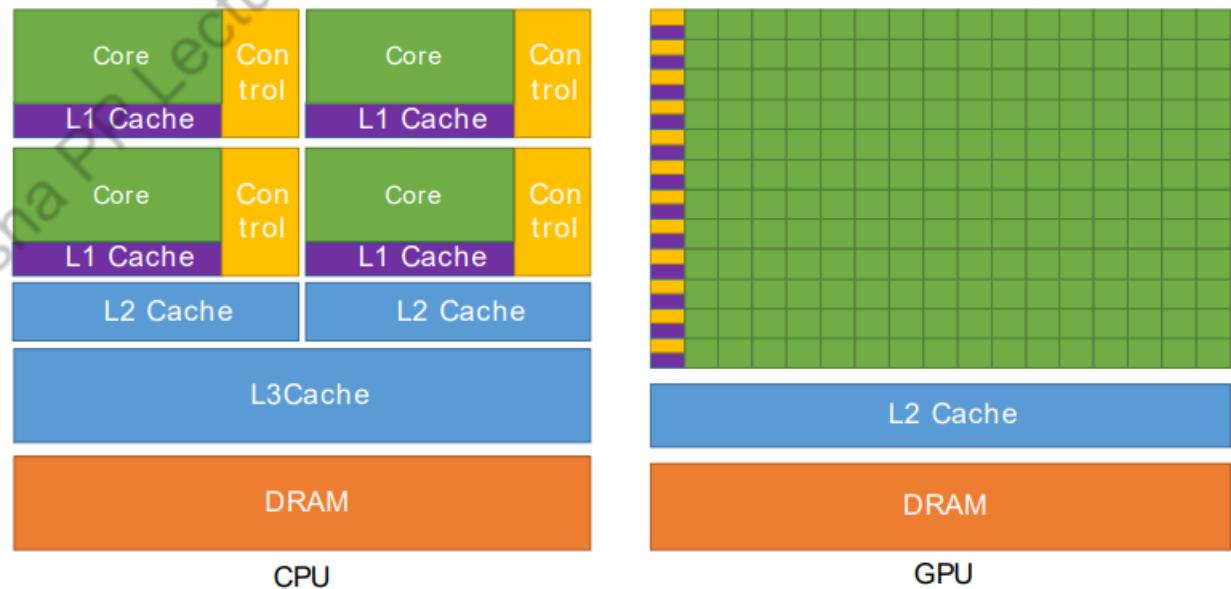
- Sometimes the solution of a “big” numerical problem can be split into separate subtasks
- Which work together by **data exchange** and **synchronization**
- In this case, the subtasks execute completely different code on different data items, which is why functional parallelism is also called MPMD
- Functional parallelism bears pros and cons:
 - When different parts of the problem have different performance properties and hardware requirements, **bottlenecks and load imbalance** can easily arise
 - On the other hand, overlapping tasks that would otherwise be executed sequentially could accelerate execution considerably

Introduction

- The **Graphics Processing Unit (GPU)** provides much higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope
- Many applications leverage these higher capabilities to run faster on the GPU than on the CPU
- This difference in capabilities between the **GPU** and the **CPU** exists because they are designed with different goals in mind
 - The CPU is designed to excel at executing a **sequence of operations**, called a **thread**, as fast as possible and can execute a few tens of these threads in parallel
 - The GPU is designed to excel at executing **thousands of them in parallel**
 - The GPU is specialized for highly parallel computations
 - Designed such that more transistors are devoted to data processing rather than data caching and flow control

Introduction

- Devoting more transistors to data processing is beneficial for high parallel computing
 - e.g: floating-point computations
- The GPU can hide memory access latencies with computation, instead of relying on large data caches and complex flow control
- Avoids long memory access latencies, both of which are expensive in terms of transistors



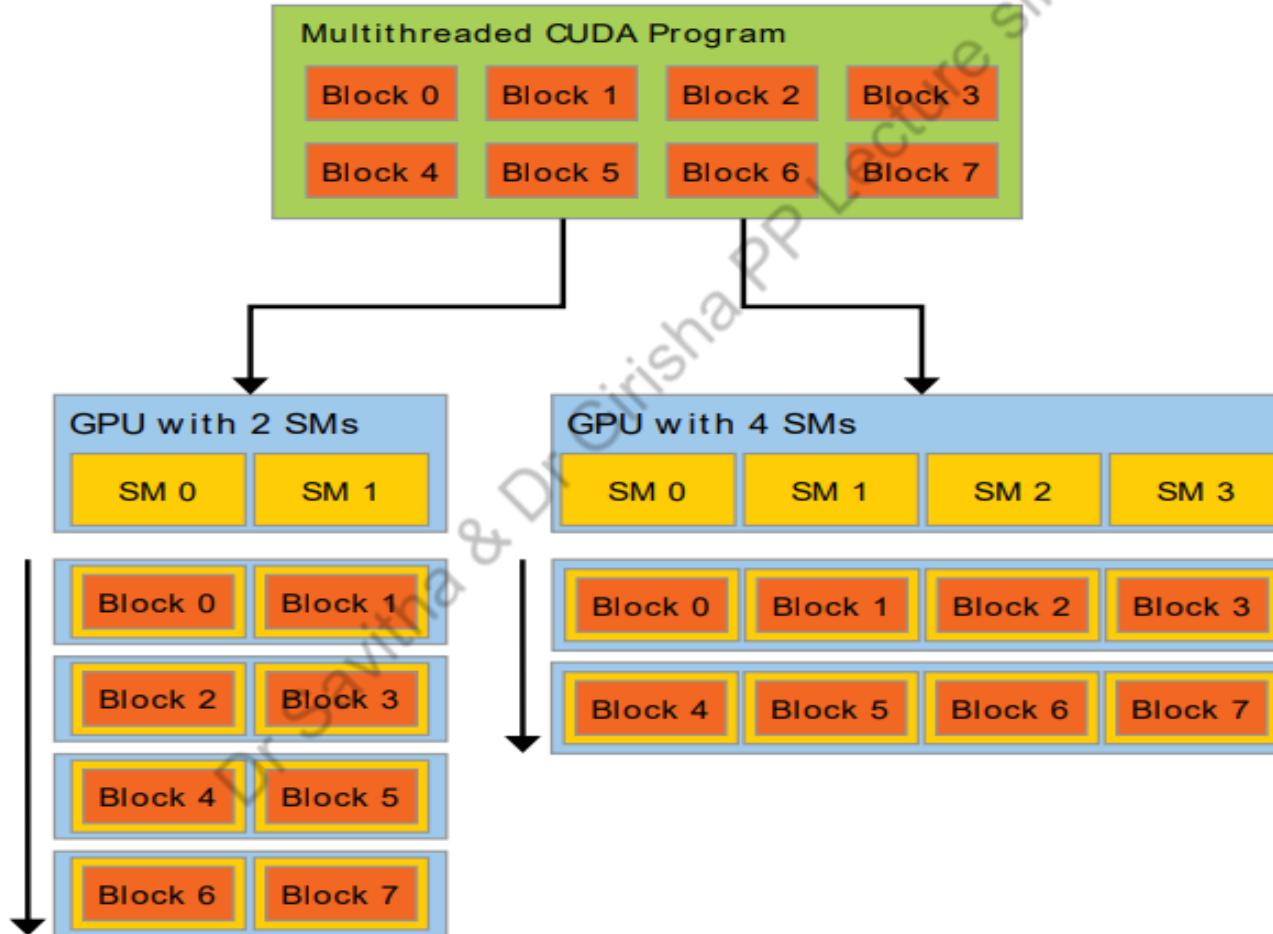
A Scalable Programming Model

- Mainstream processor chips are now parallel systems
 - Multicore CPUs and manycore GPUs
- The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores
- CUDA parallel programming at its core are three key abstractions:
 - A hierarchy of thread groups
 - Shared memories
 - Barrier synchronization
- That are simply exposed to the programmer as a minimal set of language extensions

A Scalable Programming Model

- They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads
- Each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block
- Each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors
- Only the runtime system needs to know the physical multiprocessor count

A Scalable Programming Model

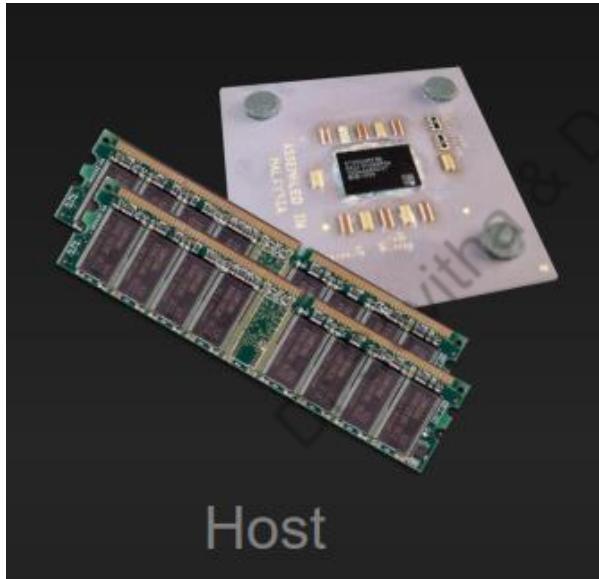


Development environment

- Every NVIDIA GPU since the 2006 release of the GeForce 8800 GTX has been CUDA-enabled
 - Has been built on the CUDA Architecture
- NVIDIA DEVICE DRIVER
 - NVIDIA provides system software that allows your programs to communicate with the CUDA-enabled hardware
- CUDA Development Toolkit
 - CUDA C applications are going to be computing on two different processors and we need two compilers
 - One compiler will compile code for your GPU, and one will compile code for your CPU
 - NVIDIA provides the compiler for your GPU code (CUDA Toolkit)

Heterogeneous Computing

- Terminology:
 - Host: The CPU and its memory (host memory)
 - Device: The GPU and its memory (device memory)



Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N      1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int index = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[index] = in[index];
    if (threadIdx.x < RADIUS) {
        temp[index - RADIUS] = in[gindex - RADIUS];
        temp[index + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[index + offset];

    // Store the result
    out[index] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out;           // host copies of a, b, c
    int *d_in, *d_out;       // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **) &d_in, size);
    cudaMalloc((void **) &d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<N(BLOCK_SIZE,BLOCK_SIZE>>(d_in + RADIUS, d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

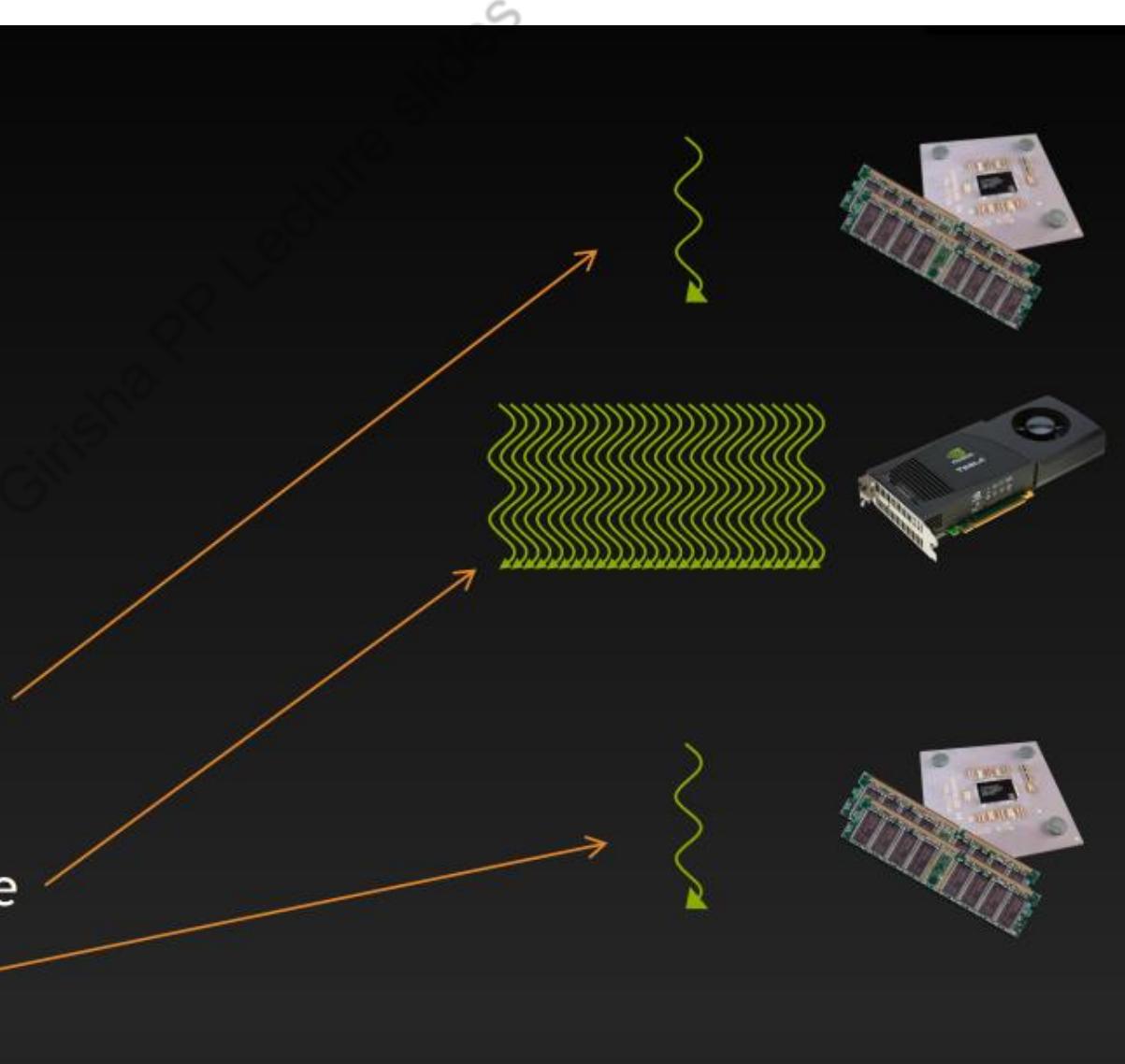
    // Cleanup
    free(in);
    free(out);
    cudaFree(d_in);
    cudaFree(d_out);
    return 0;
}
```

parallel fn

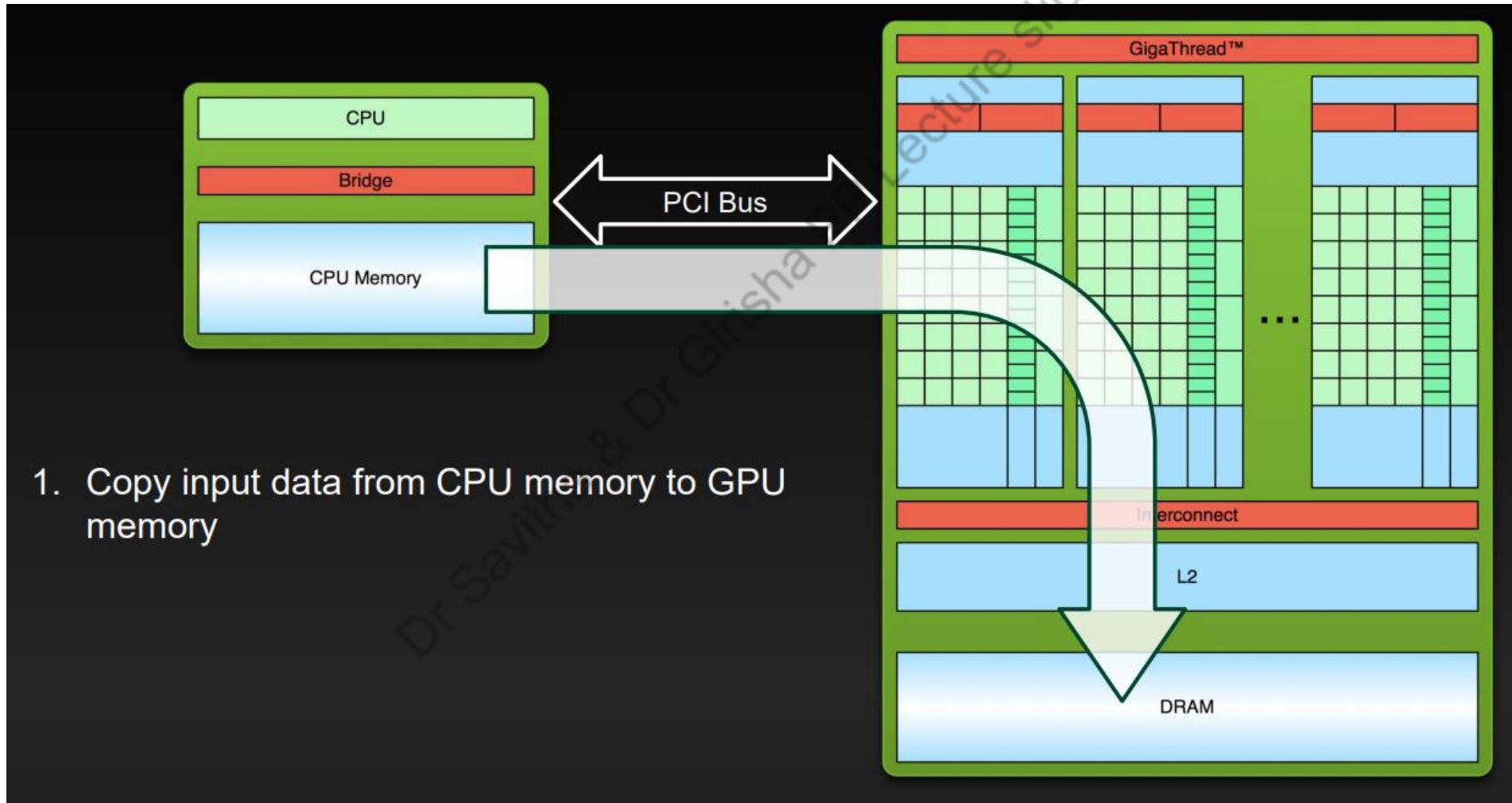
serial code

parallel code

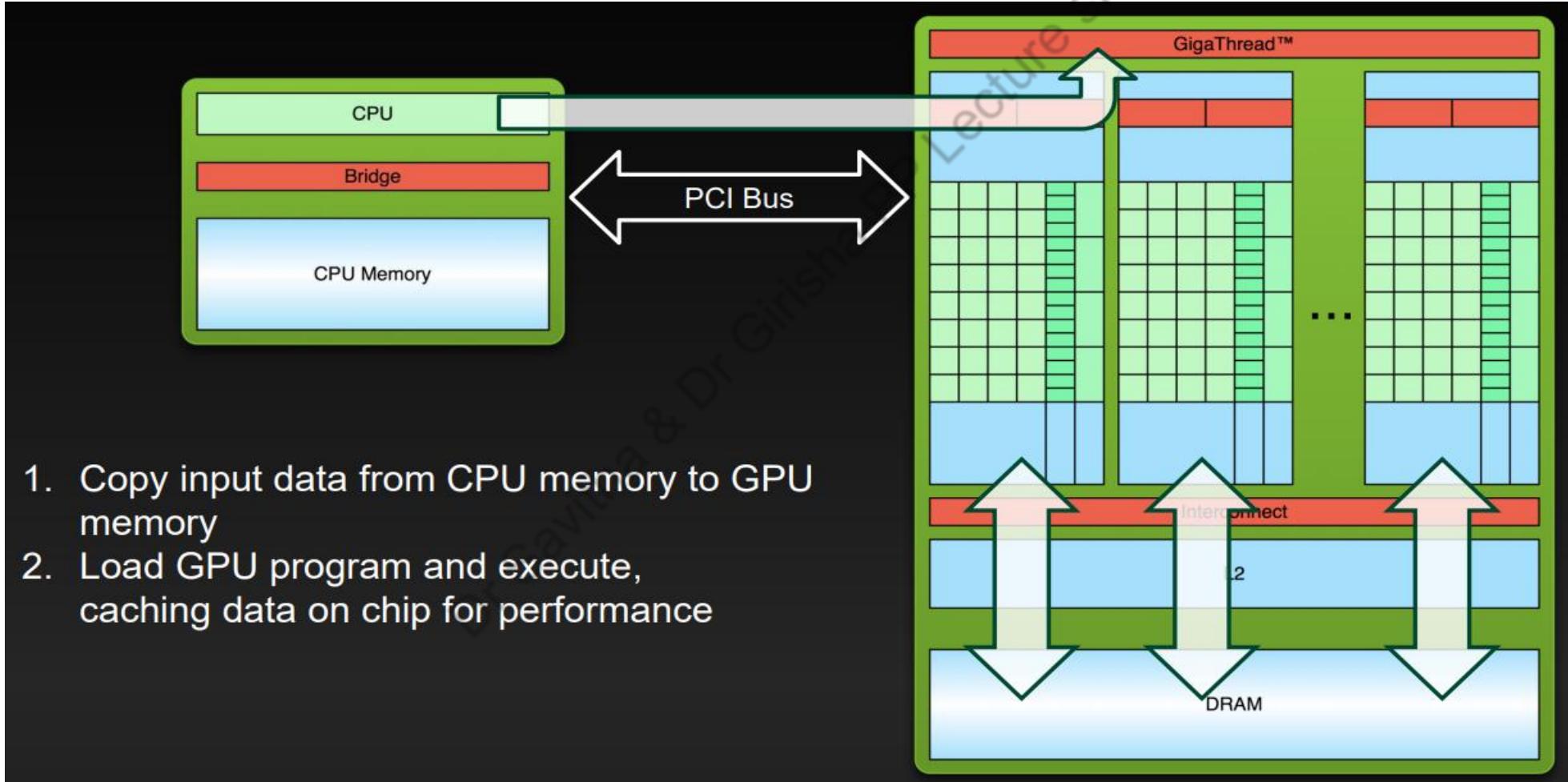
serial code



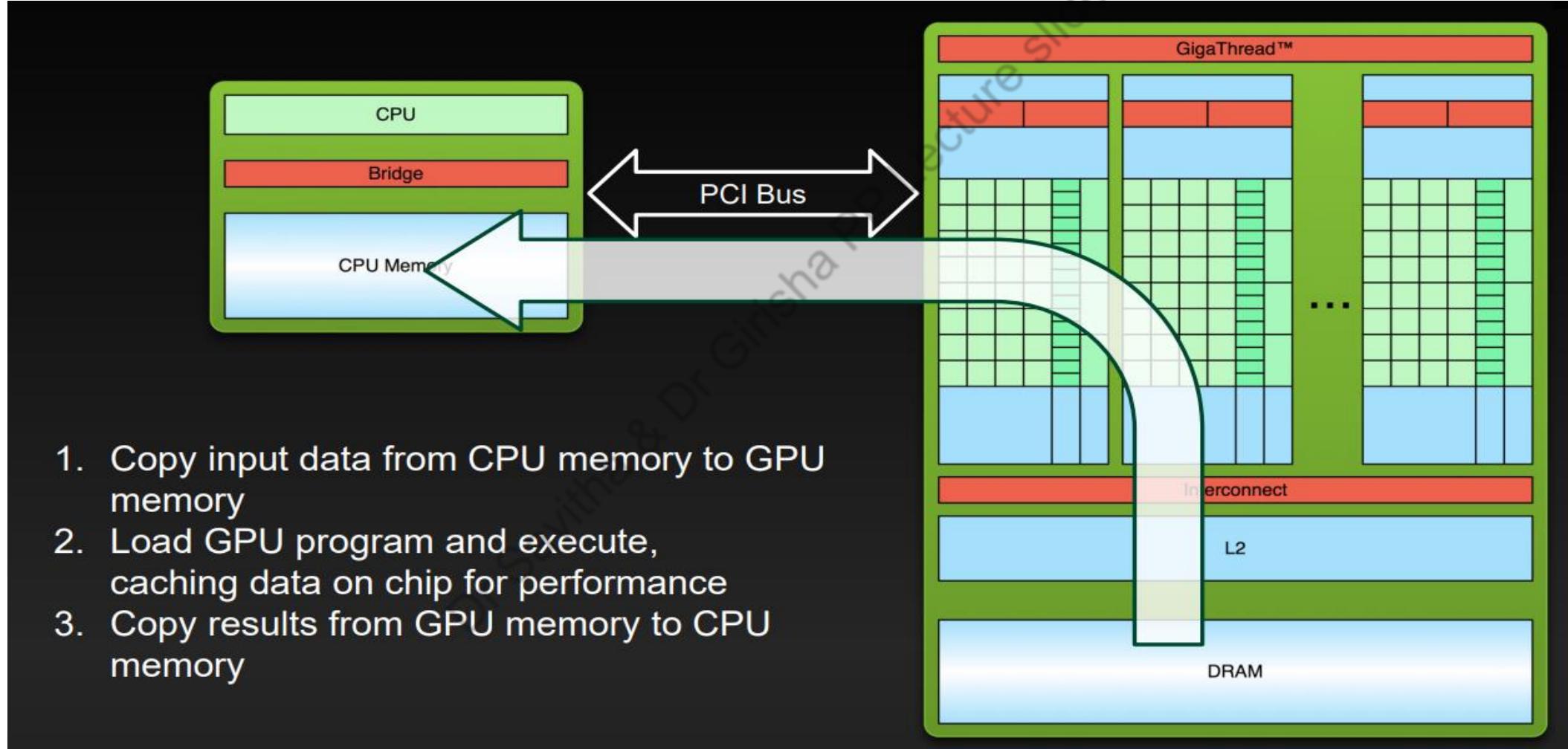
Process flow



Process flow



Process flow



Hello world example

- The `__global__` specifier indicates a function that runs on device (GPU)
 - Such function can be called through host code, e.g. the `main()` function in the example, and is also known as "kernels"
 - When a kernel is called, its execution configuration is provided through `<<<...>>>`. This is called kernel launch
- This is a simple CUDA C program with two distinctions:
 - An empty function named `kernel()` qualified with `__global__`
 - A call to the empty function, embellished with `<<<1,1>>>`
- Code is compiled by our system's standard C compiler by default
 - `__global__` qualifier: This mechanism alerts the compiler that a function should be compiled to run on a device instead of the host
 - In this simple example, `nvcc` gives the function `kernel()` to the compiler that handles device code
 - And it feeds `main()` to the host compiler

```
#include <iostream>

__global__ void kernel( void ) {
}

int main( void ) {
    kernel<<<1,1>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

Kernels

- CUDA C/C++ extends C/C++ by allowing the programmer to define C/C++ functions, called **kernels**
- When called, they are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C++ functions
- A kernel is defined using the **__global__** declaration specifier
- The number of CUDA threads that execute that kernel for a given kernel call is specified using a new **<<< >>>** execution configuration syntax
- Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through built-in variables

Kernels

- The following sample code, using the built-in variable **threadIdx**, adds two vectors A and B of size N and stores the result into vector C
- Each of the N threads that execute **VecAdd()** performs one pair-wise addition

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Thread Hierarchy

- **threadIdx** is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index
- Forming a one dimensional, two-dimensional, or three-dimensional block of threads, called a **thread block**
- The index of a thread and its thread ID relate to each other in a straightforward way

Thread Hierarchy

- Dx defines the dimension of the block
- For a one-dimensional block, they are the same;
- For a two-dimensional block of size (Dx, Dy) , the thread ID of a thread of index (x, y) is $(x + y Dx)$
- For a three-dimensional block of size (Dx, Dy, Dz) , the thread ID of a thread of index (x, y, z) is $(x + y Dx + z Dx Dy)$

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                      float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

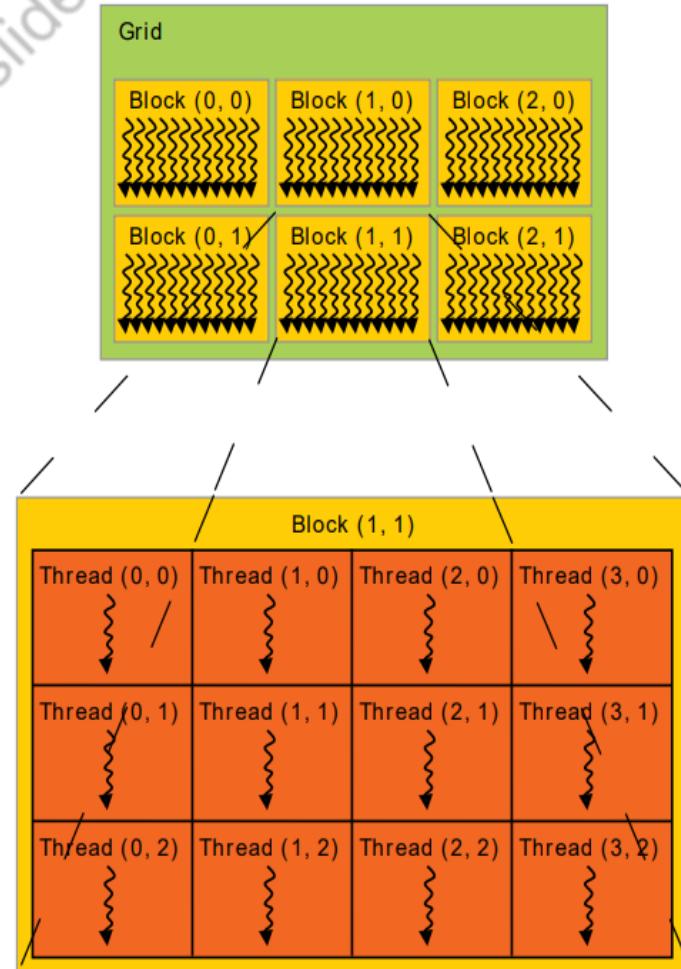
int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Thread Hierarchy

- There is a limit to the number of threads per block
- Since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core
- On current GPUs, a thread block may contain up to 1024 threads
- However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks
- Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks

Thread Hierarchy

- The number of threads per block and the number of blocks per grid is specified in the <<<>>> syntax
- Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional unique index accessible within the kernel through the built-in **blockIdx** variable
- The dimension of the thread block is accessible within the kernel through the built-in **blockDim** variable



Thread Hierarchy

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Thread Hierarchy

- A thread block size of 16x16 (256 threads)
- The grid is created with enough blocks to have one thread per matrix element as before
- The number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension
- Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series
- This independence requirement allows thread blocks to be scheduled in any order across any number of cores
- Enabling programmers to write code that scales with the number of cores

Thread Hierarchy

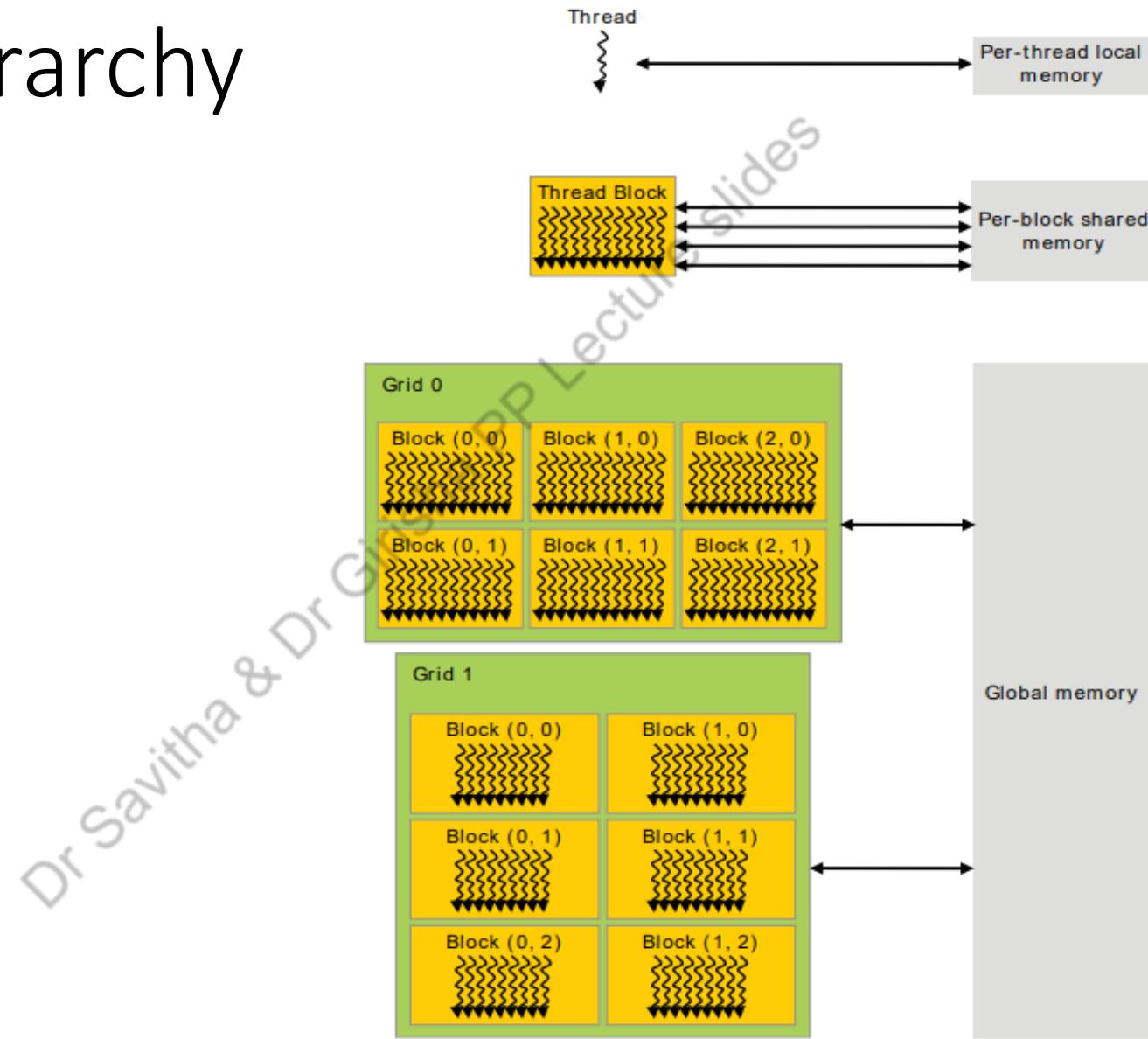
- Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses

Dr Savitha & Dr Girisha PP Lecture slides

Memory Hierarchy

- CUDA threads may access data from multiple memory spaces during their execution
- Each thread has private local memory
- Each thread block has **shared memory** visible to all threads of the block and with the same lifetime as the block
- All threads have access to the same **global memory**
- There are also two additional read-only memory spaces accessible by all threads: the **constant** and **texture memory** spaces
- The global, constant, and texture memory spaces are optimized for different memory usages
- The global, constant, and texture memory spaces are persistent across kernel launches by the same application

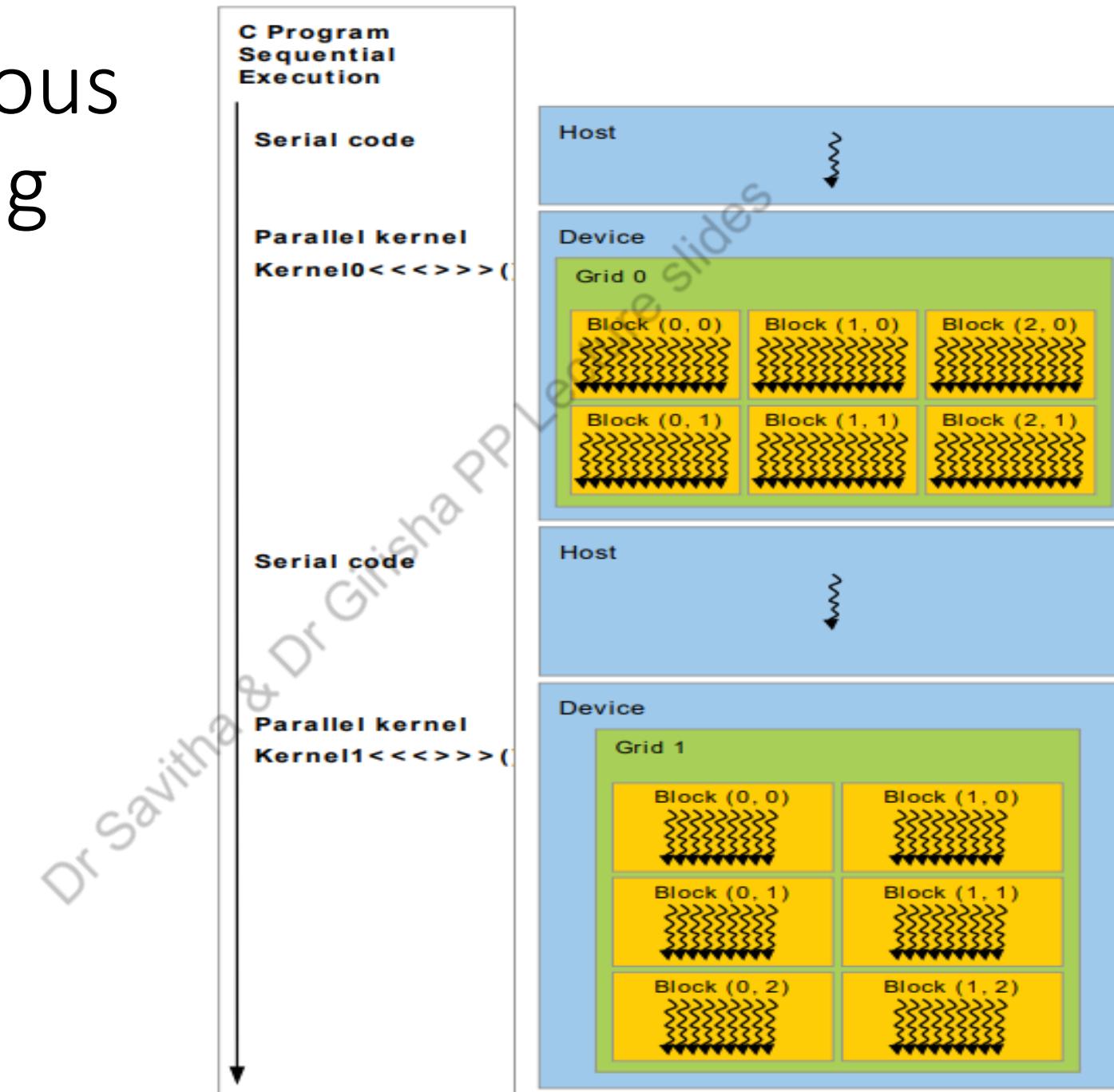
Memory Hierarchy



Heterogeneous Programming

- The CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C++ program
- The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM, referred to as host memory and device memory
- Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime
- This includes device memory allocation and deallocation as well as data transfer between host and device memory

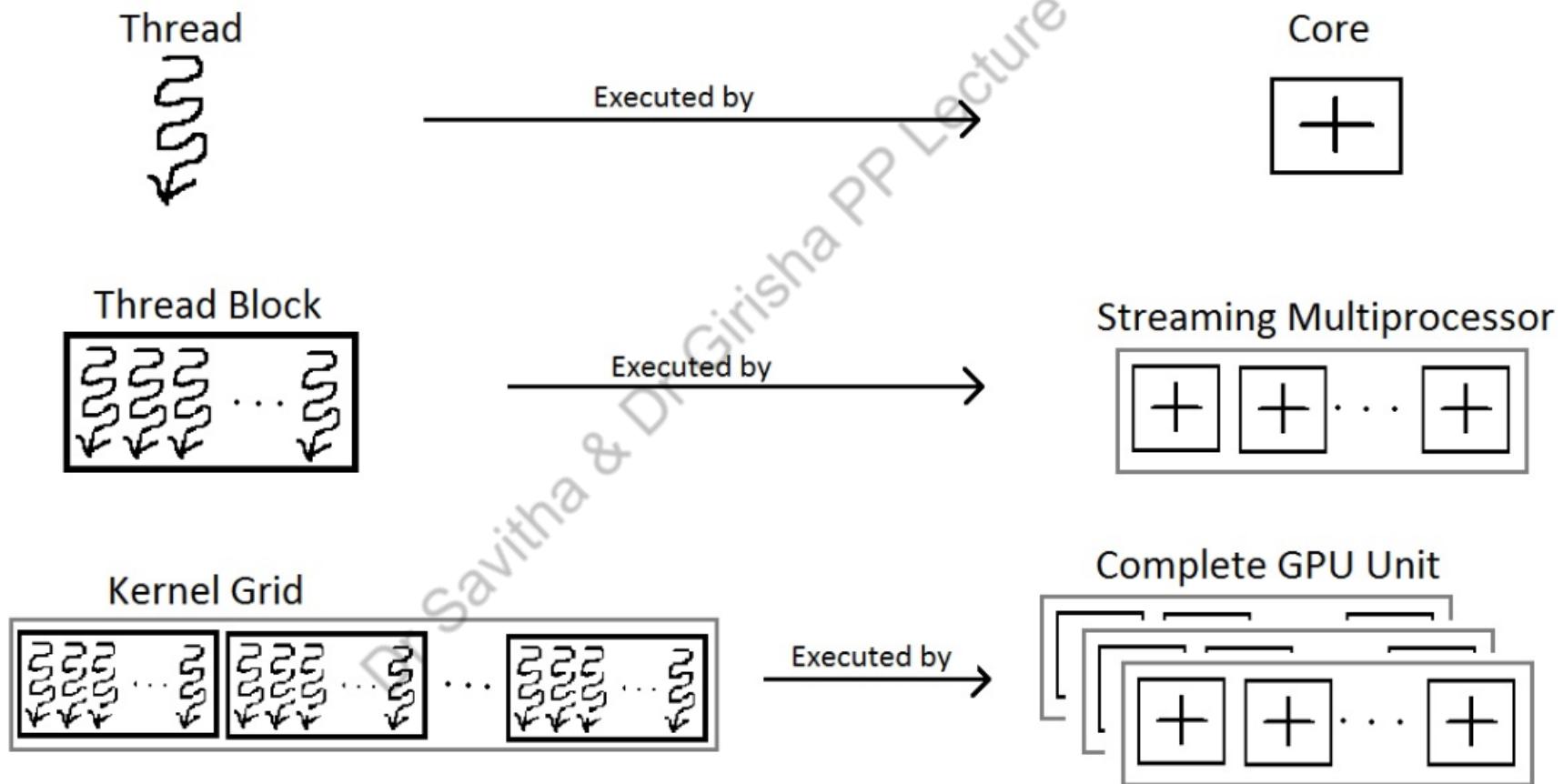
Heterogeneous Programming



Streaming Multiprocessors

- Hardware groups several threads that execute the same instructions into **wraps**
- Several wraps constitute a thread block
- Several thread blocks are assigned to a Streaming Multiprocessors (SM)
- Several SM constitute a GPU
- SM: These are general purpose processors with low clock rate and small cache
- The primary task of an SM is that it must execute several thread blocks in parallel
- As soon as one of its thread block has completed execution, it takes up the serially next thread block

Streaming Multiprocessors



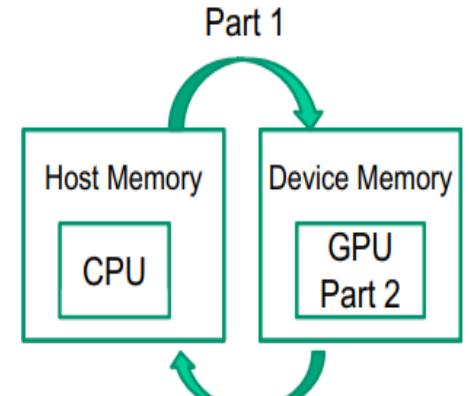
Streaming Multiprocessors

- 1. Execution cores:** single precision floating-point units, double precision floating-point units, special function units (SFUs)
- 2. Caches**
 1. L1 cache. (for reducing memory access latency).
 2. Shared memory (for shared data between threads).
 3. Constant cache (for broadcasting of reads from a read-only memory).
 4. Texture cache (for aggregating bandwidth from texture memory).
- 3. Schedulers for warps:** these are for issuing instructions to warps based on a particular scheduling policies
- 4. A substantial number of registers:** an SM may be running a large number of active threads at a time, so it is a must to have registers in thousands

DEVICE GLOBAL MEMORY AND DATA TRANSFER

- Host and devices have separate memory spaces
- To execute a kernel on a device, the programmer needs to allocate global memory (device memory) on the device and transfer pertinent data from the host memory to the allocated device memory
- After device execution, the programmer needs to transfer result data from the device memory back to the host memory and free up the device memory that is no longer needed.

```
#include <cuda.h>
...
void vecAdd(float* A, float*B, float* C, int n)
{
    int size = n* sizeof(float);
    float *A_d, *B_d, *C_d;
    ...
    1. // Allocate device memory for A, B, and C
       // copy A and B to device memory
    ...
    2. // Kernel launch code – to have the device
       // to perform the actual vector addition
    ...
    3. // copy C from the device memory
       // Free device vectors
}
```



DEVICE GLOBAL MEMORY AND DATA TRANSFER

- The CUDA runtime system provides API functions for managing data in the device memory
 - `cudaMalloc()`
 - Allocates object in the device global memory
 - Two parameters
 - **Address of a pointer** to the allocated object
 - **Size** of allocated object in terms of bytes
 - `cudaFree()`
 - Frees object from device global memory
 - **Pointer** to freed object

DEVICE GLOBAL MEMORY AND DATA TRANSFER

- Function **cudaMalloc()** can be called from the host code to allocate a piece of device global memory for an object
- The first parameter to the **cudaMalloc()** function is the **address of a pointer variable** that will be set to point to the allocated object
- The address of the pointer variable should be cast to **(void)** because the function expects a generic pointer
- This parameter allows the `cudaMalloc()` function to write the address of the allocated memory into the pointer variable
- The host code passes this pointer value to the kernels that need to access the allocated memory
- The second parameter to the `cudaMalloc()` function gives the size of the data to be allocated, in terms of bytes

DEVICE GLOBAL MEMORY AND DATA TRANSFER

```
float *d_A  
int size = n * sizeof(float);  
cudaMalloc((void**)&d_A, size);  
...  
cudaFree(d_A);
```

After the computation, cudaFree() is called with pointer **d_A** as input to free the storage space

DEVICE GLOBAL MEMORY AND DATA TRANSFER

- Once the host code has allocated device memory for the data objects, it can request that data be transferred from host to device
- This is accomplished by calling one of the CUDA API functions **cudaMemcpy()**
- The **cudaMemcpy()** function takes four parameters
 - A pointer to the **destination location** for the data object to be copied
 - The second parameter points to the **source location**
 - The third parameter specifies the **number of bytes to be copied**
 - The fourth parameter indicates the **types of memory involved** in the copy:
 - from host memory to host memory
 - from host memory to device memory
 - from device memory to host memory
 - from device memory to device memory

DEVICE GLOBAL MEMORY AND DATA TRANSFER

`cudaMemcpy()`

- memory data transfer
- Requires four parameters
 - Pointer to destination
 - Pointer to source
 - Number of bytes copied
 - Type/Direction of transfer

DEVICE GLOBAL MEMORY AND DATA TRANSFER

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code - to be shown later
    ...

cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

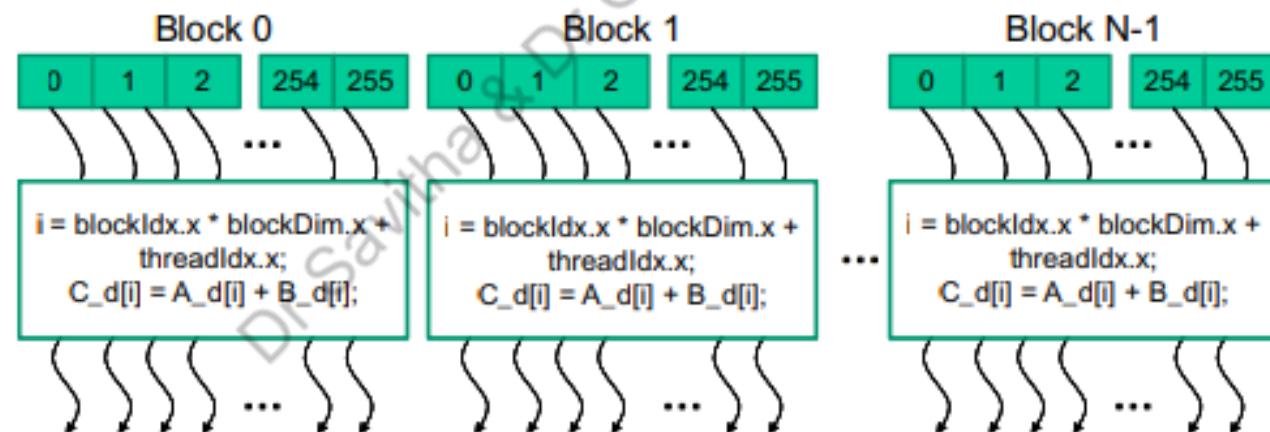
    // Free device memory for A, B, C
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

KERNEL FUNCTIONS AND THREADING

- In CUDA, a **kernel function** specifies the code to be executed by all threads during a parallel phase by the device
- Since all these threads execute the same code, CUDA programming is an instance of the well-known **SPMD**
- When a host code launches a kernel, the CUDA runtime system generates a grid of threads that are organized in a **two-level hierarchy**
- Grid is organized into an array of thread blocks
- All blocks of a grid are of the same size; each block can contain up to 1,024 threads

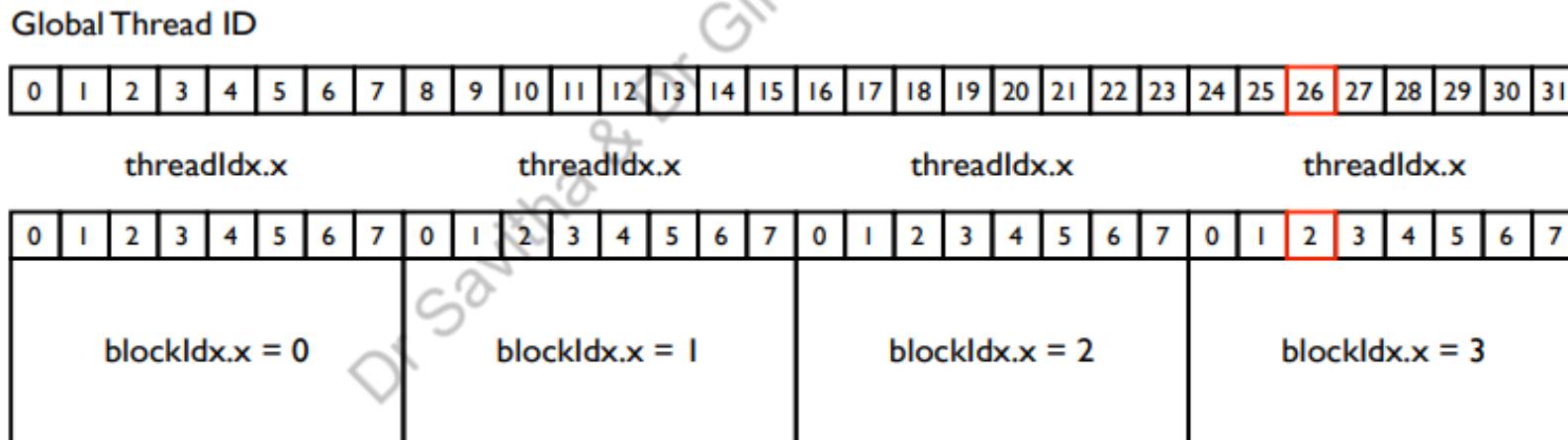
KERNEL FUNCTIONS AND THREADING

- The number of threads in each thread block is specified by the host code when a kernel is launched
- The same kernel can be launched with different numbers of threads at different parts of the host code
- For a given grid of threads, the number of threads in a block is available in the **blockDim** variable



KERNEL FUNCTIONS AND THREADING

- Each thread in a block has a unique **threadIdx** value
- This allows each thread to combine its **threadIdx** and **blockIdx** values to create a unique **global index** for itself with the entire grid



KERNEL FUNCTIONS AND THREADING

- By launching the kernel with a larger number of blocks, one can process larger vectors
- By launching a kernel with n or more threads, one can process vectors of length n

```
// Compute vector sum C = A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

KERNEL FUNCTIONS AND THREADING

- **__global__**

- keyword indicates that the function being declared is a CUDA kernel function
- Function is to be executed on the device and can only be called from the host code

- **__device__**

- keyword indicates that the function being declared is a CUDA device function
- A device function executes on a CUDA device and can only be called from a kernel function or another device function

- **__host__**

- Host function is simply a traditional C function that executes on the host and can only be called from another host function

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

KERNEL FUNCTIONS AND THREADING

```
void vecAdd(float* A, float* B, float* C, int n)
{
    int size = n * sizeof(float);
    float *d_A, *d_B, *d_C;

    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &B_d, size);
    cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

    cudaMalloc((void **) &d_C, size);
    vecAddKernel<<<ceil(n/2560), 256>>>(d_A, d_B, d_C, n);

    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);
    // Free device memory for A, B, C
    cudaFree(d_Ad); cudaFree(d_B); cudaFree (d_C);
}
```

KERNEL FUNCTIONS AND THREADING

- The code is hardwired to use thread blocks of 256 threads each
- The number of thread blocks used, however, depends on the length of the vectors (n)
- If n is 750, three thread blocks will be used; if n is 4,000, 16 thread blocks will be used; if n is 2,000,000, 7,813 blocks will be used
- A small GPU with a small amount of execution resources may execute one or two of these thread blocks in parallel
- A larger GPU may execute 64 or 128 blocks in parallel

CUDA THREAD ORGANIZATION

- Threads are organized into a two-level hierarchy
 - **Grid**
 - **Blocks**
- All threads in a block share the same block index
- CUDA provides built-in, preinitialized variables that can be accessed within kernel functions
 - **blockIdx**
 - **threadIdx**
 - **gridDim**
 - **blockDim**

CUDA THREAD ORGANIZATION

- A grid is a 3D array of blocks and each block is a 3D array of thread
- The programmer can choose to use fewer dimensions by setting the unused dimensions to 1
- The exact organization of a grid is determined by the execution configuration parameters (within <<< and >>>) of the kernel launch statement
- The first execution configuration parameter specifies the dimensions of the grid in number of blocks. The second specifies the dimensions of each block in number of thread

CUDA THREAD ORGANIZATION

```
dim3 dimGrid(ceil(n/256.0), 1, 1);
dim3 dimBlock(256, 1, 1);
vecAddKernel <<< dimGrid, dimBlock >>> (...);
```

- This allows the number of blocks to vary with the size of the vectors so that the grid will have enough threads to cover all vector elements
- The value of variable n at kernel launch time will determine the dimension of the grid
- If n is equal to 1,000, the grid will consist of four blocks. If n is equal to 4,000, the grid will have 16 blocks

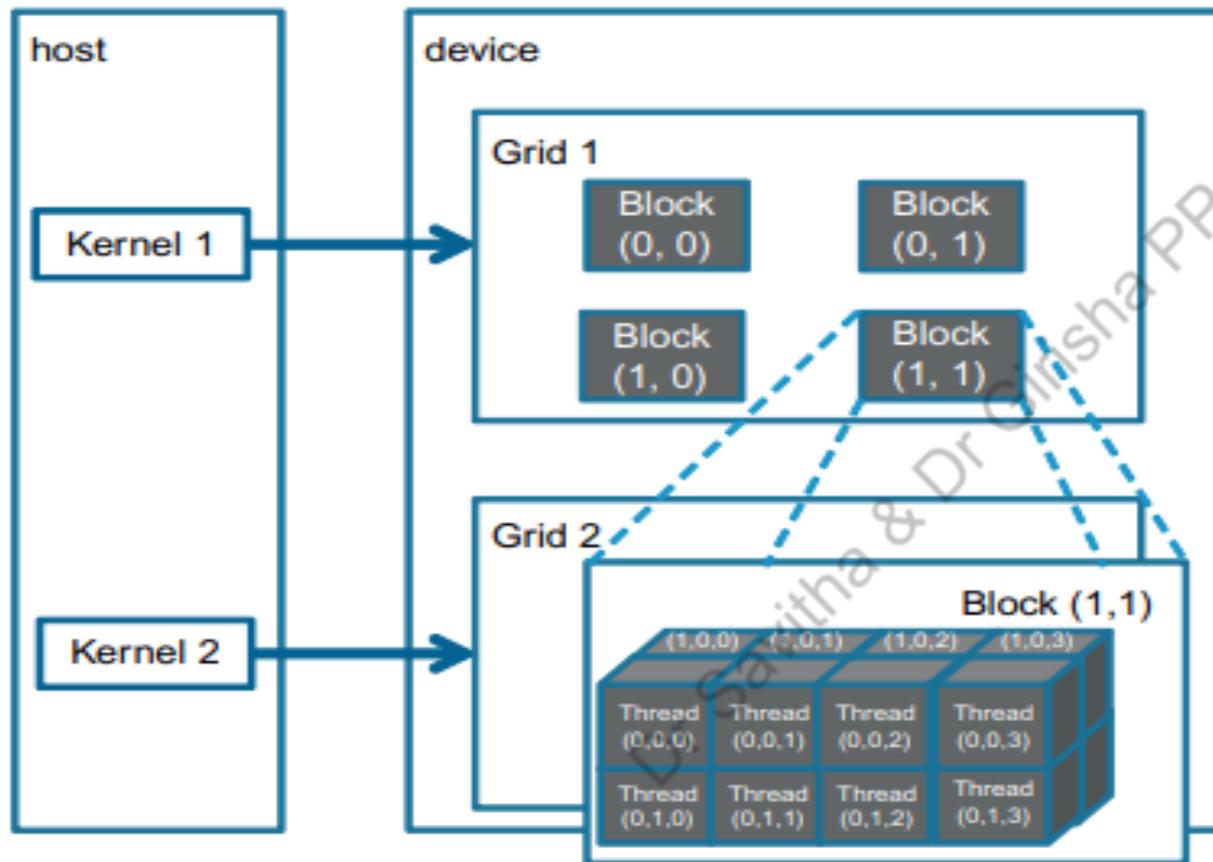
CUDA THREAD ORGANIZATION

- In CUDA C, the allowed values of `gridDim.x`, `gridDim.y`, and `gridDim.z` range from 1 to 65,536
- All threads in a block share the same `blockIdx.x`, `blockIdx.y`, and `blockIdx.z` values
- Among all blocks, the `blockIdx.x` value ranges between 0 and `gridDim.x-1`, the `blockIdx.y` value between 0 and `gridDim.y-1`, and the `blockIdx.z` value between 0 and `gridDim.z-1`.

CUDA THREAD ORGANIZATION

- Blocks are organized into 3D arrays of threads
- Two-dimensional blocks can be created by setting the z dimension to 1. One-dimensional blocks can be created by setting both the y and z dimensions to 1
- The number of threads in each dimension of a block is specified by the second execution configuration parameter at the kernel launch
- The total size of a block is limited to 1,024 threads, with flexibility in distributing these elements into the three dimensions as long as the total number of threads does not exceed 1,024
 - For example, (512, 1, 1), (8, 16, 4), and (32, 16, 2) are all allowable blockDim values, but (32, 32, 2) is not allowable since the total number of threads would exceed 1,024

CUDA THREAD ORGANIZATION



```
dim3 dimBlock(2, 2, 1);  
dim3 dimGrid(4, 2, 2);  
KernelFunction<<<dimGrid, dimBlock>>>(...);  
dimBlock, dimGrid
```

MAPPING THREADS TO MULTIDIMENSIONAL DATA

- The choice of 1D, 2D, or 3D thread organizations is usually based on the nature of the data
- For example, pictures are a 2D array of pixels
- It is often convenient to use a 2D grid that consists of 2D blocks to process the pixels in a picture
- Consider a picture of 76×62 picture

MAPPING THREADS TO MULTIDIMENSIONAL DATA

- Assume that we decided to use a 16×16 block, with 16 threads in the x direction and 16 threads in the y direction
- We will need five blocks in the x direction and four blocks in the y direction, which results in $5 \times 4 = 20$ blocks



MAPPING THREADS TO MULTIDIMENSIONAL DATA

- Assume that the host code uses an integer variable n to track the number of pixels in the x direction, and another integer variable m to track the number of pixels in the y direction
- We further assume that the input picture data has been copied to the **device memory** and can be accessed through a pointer variable **d_Pin**
- The output picture has been allocated in the device memory and can be accessed through a pointer variable **d_Pout**

```
dim3 dimBlock(ceil(n/16.0), ceil(m/16.0), 1);
dim3 dimGrid(16, 16, 1);
pictureKernel<<<dimGrid, dimBlock>>>(d_Pin, d_Pout, n, m);
```

MAPPING THREADS TO MULTIDIMENSIONAL DATA

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout, int n, int m) {  
  
    // Calculate the row # of the d_Pin and d_Pout element to process  
    int Row = blockIdx.y*blockDim.y + threadIdx.y;  
  
    // Calculate the column # of the d_Pin and d_Pout element to process  
    int Col = blockIdx.x*blockDim.x + threadIdx.x;  
  
    // each thread computes one element of d_Pout if in range  
    if ((Row < m) && (Col < n)) {  
        d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];  
    }  
}
```

MAPPING THREADS TO MULTIDIMENSIONAL DATA

- We can easily extend our discussion of 2D arrays to 3D arrays
- This is done by placing each “plane” of the array one after another
- The programmer also needs to determine the values of `blockDim.z` and `gridDim.z` when launching a kernel
- In the kernel, the array index will involve another global index:

`int Plane = blockIdx.z*blockDim.z + threadIdx.z`
- One would of course need to test if all the three global indices—`Plane`, `Row`, and `Col`—fall within the valid range of the array

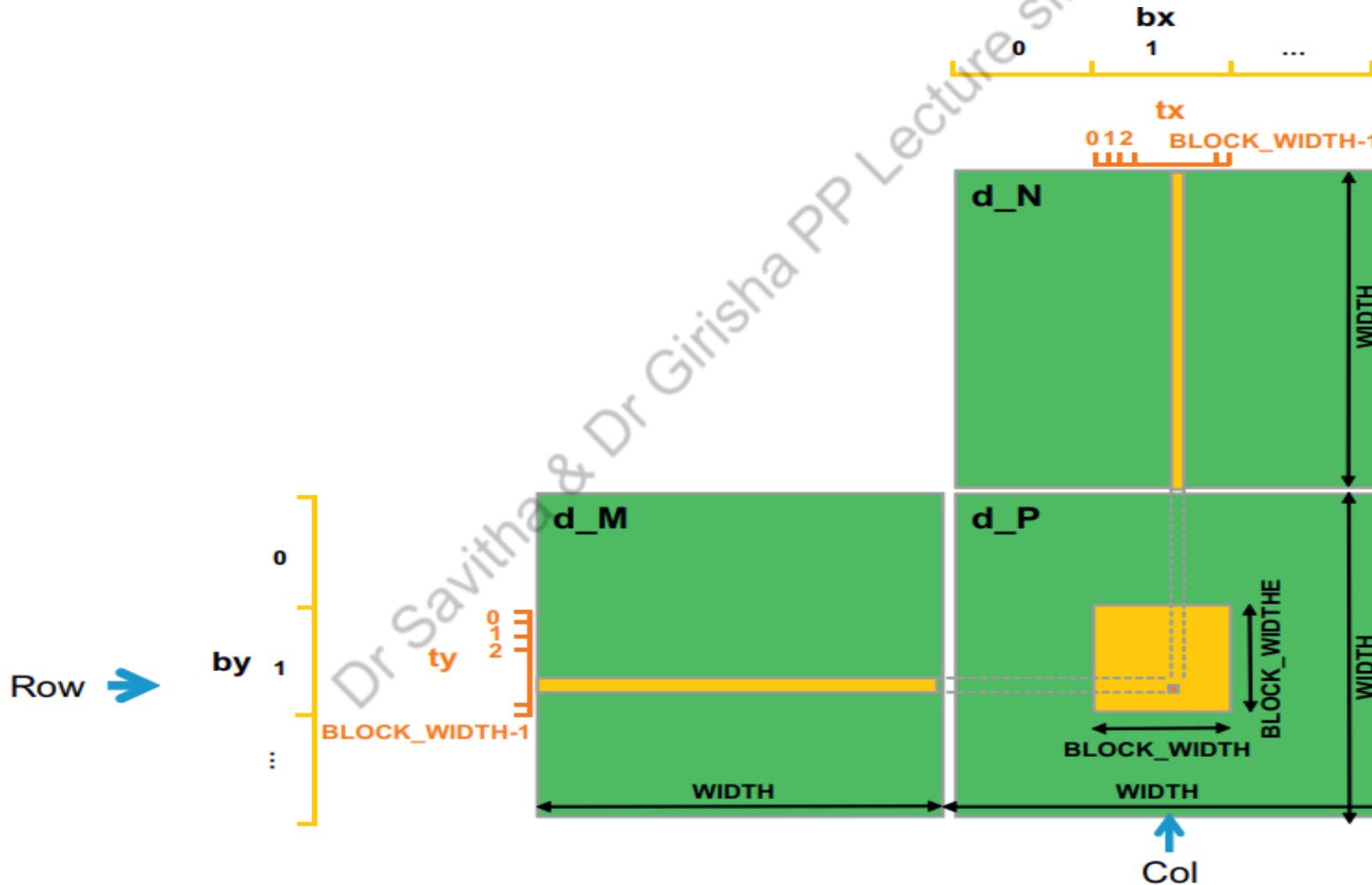
MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

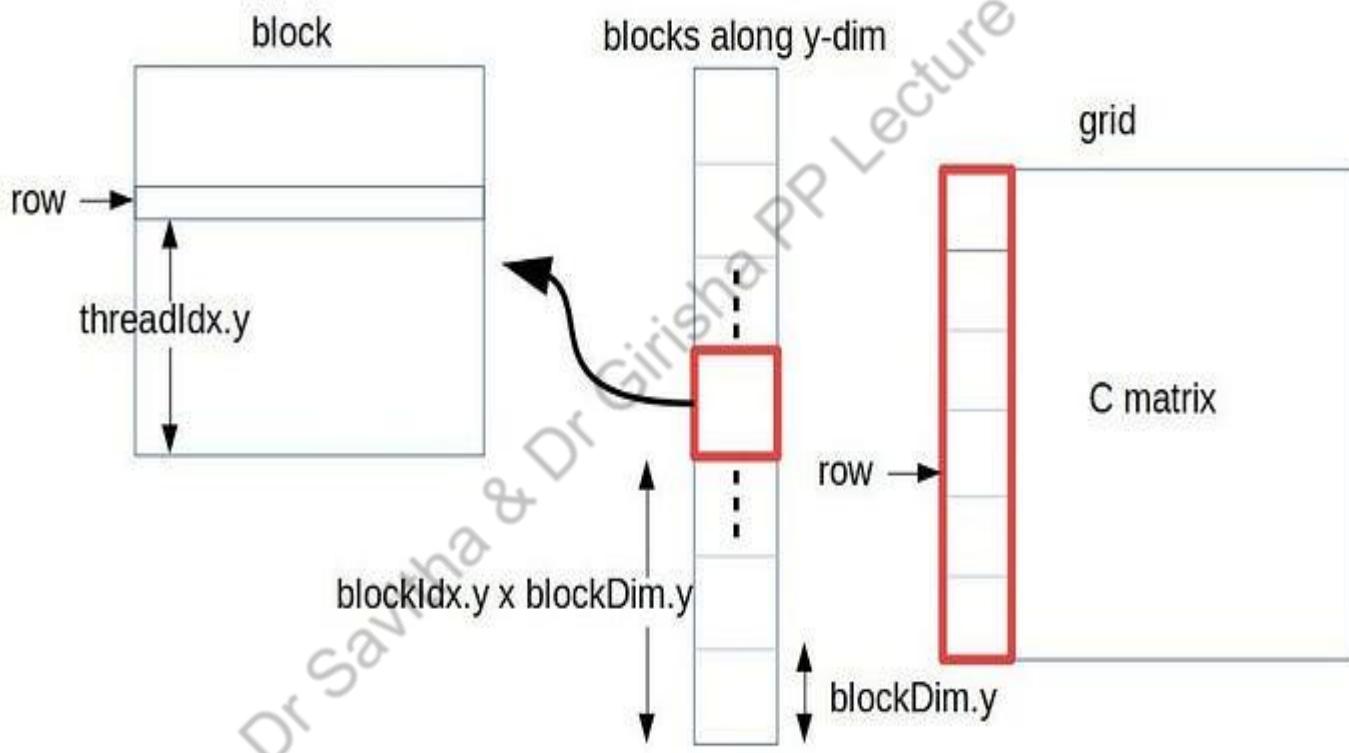
- We have studied **vecAddkernel()** and **pictureKernel()** where each thread performs only one floating-point arithmetic operation on one array element
 - two simple kernels were selected for teaching the mapping of threads to data using **threadIdx**, **blockIdx**, **blockDim**, and **gridDim** variables
 - the number of threads that we create is a multiple of the block dimension
 - As a result, we will likely end up with more threads than data elements
 - Not all threads will process elements of an array. We use an if statement to test if the global index values of a thread are within the valid range

MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

- Matrix-matrix multiplication between an $I \times J$ matrix d_M and a $J \times K$ matrix d_N produces an $I \times K$ matrix d_P .
- When performing a matrix multiplication, each element of the product matrix d_P is an inner product of a row of d_M and a column of d_N .
- The inner product between two vectors is the sum of products of corresponding elements

MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL





MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {  
    // Calculate the row index of the d_Pelement and d_M  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
  
    // Calculate the column index of d_P and d_N  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        // each thread computes one element of the block sub-matrix  
        for (int k = 0; k < Width; ++k) {  
            Pvalue += d_M[Row*Width+k] *d_N[k*Width+Col];  
        }  
        d_P[Row*Width+Col] = Pvalue;  
    }  
}
```

MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

- Let us assume that, BLOCK_WIDTH=16
- Assume that we have a Width value of 1,000. That is, we need to do 1,000 x 1,000 matrix multiplication
- For a BLOCK_WIDTH value of 16, we will generate 16 x 16 blocks
- There will be 64 x 64 blocks in the grid to cover all d_P elements.

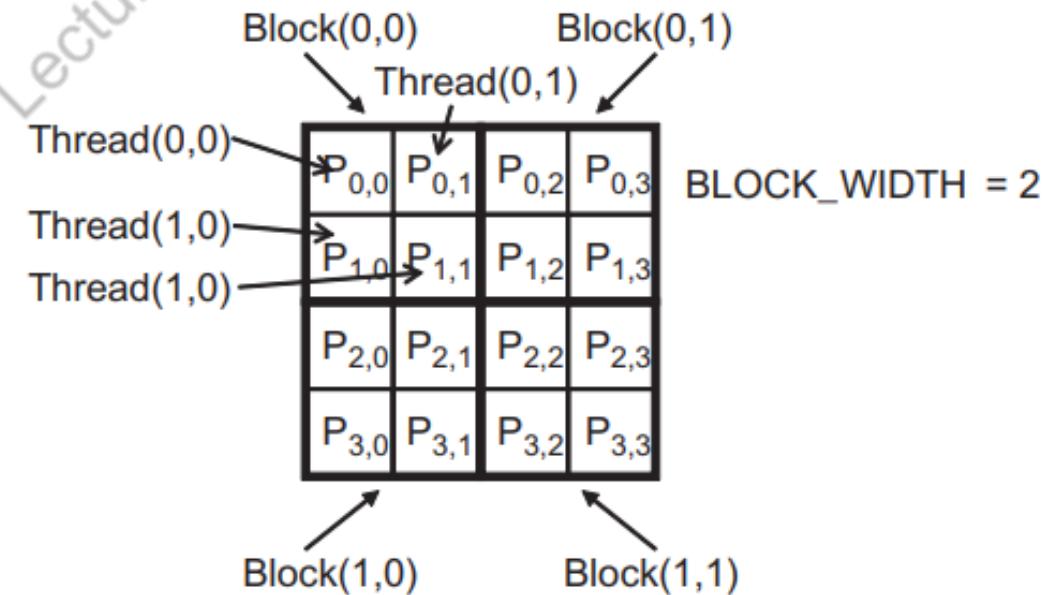
```
#define BLOCK_WIDTH 16

// Setup the execution configuration
int NumBlocks = Width/BLOCK_WIDTH;
if (Width % BLOCK_WIDTH) NumBlocks++;
dim3 dimGrid(NumBlocks, NumBlocks);
dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);

// Launch the device computation threads!
matrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

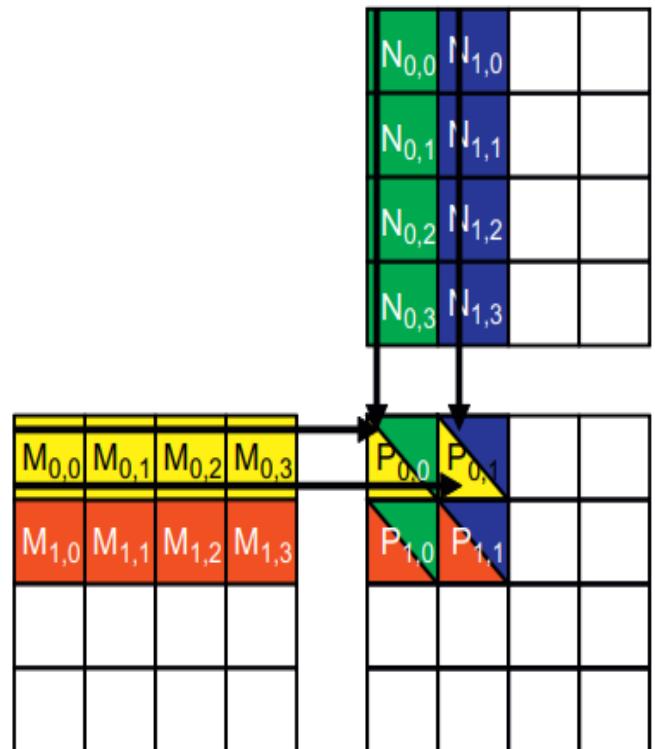
MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

- For a `BLOCK_WIDTH=32`, there will be 32×32 blocks in the grid to cover all `d_P` elements
- Let us consider a smaller matrix of size 4×4
- `BLOCK_WIDTH = 2`
- The `d_P` matrix is now divided into four tiles and each block calculates one tile



MATRIX-MATRIX MULTIPLICATION—A MORE COMPLEX KERNEL

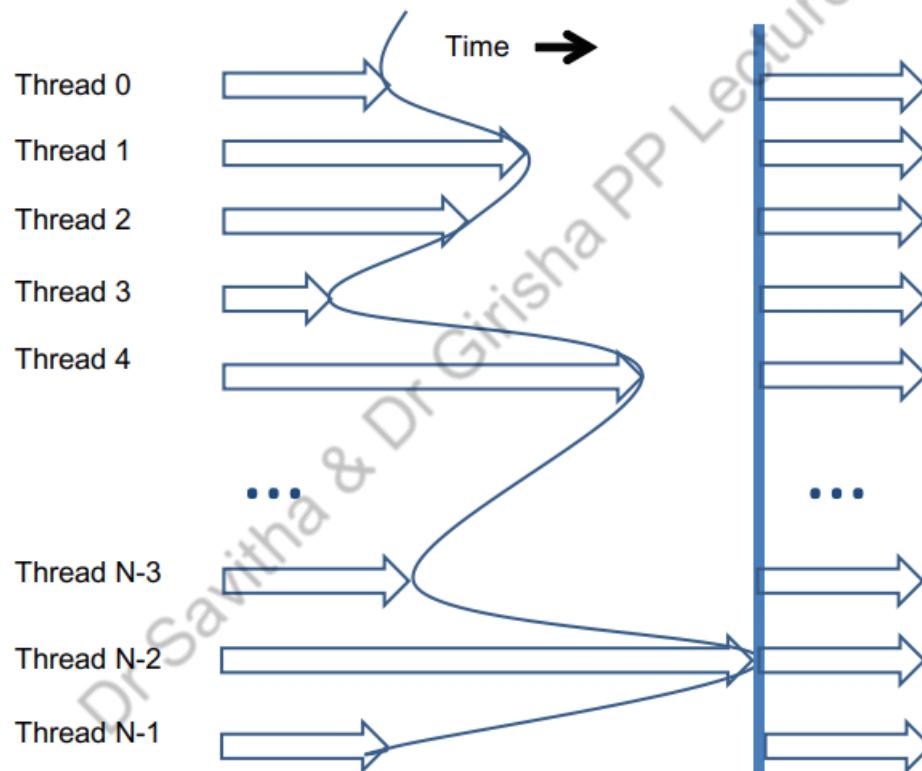
- For the small matrix multiplication, threads in block(0,0) produce four dot products
- The Row and Col variables of thread(0,0) in block(0,0) are $0*0 + 0 = 0$ and $0*0 + 0 = 0$
- It maps to P(0,0) and calculates the dot product of row 0 of M and column 0 of N



SYNCHRONIZATION AND TRANSPARENT SCALABILITY

- CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function **`__syncthreads()`**
- When a kernel function calls **`__syncthreads()`**, all threads in a block will be held at the calling location until every thread in the block reaches the location
- This ensures that all threads in a block have completed a phase of their execution of the kernel before any of them can move on to the next phase

SYNCHRONIZATION AND TRANSPARENT SCALABILITY



SYNCHRONIZATION AND TRANSPARENT SCALABILITY

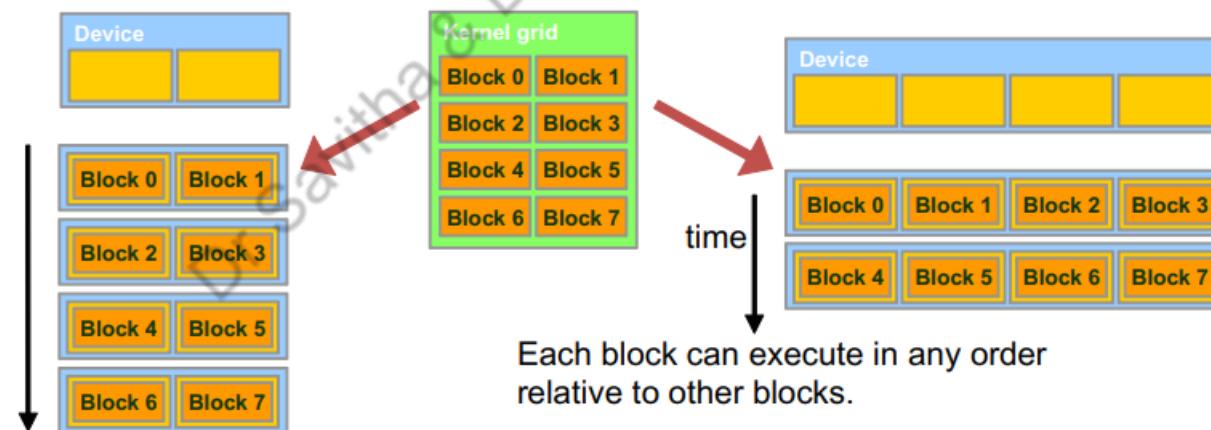
- In CUDA, a `__syncthreads()` statement, if present, must be executed by all threads in a block
- When a `__syncthread()` statement is placed in an **if statement**, either all threads in a block execute the path that includes the `__syncthreads()` or none of them does
- For an **if-then-else statement**, if each path has a `__syncthreads()` statement, either all threads in a block execute the `__syncthreads()` on the **then** path or all of them execute the **else** path
- The two `__syncthreads()` are different barrier synchronization points
- If a thread in a block executes the **then** path and another executes the **else** path, they would be waiting at different barrier synchronization points
- They would end up waiting for each other forever

SYNCHRONIZATION AND TRANSPARENT SCALABILITY

- One needs to make sure that all threads involved in the barrier synchronization have access to the necessary resources to eventually arrive at the barrier
- Otherwise, a thread that never arrived at the barrier synchronization point can cause everyone else to wait forever
- CUDA runtime systems satisfy this constraint by assigning execution resources to all threads in a block as a unit
- A block can begin execution only when the runtime system has secured all the resources needed for all threads in the block to complete execution
- When a thread of a block is assigned to an execution resource, all other threads in the same block are also assigned to the same resource

SYNCHRONIZATION AND TRANSPARENT SCALABILITY

- Threads in different blocks cannot perform barrier synchronization with each other, the CUDA runtime system can execute blocks in any order relative to each other since none of them need to wait for each other
- This flexibility enables scalable implementations

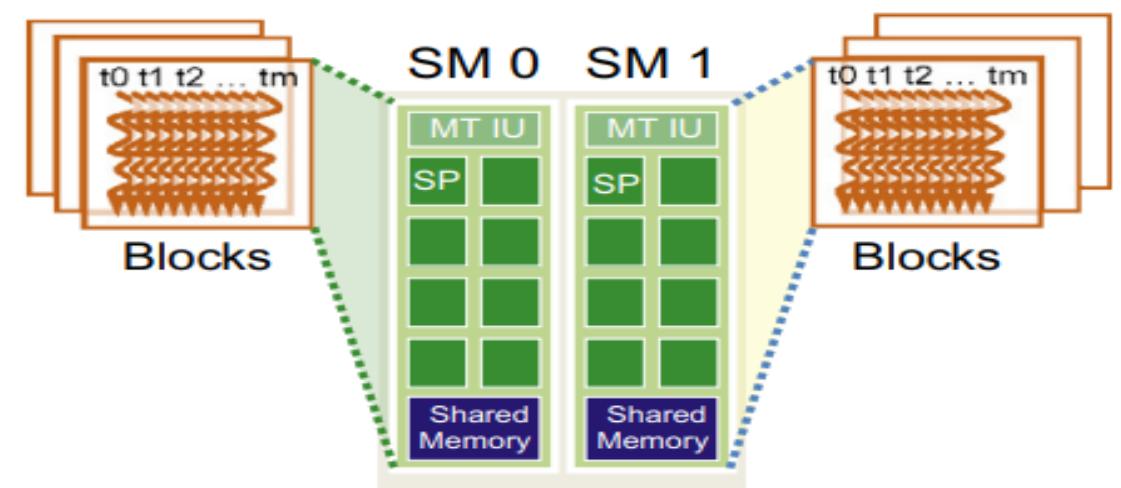


SYNCHRONIZATION AND TRANSPARENT SCALABILITY

- The ability to execute the same application code at a wide range of speeds allows the production of a wide range of implementations according to the cost, power, and performance requirements of particular market segments
- Both execute exactly the same application program with no change to the code
- The ability to execute the same application code on hardware with a different number of execution resources is referred to as ***transparent scalability***, which reduces the burden on application developers and improves the usability of applications.

ASSIGNING RESOURCES TO BLOCKS

- Once a kernel is launched, the CUDA runtime system generates the corresponding grid of threads
- Threads are assigned to execution resources on a block-by-block basis
- In the current generation of hardware, the execution resources are organized into streaming multiprocessors (SMs)



ASSIGNING RESOURCES TO BLOCKS

- For example, a CUDA device may allow up to eight blocks to be assigned to each SM
- In situations where there is an insufficient amount of any one or more types of resources needed for the simultaneous execution of eight blocks, the CUDA runtime automatically reduces the number of blocks assigned to each SM until their combined resource usage falls under the limit
- With a limited numbers of SMs and a limited number of blocks that can be assigned to each SM, there is a limit on the number of blocks that can be actively executing in a CUDA device

ASSIGNING RESOURCES TO BLOCKS

- One of the SM resource limitations is the number of threads that can be simultaneously tracked and scheduled
- It takes hardware resources for SMs to maintain the thread and block indices and track their execution status
- In more recent CUDA device designs, up to 1,536 threads can be assigned to each SM
- If a CUDA device has 30 SMs and each SM can accommodate up to 1,536 threads, the device can have up to 46,080 threads simultaneously residing in the CUDA device for execution

THREAD SCHEDULING AND LATENCY TOLERANCE

- Once a block is assigned to a SM, it is further divided into 32-thread units called **warps**
- The size of warps is implementation-specific
- In fact, warps are not part of the CUDA specification
- The size of warps is a property of a CUDA device, which is in the **dev_prop.warpSize**
- The warp is the unit of thread scheduling in SMs

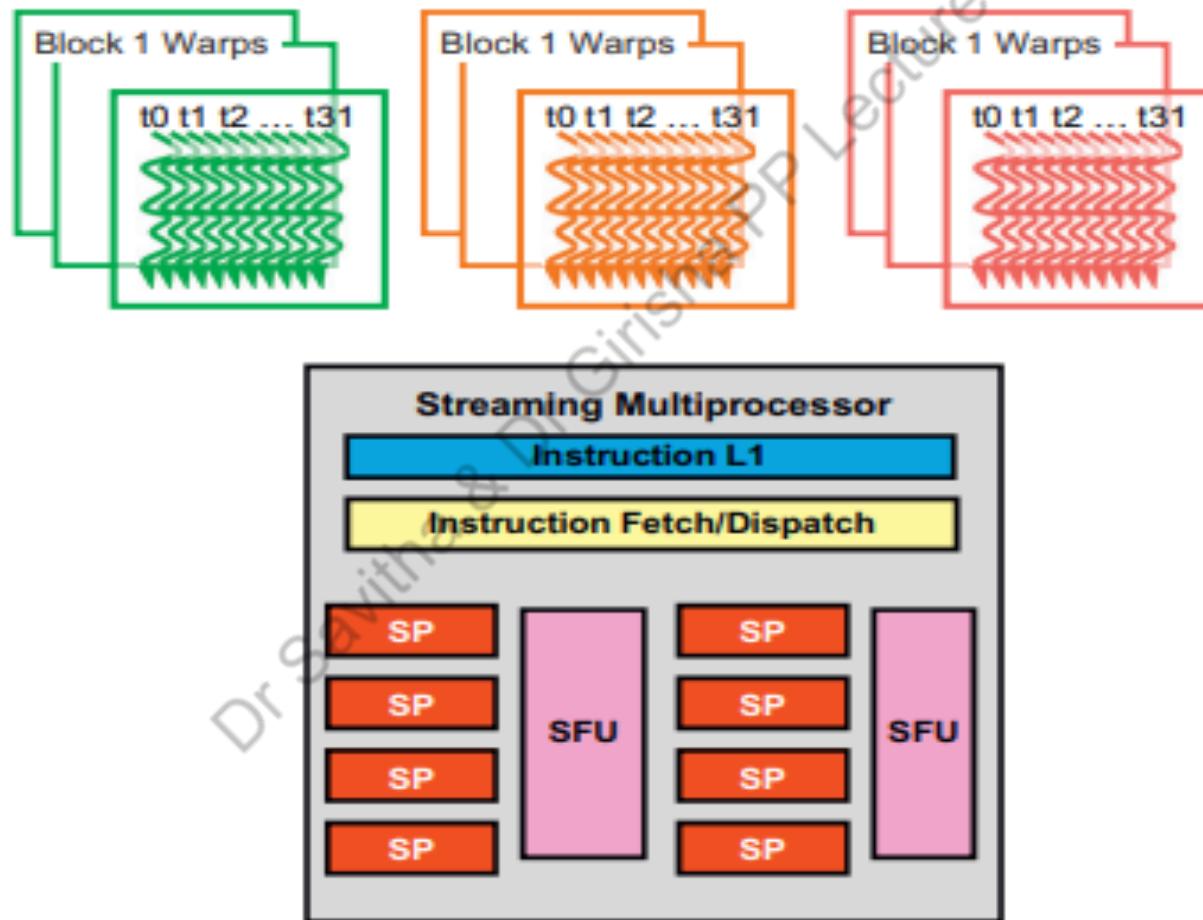
THREAD SCHEDULING AND LATENCY TOLERANCE

- Each warp consists of 32 threads of consecutive **threadIdx** values: threads 0-31 form the first warp, 32-63 the second warp, and so on
- We can calculate the number of warps that reside in an SM for a given block size and a given number of blocks assigned to each SM
- If each block has 256 threads, we can determine that each block has $256/32$ or 8 warps. With three blocks in each SM, we have $8 * 3 = 24$ warps in each SM

THREAD SCHEDULING AND LATENCY TOLERANCE

- An SM is designed to execute all threads in a warp following the **single instruction, multiple data (SIMD)** model
- That is, at any instant in time, one instruction is fetched and executed for all threads in the warp
- These threads will apply the same instruction to different portions of the data
- As a result, all threads in a warp will always have the **same execution timing**

THREAD SCHEDULING AND LATENCY TOLERANCE



THREAD SCHEDULING AND LATENCY TOLERANCE

- In general, there are fewer SPs than the number of threads assigned to each SM
- That is, each SM has only enough hardware to execute instructions from a small **subset of all threads** assigned to the SM at any point in time
- This is how CUDA processors efficiently execute **long-latency operations such as global memory accesses**

THREAD SCHEDULING AND LATENCY TOLERANCE

- When an instruction executed by the threads in a warp needs to wait for the result of a previously initiated long-latency operation, the warp is not selected for execution
- Another resident warp that is no longer waiting for results will be selected for execution
- If more than one warp is ready for execution, a **priority mechanism** is used to select one for execution
- This mechanism of filling the latency time of operations with work from other threads is often called **latency tolerance** or **latency hiding**

THREAD SCHEDULING AND LATENCY TOLERANCE

- Warp scheduling is also used for tolerating other types of operation latencies:
pipelined floating-point arithmetic and branch instructions
- With enough warps around, the hardware will likely find a warp to execute at any point in time, thus making full use of the execution hardware in spite of these long-latency operations
- The selection of ready warps for execution does not introduce any idle time into the execution timeline, which is referred to as **zero-overhead thread scheduling**

THREAD SCHEDULING AND LATENCY TOLERANCE

- This ability to tolerate long operation latencies is the main reason why GPUs do not dedicate nearly as much **chip area** to **cache memories** and **branch prediction mechanisms** as CPUs
- Assume that a CUDA device allows up to 8 blocks and 1,024 threads per SM, whichever becomes a limitation first.
- Furthermore, it allows up to 512 threads in each block. For matrix matrix multiplication, should we use 8×8 , 16×16 , or 32×32 thread blocks?

THREAD SCHEDULING AND LATENCY TOLERANCE

- If we use 8×8 blocks, each block would have only 64 threads. We will need $1,024/64 = 12$ blocks to fully occupy an SM
- However, the limitation on number of blocks is 8
- Hence, at max, $8 \times 64 = 512$ threads in each SM (Under utilized)
- **The 16×16 blocks give 256 threads per block. This means that each SM can take $1,024/256 = 4$ blocks. This is within the 8-block limitation (best choice)**
- The 32×32 blocks would give 1,024 threads in each block, exceeding the limit of 512 threads per block for this device

IMPORTANCE OF MEMORY ACCESS EFFICIENCY

- The global memory, which is typically implemented with dynamic random access memory (DRAM), tends to have **long access latencies** (hundreds of clock cycles) and **finite access bandwidth**
- Consider the example of matrix multiplication: The most important part of the kernel in terms of execution time is the for loop that performs inner product calculation

```
for (int k = 0; k < Width; ++k)
    Pvalue += d_M[Row*Width + k] * d_N[k*Width + Col];
```

IMPORTANCE OF MEMORY ACCESS EFFICIENCY

- In every iteration of this loop, two global memory accesses are performed for one floating-point multiplication and one floating-point addition
- One global memory access fetches a $d_M[]$ element and the other fetches a $d_N[]$ element
- One floating-point operation multiplies the $d_M[]$ and $d_N[]$ elements fetched
- The other accumulates the product into Pvalue
- Thus, the ratio of **floating-point calculation to global memory access operation** is 1:1, or 1.0

IMPORTANCE OF MEMORY ACCESS EFFICIENCY

- This ratio is called as the **compute to global memory access (CGMA)** ratio
- Defined as the number of floating point calculations performed for each access to the global memory within a region of a CUDA program
- With a CGMA ratio of 1.0, the matrix multiplication kernel will execute no more than 50 giga floating-point operations per second
- We need to increase the CGMA ratio to achieve a higher level of performance for the kernel

CUDA DEVICE MEMORY TYPES

- Global memory and constant memory:

- These types of memory can be written (W) and read (R) by the host by calling API functions
- The constant memory supports short-latency, high-bandwidth, read-only access by the device

- Registers and shared memory:

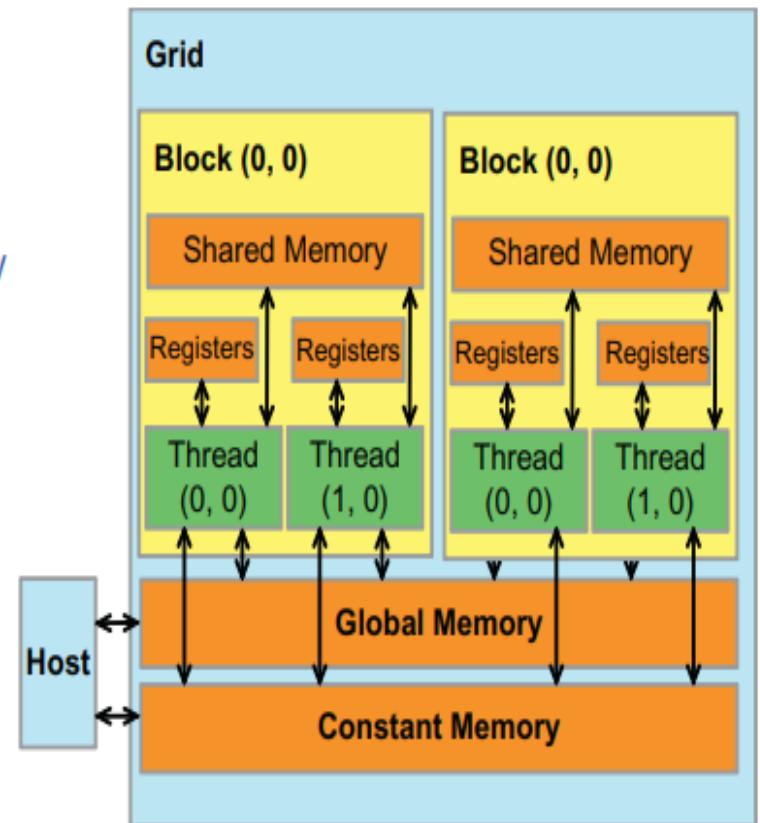
- On chip memories
- Variables that reside in these types of memory can be accessed at very high speed in a highly parallel manner
- Registers are allocated to individual threads; each thread can only access its own registers

Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

- Transfer data to/from per grid global and constant memories



CUDA DEVICE MEMORY TYPES

- Shared memory is allocated to thread blocks; all threads in a block can access variables in the shared memory locations allocated to the block
- Shared memory is an efficient means for threads to cooperate by sharing their input data and the intermediate results of their work
- By declaring a CUDA variable in one of the CUDA memory types, a CUDA programmer dictates the visibility and access speed of the variable

CUDA DEVICE MEMORY TYPES

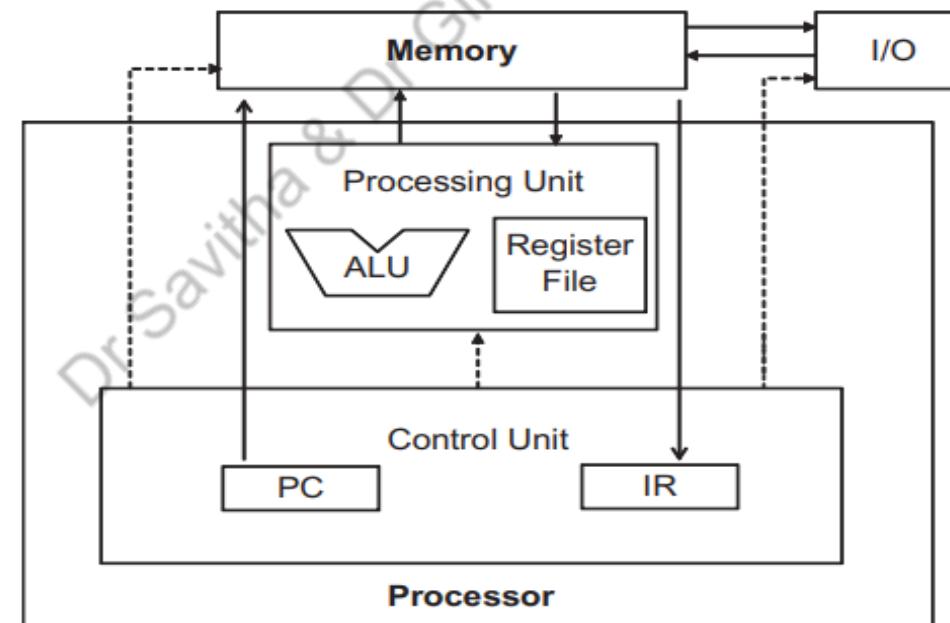
- The global memory in the CUDA programming model maps to the memory of the von Neumann model
- The global memory is off the processor chip and is implemented with DRAM technology
 - Which implies **long access latencies** and relatively **low access bandwidth**
- The registers correspond to the “register file” of the von Neumann model
 - It is on the processor chip
 - Which implies very **short access latency** and **drastically higher access bandwidth**
 - Whenever a variable is stored in a register, its accesses no longer consume off-chip global memory bandwidth
 - This will be reflected as an **increase in the CGMA ratio**

CUDA DEVICE MEMORY TYPES

- *Each access to registers involves fewer instructions than global memory*
- The processor uses the **PC**(Program Counter) value to fetch instructions from memory and puts into the **IR**(Instruction Register)
- The bits of the fetched instructions are then used to control the activities of the components of the computer
- Using the instruction bits to control the activities of the computer is referred to as **instruction execution**

CUDA DEVICE MEMORY TYPES

- The number of instructions that can be fetched and executed in each **clock cycle is limited**
- Therefore, the more instructions that need to be executed for a program, the more time it can take to execute the program



CUDA DEVICE MEMORY TYPES

- Arithmetic instructions in most modern processors have “built-in” register operands

fadd r1, r2, r3

- Where **r2** and **r3** : the input operand values can be found
- The location for storing the floating-point addition result value is specified by **r1**

CUDA DEVICE MEMORY TYPES

- When an operand of an arithmetic instruction is in a register, **there is no additional instruction required to make the operand value available to the arithmetic and logic unit**
- If an operand value is in global memory, one needs to perform a **memory load operation** to make the operand value available to the ALU
- If the first operand of a floating-point addition instruction is in global memory,

load r2, r4, offset

fadd r1, r2, r3

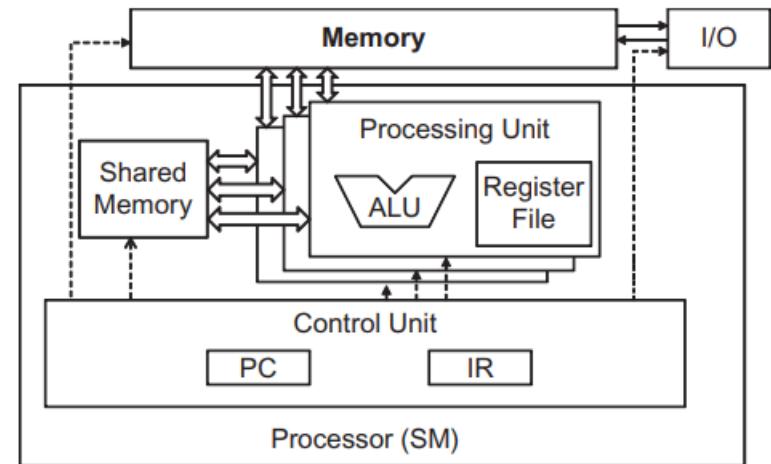
- Since the processor can only fetch and execute a limited number of instructions per clock cycle, the version with an additional load will likely take more time to process than the one without

CUDA DEVICE MEMORY TYPES

- In modern computers, the energy consumed for accessing a value from the register file is at least an order of magnitude lower than for accessing a value from the global memory

CUDA DEVICE MEMORY TYPES

- **Shared memory** is designed as part of the memory space that resides on the processor chip
- When the processor accesses data that resides in the shared memory, it needs to perform a **memory load operation**, just like accessing data in the global memory
- However, because shared memory resides on-chip, it can be accessed with much **lower latency and much higher bandwidth** than the global memory
- Because of the need to perform a load operation, share memory has **longer latency and lower bandwidth** than registers
- In computer architecture, share memory is a form of scratchpad memory



CUDA DEVICE MEMORY TYPES

- Registers, shared memory, and global memory all have different functionalities, latencies, and bandwidth
- Each such declaration also gives its declared CUDA variable a **scope and lifetime**

Table 5.1 CUDA Variable Type Qualifiers

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
<code>__device__ __shared__ int SharedVar;</code>	Shared	Block	Kernel
<code>__device__ int GlobalVar;</code>	Global	Grid	Application
<code>__device__ __constant__ int ConstVar;</code>	Constant	Grid	Application

CUDA DEVICE MEMORY TYPES

- **Scope identifies the range of threads that can access the variable:** by a single thread only, by all threads of a block, or by all threads of all grids
- If a variable's scope is a single thread, a private version of the variable will be created for every thread; each thread can only access its private version of the variable
- For example, if a kernel declares a variable of which the scope is a thread and it is launched with one million threads, one million versions of the variable will be created so that each thread initializes and uses its own version of the variable

CUDA DEVICE MEMORY TYPES

- Lifetime tells the portion of the program's execution duration when the variable is available for use:
 - Either within a kernel's execution or throughout the entire application
- If a variable's lifetime is within a kernel's execution, it must be declared within the kernel function body and will be available for use only by the kernel's code
- If the kernel is invoked several times, the value of the variable is not maintained across these invocations. Each invocation must initialize the variable to use them
- On the other hand, if a variable's lifetime is throughout the entire application, it must be declared outside of any function body
- The contents of these variables are maintained throughout the execution of the application and available to all kernels

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
<code>_device__shared_ int SharedVar;</code>	Shared	Block	Kernel
<code>_device_ int GlobalVar;</code>	Global	Grid	Application
<code>_device__constant_ int ConstVar;</code>	Constant	Grid	Application

CUDA DEVICE MEMORY TYPES

- All automatic **scalar variables** declared in kernel and device functions are placed into registers
- The scopes of these automatic variables are within **individual threads**
- When a kernel function declares an automatic variable, a **private copy of that variable is generated** for every thread that executes the kernel function
- When a thread terminates, all its automatic variables also cease to exist

CUDA DEVICE MEMORY TYPES

- Variables Row, Col, and Pvalue are all automatic variables

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, intWidth) {  
  
    // Calculate the row index of the d_P element and d_M  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
  
    // Calculate the column index of d_P and d_N  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        // each thread computes one element of the block sub-matrix  
        for (int k = 0; k < Width; ++k) {  
            Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];  
        }  
        d_P[Row*Width+Col] = Pvalue;  
    }  
}
```

CUDA DEVICE MEMORY TYPES

- Automatic array variables are not stored in registers
- Instead, they are stored into the **global memory** and incur long access delays and potential access congestions
- The scope of these arrays is, like automatic scalar variables, **limited to individual threads**
- That is, a private version of each automatic array is created for and used by every thread
- Once a thread terminates its execution, the contents of its automatic array variables also cease to exist

CUDA DEVICE MEMORY TYPES

- If a variable declaration is preceded by the keyword **`_shared_`** it declares a shared variable in CUDA
- One can also add an optional **`_device_`** in front of **`_shared_`** in the declaration to achieve the same effect
- Such declaration typically resides within a kernel function or a device function
- Shared variables reside in shared memory. The scope of a shared variable is within a thread block
- A private version of the shared variable is created for and used by each thread block during kernel execution. The lifetime of a shared variable is within the duration of the kernel
- CUDA programmers often use shared variables to hold the portion of global memory data that are heavily used in an execution phase of a kernel

CUDA DEVICE MEMORY TYPES

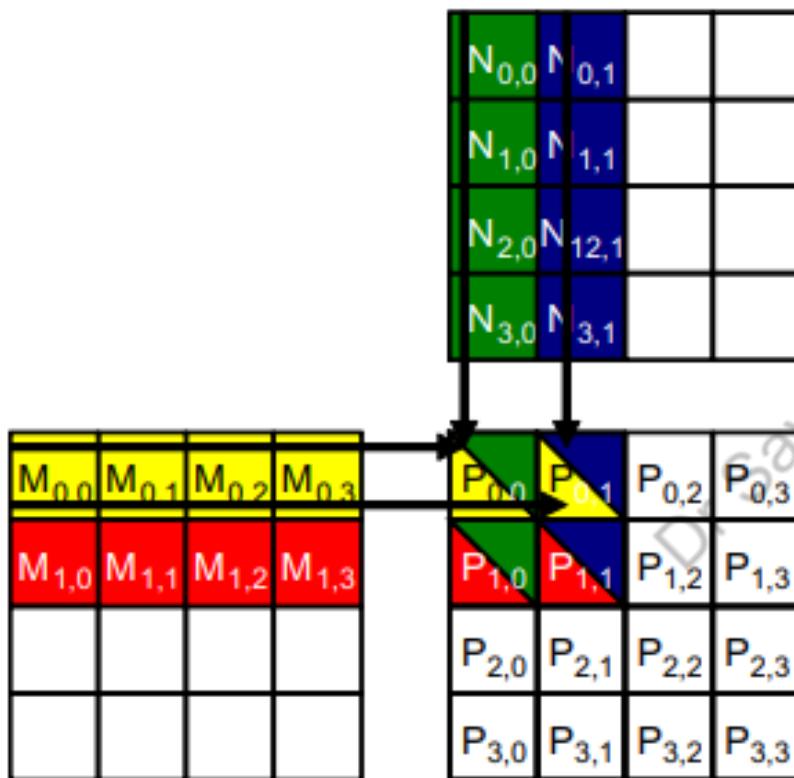
- If a variable declaration is preceded by the keyword **`_constant_`** it declares a constant variable in CUDA
- One can also add an optional **`_device_`** in front of **`_constant_`** to achieve the same effect
- Declaration of constant variables must be outside any function body
- The scope of a constant variable is all grids, meaning that all threads in all grids see the same version of a constant variable. The lifetime of a constant variable is the entire application execution
- Constant variables are often used for variables that provide input values to kernel functions. Constant variables are stored in the global memory but are cached for efficient access

A STRATEGY FOR REDUCING GLOBAL MEMORY TRAFFIC

- Trade-off in the use of device memories in CUDA:
 - Global memory is **large but slow**, whereas the shared memory is **small but fast**
- A common strategy is partition the data into subsets called **tiles** so that each tile fits into the shared memory
- An important criterion is that the kernel computation on these tiles can be done independently of each other

A STRATEGY FOR REDUCING GLOBAL MEMORY TRAFFIC

- Among the four threads highlighted, there is a significant overlap in terms of the M and N elements they access



Access order				
thread _{0,0}	M _{0,0} * N _{0,0}	M _{0,1} * N _{1,0}	M _{0,2} * N _{2,0}	M _{0,3} * N _{3,0}
thread _{0,1}	M _{0,0} * N _{0,1}	M _{0,1} * N _{1,1}	M _{0,2} * N _{2,1}	M _{0,3} * N _{3,1}
thread _{1,0}	M _{1,0} * N _{0,0}	M _{1,1} * N _{1,0}	M _{1,2} * N _{2,0}	M _{1,3} * N _{3,0}
thread _{1,1}	M _{1,0} * N _{0,1}	M _{1,1} * N _{1,1}	M _{1,2} * N _{2,1}	M _{1,3} * N _{3,1}

A STRATEGY FOR REDUCING GLOBAL MEMORY TRAFFIC

- Every M and N element is accessed exactly twice during the execution of block 0,0
- Therefore, if we can have all four threads to **collaborate in their accesses** to global memory, we can reduce the traffic to the global memory by half
- **Potential reduction** in global memory traffic in the matrix multiplication example is proportional to the **dimension of the blocks used**
- With $N \times N$ blocks, the potential reduction of global memory traffic would be N
- That is, if we use 16×16 blocks, one can potentially reduce the global memory traffic to $1/16$ through collaboration between threads

A STRATEGY FOR REDUCING GLOBAL MEMORY TRAFFIC

- When the rate of **DRAM requests** exceeds the **provisioned bandwidth** of the DRAM system, traffic congestion arises and the arithmetic units become idle
- If multiple threads access data from the same DRAM location, they can form a “carpool” and combine their accesses into one DRAM request
- This, however, requires the threads to have a **similar execution schedule** so that their data accesses can be combined into one

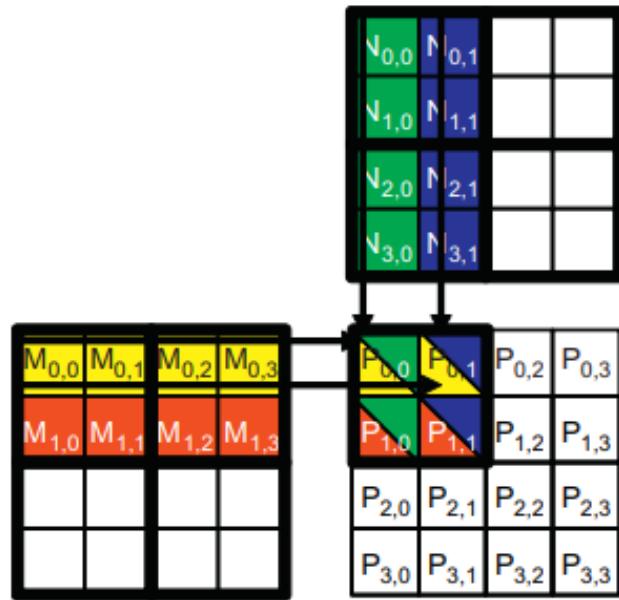
A TILED MATRIX MATRIX MULTIPLICATION KERNEL

- The basic idea is to have the **threads to collaboratively** load M and N elements into the **shared memory** before they individually use these elements in their dot product calculation
- The size of the shared memory is **quite small** and one must be careful not to exceed the capacity of the shared memory when loading these M and N elements into the shared memory
- This can be accomplished by dividing the M and N matrices into **smaller tiles**
- The size of these tiles is chosen so that they can fit into the shared memory. **In the simplest form, the tile dimensions equal those of the block**

A TILED MATRIX MATRIX MULTIPLICATION KERNEL

- We divide the M and N matrices into 2×2 tiles
- The dot product calculations performed by each thread are now divided into phases
- In each phase, all threads in a block collaborate to load a tile of M elements and a tile of N elements into the shared memory
- This is done by having every thread in a block to load one M element and one N element into the shared memory

A TILED MATRIX MATRIX MULTIPLICATION KERNEL



	Phase 1		Phase 2			
thread _{0,0}	M _{0,0} ↓ Mds _{0,0}	N _{0,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}	M _{0,2} ↓ Mds _{0,0}	N _{2,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}
thread _{0,1}	M _{0,1} ↓ Mds _{0,1}	N _{0,1} ↓ Nds _{1,0}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}	M _{0,3} ↓ Mds _{0,1}	N _{2,1} ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}
thread _{1,0}	M _{1,0} ↓ Mds _{1,0}	N _{1,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}	M _{1,2} ↓ Mds _{1,0}	N _{3,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}
thread _{1,1}	M _{1,1} ↓ Mds _{1,1}	N _{1,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}	M _{1,3} ↓ Mds _{1,1}	N _{3,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}

time →

A TILED MATRIX MATRIX MULTIPLICATION KERNEL

- After the two tiles of M and N elements are loaded into the shared memory, these values are used in the calculation of the dot product
- **Each value in the shared memory is used twice**
- For example, the M_{1,1} value, loaded by thread_{1,1} into M_{d1,1}, is used twice, once by thread_{0,1} and once by thread_{1,1}
- By loading each global memory value into shared memory so that it can be used multiple times, we reduce the number of accesses to the global memory
- In this case, we reduce the number of accesses to the global memory by half

A TILED MATRIX MATRIX MULTIPLICATION KERNEL

- If an input matrix is of dimension **N** and the tile size is **TILE_WIDTH**, the dot product would be performed in **N/TILE_WIDTH** phases
- The creation of these phases is key to the reduction of accesses to the global memory
- Note also that Mds and Nds are reused to hold the input values
- In each phase, the same Mds and Nds are used to hold the subset of M and N elements used in the phase
- This allows a much **smaller shared memory** to serve most of the accesses to global memory
- This is due to the fact that each phase focuses on a small subset of the input matrix elements. Such focused access behavior is called **locality**

A TILED MATRIX MATRIX MULTIPLICATION KERNEL

- Locality, there is an opportunity to use small, high-speed memories to serve most of the accesses and remove these accesses from the global memory
- Locality is important for achieving high performance in multicore CPUs as in many-thread GPUs

A TILED MATRIX MATRIX MULTIPLICATION KERNEL

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
                                int Width) {

    1. __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    2. __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    3. int bx = blockIdx.x; int by = blockIdx.y;
    4. int tx = threadIdx.x; int ty = threadIdx.y;

        // Identify the row and column of the d_P element to work on
    5. int Row = by * TILE_WIDTH + ty;
    6. int Col = bx * TILE_WIDTH + tx;

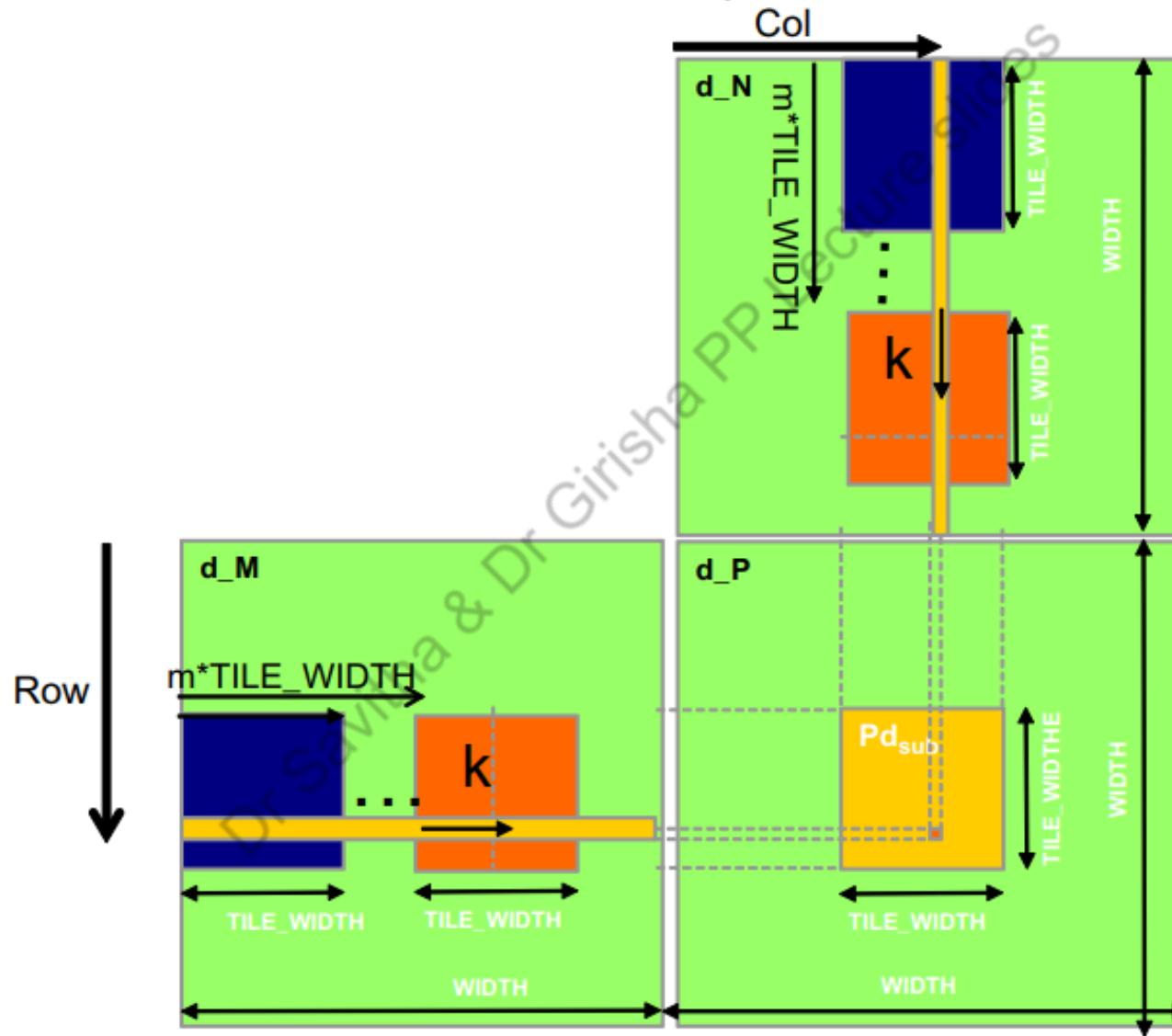
    7. float Pvalue = 0;
        // Loop over the d_M and d_N tiles required to compute d_P element
    8. for (int m = 0; m < Width/TILE_WIDTH; ++m) {

            // Collaborative loading of d_M and d_N tiles into shared memory
    9.     Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
    10.    Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];
    11.    __syncthreads();

    12.    for (int k = 0; k < TILE_WIDTH; ++k) {
    13.        Pvalue += Mds[ty][k] * Nds[k][tx];
    14.    }
    15.    __syncthreads();
    }

    d_P[Row*Width + Col] = Pvalue;
}
```

A TILED MATRIX MATRIX MULTIPLICATION KERNEL



MEMORY AS A LIMITING FACTOR TO PARALLELISM

- CUDA registers and shared memory can be extremely effective in reducing the number of accesses to global memory
 - Should not exceed the capacity of these memories
- These memories are forms of **resources** that are needed for thread execution
- Each CUDA device offers a limited amount of resources, which limits the number threads that can simultaneously reside in the SM for a given application
- In general, the more resources each thread requires, the fewer the number of threads can reside in each SM, and thus the fewer number of threads that can reside in the entire device

MEMORY AS A LIMITING FACTOR TO PARALLELISM

- Assume that in a device D, each SM can accommodate up to 1,536 threads and has 16,384 registers
- To support 1,536 threads, each thread can use only $16,384/1,536 = 10$ registers
- If each thread uses 11 registers, the number of threads able to be executed concurrently in each SM will be reduced
 - Such reduction is done at the **block granularity/block level**

MEMORY AS A LIMITING FACTOR TO PARALLELISM

- If each block contains 512 threads, the reduction of threads will be done by reducing **512 threads at a time**
- Thus, the next lower number of threads from 1,536 would be 512, a one-third reduction of threads that can simultaneously reside in each SM
- The number of registers available to each SM varies from device to device
- An application can dynamically determine the number of registers available in each SM of the device used and choose a version of the kernel that uses the number of registers appropriate for the device
- This can be done by calling the **cudaGetDeviceProperties()**
- **dev_prop.regsPerBlock** gives the number of registers available in each SM

MEMORY AS A LIMITING FACTOR TO PARALLELISM

- Shared memory usage can also limit the number of threads assigned to each SM
- Assume device D has 16,384 (16 K) bytes of shared memory in each SM
- Shared memory is used by blocks
- Assume that each SM can accommodate up to eight blocks
- To reach this maximum, each block must not use more than 2K bytes of shared memory
- If each block uses more than 2K bytes of memory, the number of blocks that can reside in each SM is such that the total amount of shared memory used by these blocks does not exceed 16 K bytes
- For example, if each block uses 5K bytes of shared memory, no more than three blocks can be assigned to each SM

Performance Considerations: Warps And Thread Execution

- The execution speed of a CUDA kernel can vary greatly depending on the **resource constraints of the device** being used
- Launching a CUDA kernel generates a **grid of threads** that are organized as a **two-level hierarchy**
- At the top level, a grid consists of a 1D, 2D, or 3D **array of blocks**
- At the bottom level, each block, in turn, consists of a 1D, 2D, or 3D **array of threads**
- Blocks can execute in any order relative to each other, which allows for transparent scalability in parallel execution of CUDA kernels

Warps And Thread Execution

- Threads in a block can execute in any order with respect to each other
- **Barrier synchronizations** should be used whenever we want to ensure all threads have completed a common phase of their execution before any of them start the next phase
- The correctness of executing a kernel should not depend on the fact that certain threads will execute in synchrony with each other
- Each thread block is partitioned into **warps**

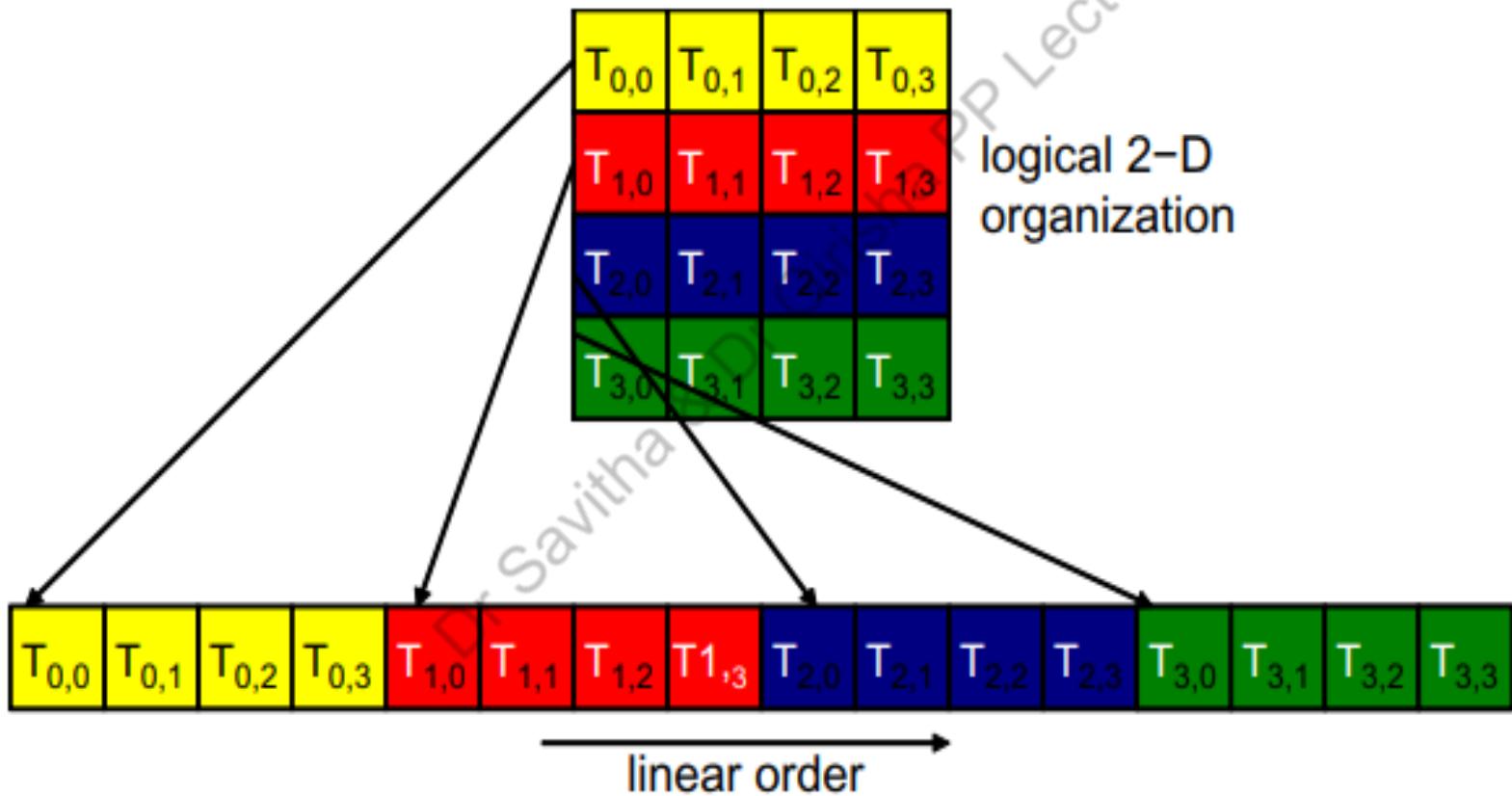
Warps And Thread Execution

- The size of a warp can easily vary from implementation to implementation
- Thread blocks are partitioned into warps based on **thread indices**
- If a thread block is organized into a 1D array (i.e., only `threadIdx.x` is used), the partition is straightforward; `threadIdx.x` values within a warp are consecutive and increasing
- For a warp size of 32, warp 0 starts with thread 0 and ends with thread 31, warp 1 starts with thread 32 and ends with thread 63
- Warp n starts with thread $32 \times n$ and ends with thread $32(n + 1) - 1$

Warps And Thread Execution

- For a block of which the size is not a multiple of 32, the last warp will be padded with extra threads to fill up the 32 threads
- For example, if a block has 48 threads, it will be partitioned into two warps, and its warp 1 will be padded with 16 extra threads
- For blocks that consist of multiple dimensions of threads, the dimensions will be projected into a linear order before partitioning into warps

Warps And Thread Execution



Warps And Thread Execution

- For a 3D block, we first place all threads of which the threadIdx.z value is 0 into the linear order
- Among these threads, they are treated as a 2D block as shown in Figure
- All threads of which the threadIdx.z value is 1 will then be placed into the linear order, and so on
- The SIMD hardware executes all threads of a warp as a **bundle**
- **An instruction** is run for all threads in the same warp. It works well when all threads within a warp follow the same execution path, or control flow, when working their data

Warps And Thread Execution

- For example, for an **if-else construct**, the execution works well when either all threads execute the if part or all execute the else part
- When threads within a warp take different control flow paths, the SIMD hardware will take **multiple passes** through these divergent paths
- One pass executes those threads that follow the **if** part and another pass executes those that follow the **else** part
- During each pass, the threads that follow the other path are not allowed to take effect
- These passes are **sequential to each other**, thus they will add to the execution time

Warps And Thread Execution

- When threads in the same warp follow different paths of control flow, we say that these **threads diverge in their execution**
- The cost of divergence is the extra pass the hardware needs to take to allow the threads in a warp to make their own decisions
- Another example: Loops
- A control construct can result in thread divergence when its decision condition is based on threadIdx values
- For example, the statement **if (threadIdx.x > 2) {}**

Reduction algorithm

- A reduction algorithm derives a **single** value from an **array of values**
- The single value could be the sum, the maximal value, the minimal value, etc. among all elements
- A reduction can be easily done by sequentially going through every element of the array
- When an element is visited, the action to take depends on the type of **reduction** being performed
- For a **sum reduction**, the value of the element being visited at the current step, or the current value, is added to a running sum

Reduction algorithm

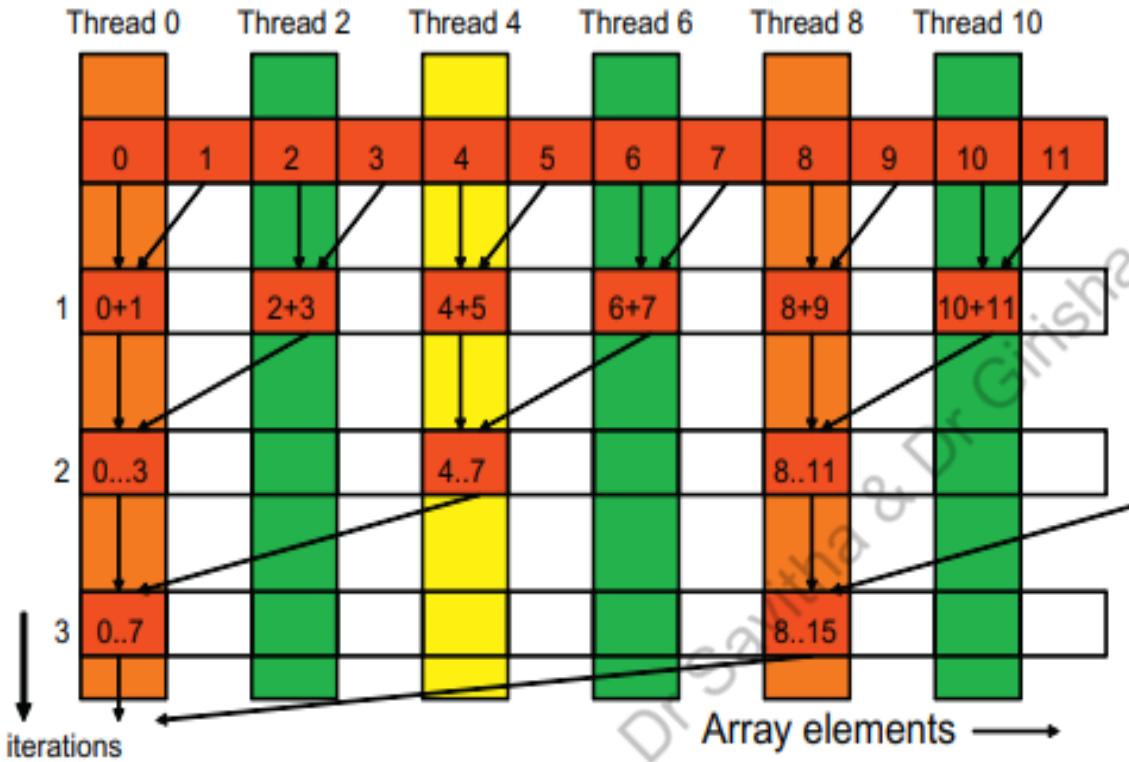
- For a **maximal reduction**, the current value is compared to a running maximal value of all the elements visited so far
- If the current value is larger than the running maximal, the current element value becomes the running maximal value
- For a **minimal reduction**, the value of the element currently being visited is compared to a running minimal
- If the current value is smaller than the running minimal, the current element value becomes the running minimal
- The sequential algorithm ends when all the elements are visited
- The sequential reduction algorithm is **work-efficient**. Every element is visited once and only a minimal amount of work is performed when each element is visited
- Its execution time is proportional to the number of elements involved

Reduction algorithm

- The original array is in the **global memory**
- Each **thread block** reduces a section of the array by loading the elements of the section into the shared memory and performing parallel reduction
- The reduction is done in place, which means the elements in the shared memory will be replaced by partial sums

```
1. __shared__ float partialSum[]  
...  
2. unsigned int t = threadIdx.x;  
3. for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)  
4. {  
5.     __syncthreads();  
6.     if (t % (2*stride) == 0)  
7.         partialSum[t] += partialSum[t+stride];  
8 }
```

Reduction algorithm



```
1. __shared__ float partialSum[]  
...  
2. unsigned int t = threadIdx.x;  
3. for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)  
4. {  
5.     __syncthreads();  
6.     if (t % (2*stride) == 0)  
7.         partialSum[t] += partialSum[t+stride];  
8. }
```

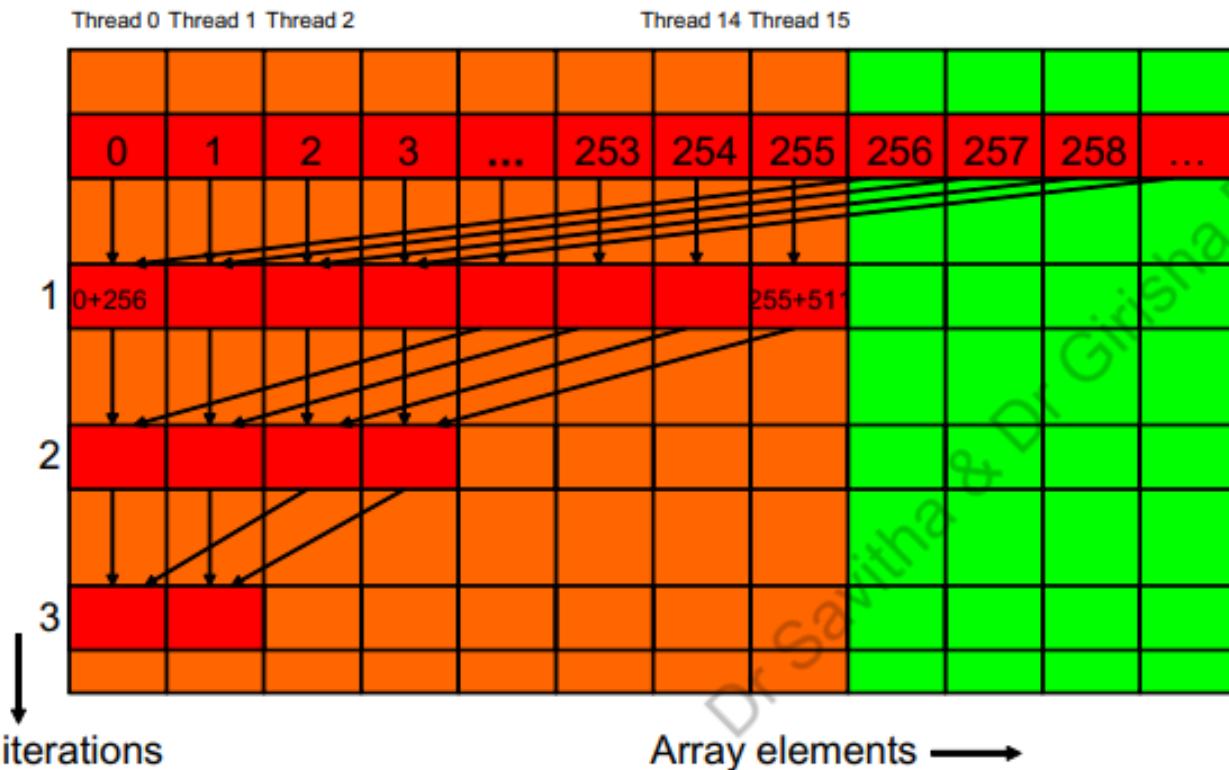
Reduction algorithm

- By using blockDim.x as the loop bound in line 4, the kernel assumes that it is launched with the same number of threads as the number of elements in the section
- Assume that the total number of elements to be reduced is N
 - The first round requires $N/2$ additions
 - The second round requires $N/4$ additions
 - The final round has only one addition
 - There are $\log_2(N)$ rounds
 - The total number of additions performed by the kernel is
 $N/2 + N/4 + N/8 + \dots + 1 = N - 1$

Reduction algorithm

- Therefore, the computational complexity of the reduction algorithm is $O(N)$
- During the first iteration of the loop, only those threads of which the `threadIdx.x` are even will execute the add statement
- One pass will be needed to execute these threads and one additional pass will be needed to execute those that do not execute line 8
- In each successive iteration, fewer threads will execute line 8 but two passes will be still needed to execute all the threads during each iteration

Reduction algorithm



```
1. __shared__ float partialSum[]  
2. unsigned int t = threadIdx.x;  
3. for (unsigned int stride = blockDim.x; stride > 1; stride /= 2)  
4. {  
5.     __syncthreads();  
6.     if (t < stride)  
7.         partialSum[t] += partialSum[t+stride];  
8. }
```

Reduction algorithm

- Instead of adding neighbor elements in the first round, it adds elements that are half a section away from each other
- It does so by initializing the stride to be half the size of the section
- After the first iteration, all the pairwise sums are stored in the first half of the array
- The loop divides the stride by 2 before entering the next iteration

Reduction algorithm

- During the first iteration, all threads of which the `threadIdx.x` values are less than half of the size of the section execute line 7
- For a section of 512 elements, threads 0-255 execute the add statement during the first iteration
- While threads 255-511 do not
- The pairwise sums are stored in elements 0-255 after the first iteration
- Since the warps consist of 32 threads with consecutive `threadIdx.x` values, all threads in warps **0-7** execute the add statement, whereas warps **8-15** all skip the add statement
- **Since all threads in each warp take the same path, there is no thread divergence**

GLOBAL MEMORY BANDWIDTH

- One of the most important factors of CUDA kernel performance is accessing data in the global memory
- CUDA applications exploit **massive data parallelism**
- CUDA applications tend to process a massive amount of data from the global memory within a short period
- **Tiling techniques** utilize shared memories to reduce the total amount of data that must be accessed by a collection of threads in the thread block
- **Memory coalescing** techniques can more effectively move data from the global memory into shared memories and registers

GLOBAL MEMORY BANDWIDTH

- The global memory of a CUDA device is implemented with **DRAMs (Dynamic Random Access Memory)**
- Data bits are stored in **DRAM cells** that are **small capacitors**, where the presence or absence of a tiny amount of electrical charge distinguishes between 0 and 1
- Reading data from a DRAM cell requires the small capacitor to use its tiny electrical charge to drive a highly capacitive line leading to a **sensor**
- It will set off its detection mechanism that determines whether a **sufficient amount of charge** is present in the capacitor to qualify as a “1”
- This process takes **tens of nanoseconds** in modern DRAM chips
- Because this is a very slow process relative to the desired data access speed (sub-nanosecond access per byte), modern DRAMs use parallelism to increase their rate of data access

GLOBAL MEMORY BANDWIDTH

- Each time a DRAM location is accessed, many **consecutive locations** that include the requested location are accessed
- **Many sensors** are provided in each DRAM chip and they work in parallel
- Each senses the content of a bit within these consecutive locations
- Once detected by the sensors, the data from all these consecutive locations can be transferred at very high speed to the processor
- If an application can make **focused use of data from consecutive locations**, the DRAMs can supply the data at a much higher rate than if a truly random sequence of locations were accessed

GLOBAL MEMORY BANDWIDTH

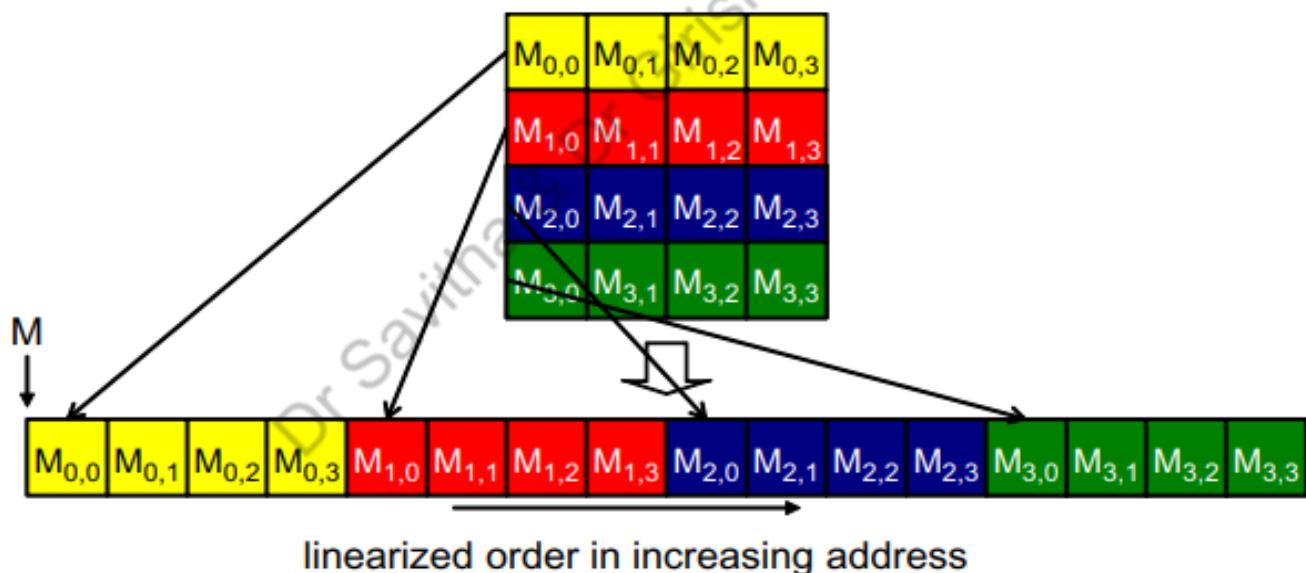
- Current CUDA devices employ a technique that allows the programmers to achieve high global memory access efficiency by **organizing memory accesses of threads into favorable patterns**
- **This technique takes advantage of the fact that threads in a warp execute the same instruction at any given point in time**
- When all threads in a warp execute a load instruction, the hardware detects whether they access consecutive global memory locations
 - The most favorable access pattern is achieved when all threads in a warp access consecutive global memory locations
- In this case, the hardware **combines, or coalesces**, all these accesses into a consolidated access to consecutive DRAM locations

GLOBAL MEMORY BANDWIDTH

- For example, for a given load instruction of a warp, if thread 0 accesses global memory location N , thread 1 location $N + 1$, thread 2 location $N + 2$, and so on, all these accesses will be coalesced, or combined into a single request for consecutive locations when accessing the DRAMs
- Such coalesced access allows the DRAMs to deliver data at a rate close to the **peak global memory bandwidth**

GLOBAL MEMORY BANDWIDTH

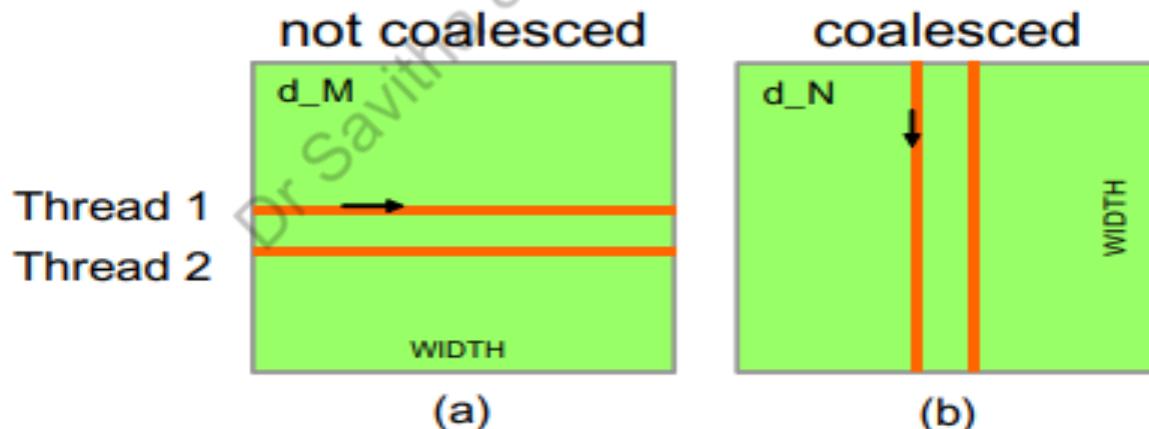
- Multidimensional array elements in C and CUDA are placed into the linearly addressed memory space according to the **row-major convention**



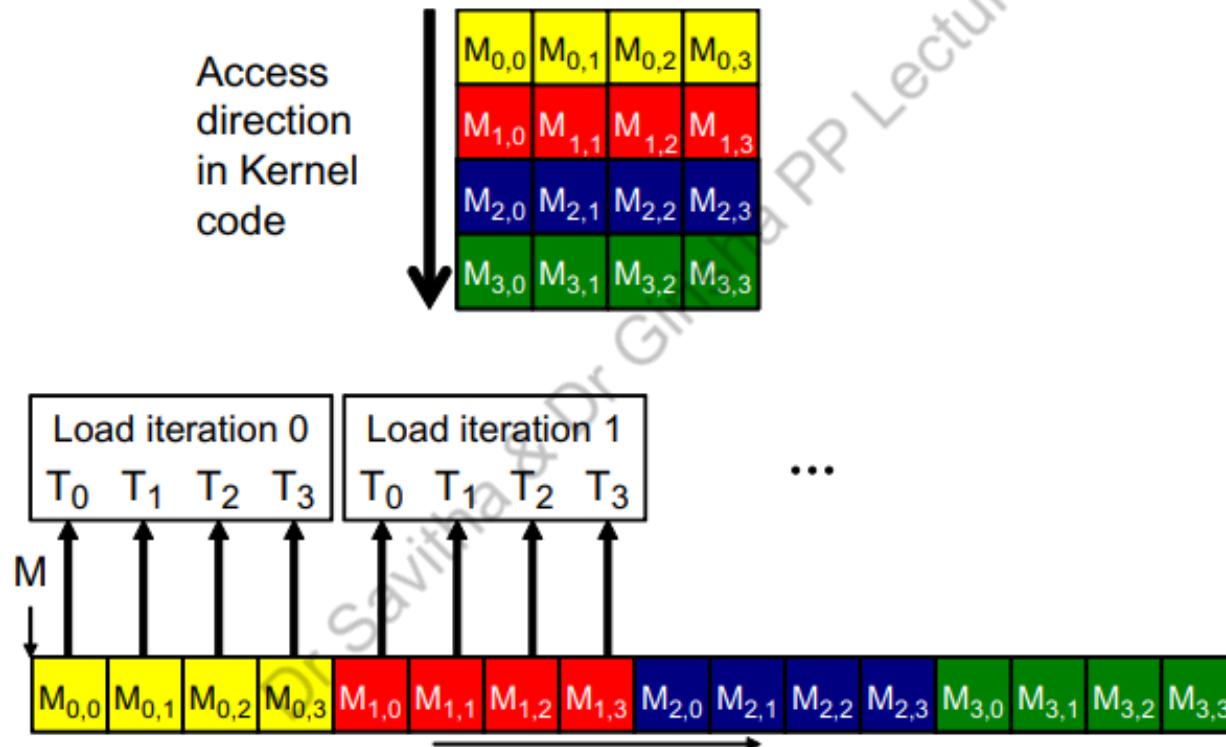
GLOBAL MEMORY BANDWIDTH

Favorable versus unfavorable CUDA kernel 2D row-major array data access patterns for memory coalescing

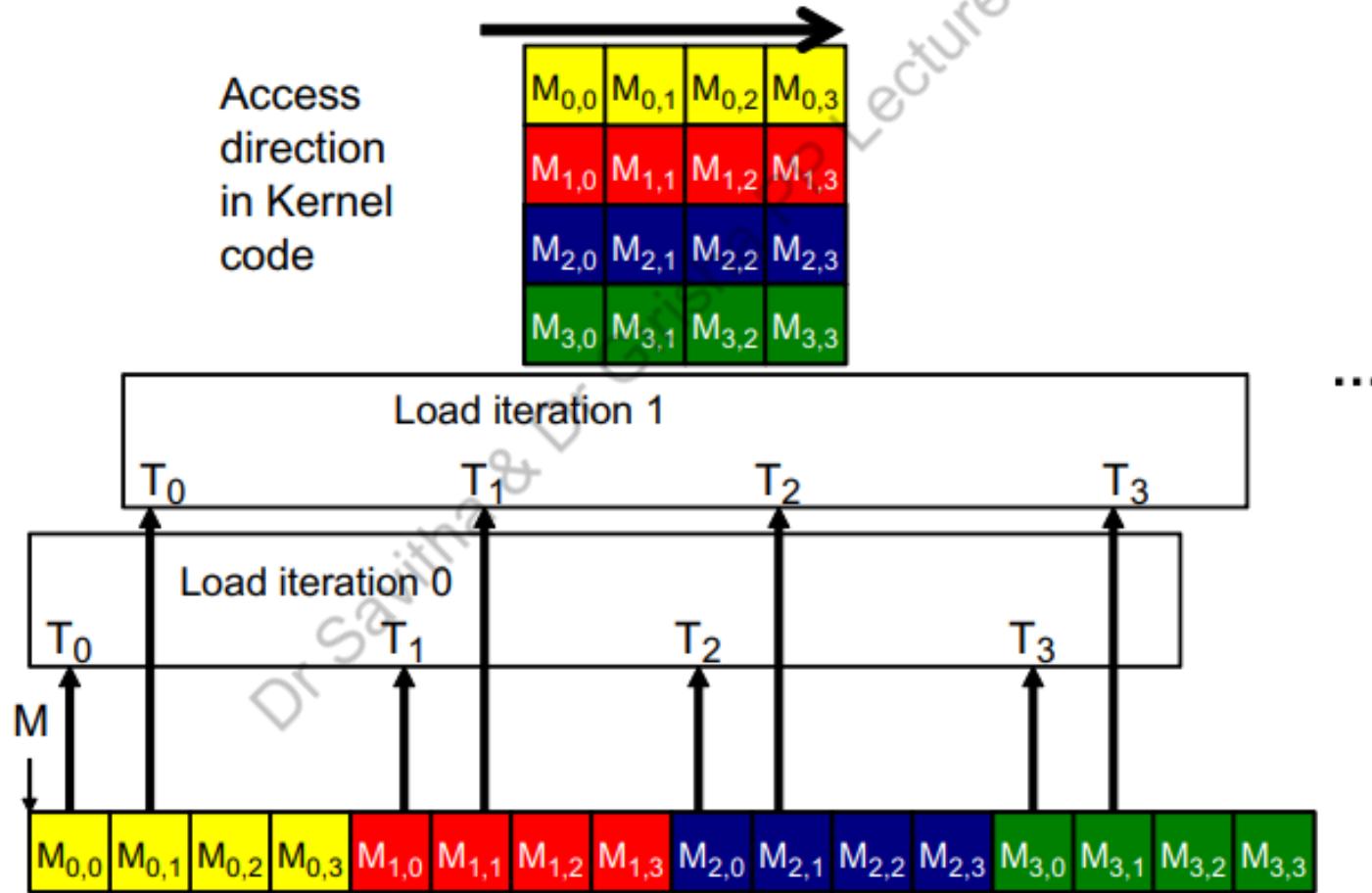
- During iteration 0, threads in a warp read element 0 of rows 0-31. During iteration 1, these same threads read element 1 of rows 0-31. None of the accesses will be coalesced
- During iteration 0, threads in warp 0 read element 1 of columns 0-31. All these accesses will be coalesced



GLOBAL MEMORY BANDWIDTH



GLOBAL MEMORY BANDWIDTH



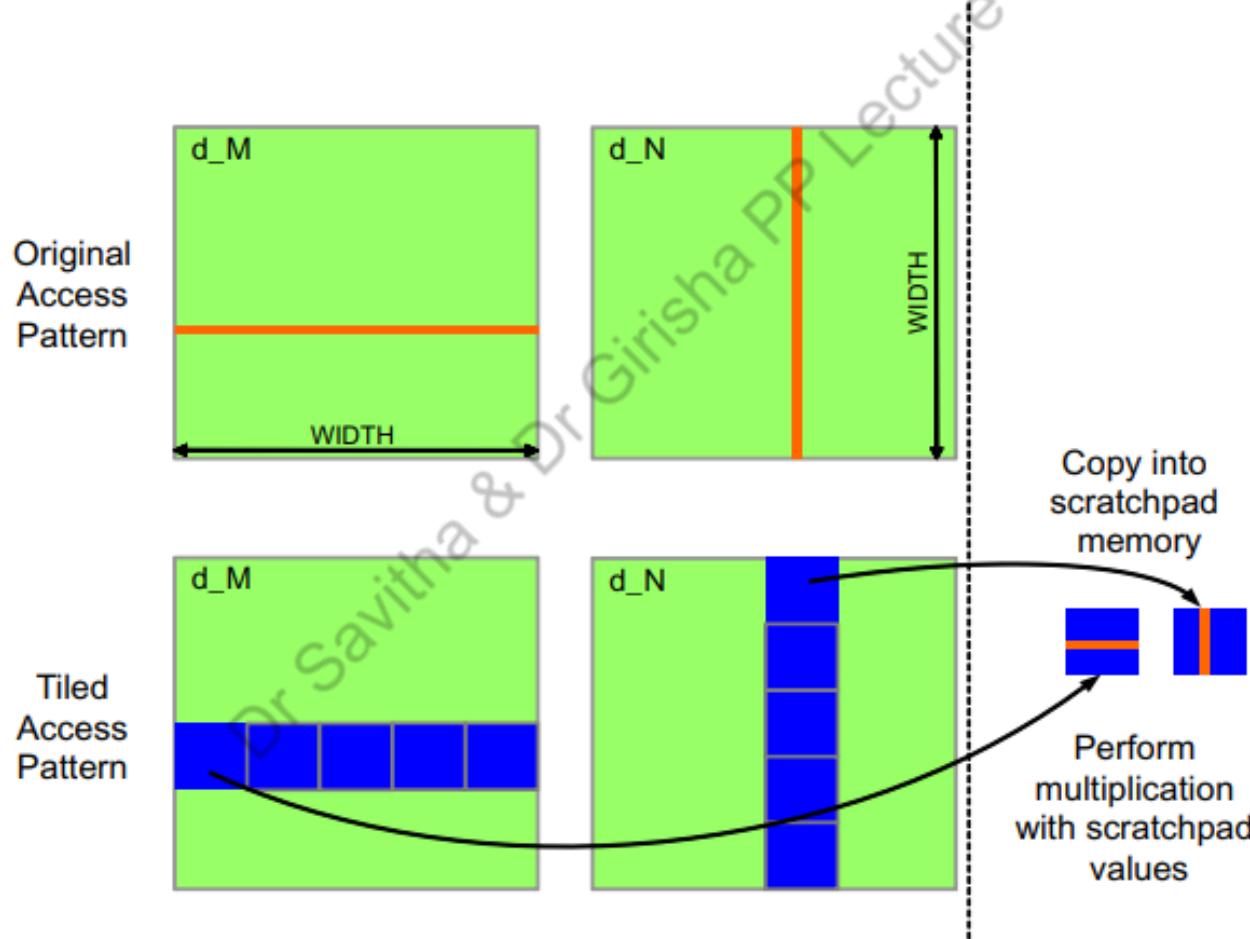
GLOBAL MEMORY BANDWIDTH

- In a realistic matrix, there are typically hundreds or even thousands of elements in each dimension
- The elements accessed in each iteration by neighboring threads can be hundreds or even thousands of elements apart
- The hardware will determine that accesses to these elements are far away from each other and **cannot be coalesced**
- As a result, when a kernel loop iterates through a row, the accesses to global memory are much less efficient than the case where a kernel iterates through a column

GLOBAL MEMORY BANDWIDTH

- If an algorithm intrinsically requires a kernel code to iterate through data along the row direction, one can use the **shared memory to enable memory coalescing**
- Each thread reads a row from d_M (**Cannot be coalesced**)
- A **tiled algorithm** can be used to enable coalescing
- Threads of a block can first cooperatively load the tiles into the shared memory
- Care must be taken to ensure that these tiles are **loaded in a coalesced pattern**

GLOBAL MEMORY BANDWIDTH



GLOBAL MEMORY BANDWIDTH

- Once the data is in shared memory, it can be accessed either on a **row basis** or a **column basis** with much less performance variation because the shared memories are implemented as intrinsically **high-speed, on-chip memory** that does not require coalescing to achieve a high data access rate
- The index calculation `d_M[row][m*TILE_SIZE + tx]` makes these threads access elements in the same row
- Elements in the same row are placed into consecutive locations of the global memory

GLOBAL MEMORY BANDWIDTH

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
1. __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2. __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

3. int bx = blockIdx.x; int by = blockIdx.y;
4. int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the d_P element to work on
5. int Row = by * TILE_WIDTH + ty;
6. int Col = bx * TILE_WIDTH + tx;

7. float Pvalue = 0;
// Loop over the d_M and d_N tiles required to compute the d_P element
8. for (int m = 0; m < Width/TILE_WIDTH; ++m) {

// Collaborative loading of d_M and d_N tiles into shared memory
9.     Mds[tx][ty] = d_M[Row*Width + m*TILE_WIDTH+tx];
10.    Nds[tx][ty] = d_N[(m*TILE_WIDTH+ty)*Width + Col];
11.    __syncthreads();

12.    for (int k = 0; k < TILE_WIDTH; ++k)
13.        Pvalue += Mds[tx][k] * Nds[k][ty];
14.    __syncthreads();
}
15. d_P[Row*Width+Col] = Pvalue;
}
```

GLOBAL MEMORY BANDWIDTH

- In the case of d_N , the row index $m*TILE_SIZE + ty$ has the same value for all threads in the same warp
- They all have the same ty value
- Thus, threads in the same warp access the same row
- The question is whether the adjacent threads in a warp access adjacent elements of a row?
- Note that the column index calculation for each thread Col is based on $bx*TILE_SIZE + tx$
- Therefore, adjacent threads in a warp access adjacent elements in a row
- The hardware detects that these threads in the same warp access consecutive locations in the global memory and combine them into a coalesced access

DYNAMIC PARTITIONING OF EXECUTION RESOURCES

- The execution resources in a streaming multiprocessor (SM) includes:
 - **Registers, Shared memory, Thread block slots, and Thread slots**
- These resources are dynamically partitioned and assigned to threads to support their execution
- The current generation of devices have **1,536 thread slots**, each of which can accommodate one thread
- These thread slots are partitioned and assigned to thread blocks during runtime

DYNAMIC PARTITIONING OF EXECUTION RESOURCES

- If each thread block consists of 512 threads, the 1,536 thread slots are partitioned and assigned to three blocks
- Each SM can accommodate up to three thread blocks due to **limitations on thread slots**
- If each thread block contains 128 threads, the 1,536 thread slots are partitioned and assigned to 12 thread blocks
- The ability to **dynamically partition** the thread slots among thread blocks makes SMs versatile
- They can either execute **many thread blocks** each having **few threads**, or execute **few thread blocks** each having **many threads**

DYNAMIC PARTITIONING OF EXECUTION RESOURCES

- This is in contrast to a **fixed partitioning** method where each block receives a fixed amount of resources regardless of their real needs
- Fixed partitioning results in **wasted thread slots** when a block has few threads and fails to support blocks that require more thread slots than the fixed partition allows
- Dynamic partitioning of resources can lead to **subtle interactions between resource limitations**, which can cause underutilization of resources
 - Such interactions can occur between block slots and thread slots

DYNAMIC PARTITIONING OF EXECUTION RESOURCES

- For example, if each block has 128 threads, the 1,536 thread slots can be partitioned and assigned to 12 blocks
- However, since there are only 8 block slots in each SM, only 8 blocks will be allowed
- This means that only 1,024 of the thread slots will be utilized
- Therefore, to fully utilize both the block slots and thread slots, one needs at least 256 threads in each block

DYNAMIC PARTITIONING OF EXECUTION RESOURCES

- The automatic variables declared in a CUDA kernel are placed into registers
- Some kernels may use lots of automatic variables and others may use few of them
- Thus, one should expect that some kernels require many registers and some require fewer
- By dynamically partitioning the registers among blocks, the SM can accommodate more blocks if they require few registers and fewer blocks if they require more registers
- One does, however, need to be aware of potential interactions between register limitations and other resource limitations

DYNAMIC PARTITIONING OF EXECUTION RESOURCES

- In the matrix multiplication example, assume that each SM has 16,384 registers and the kernel code uses 10 registers per thread. If we have 16×16 thread blocks, how many threads can run on each SM?

DYNAMIC PARTITIONING OF EXECUTION RESOURCES

- First calculating the number of registers needed for each block, which is $10 \times 16 \times 16 = 2,560$
- The number of registers required by six blocks is 15,360, which is under the 16,384 limit
- Adding another block would require 17,920 registers, which exceeds the limit
- Therefore, the register limitation allows blocks that altogether have 1,536 threads to run on each SM

DYNAMIC PARTITIONING OF EXECUTION RESOURCES

- Now assume that the programmer declares another two automatic variables in the kernel and bumps the number of registers used by each thread to 12
- $12 \times 16 \times 16 = 3,072$ registers
- The number of registers required by six blocks is now 18,432, which exceeds the register limitation
- The CUDA runtime system deals with this situation by reducing the number of blocks assigned to each SM by one

DYNAMIC PARTITIONING OF EXECUTION RESOURCES

- This however, reduces the number of threads running on an SM from 1,536 to 1,280
- That is, by using two extra automatic variables, the program saw a **one-sixth reduction** in the warp parallelism in each SM
- This is sometimes referred to as a “**performance cliff**” where a slight increase in resource usage can result in significant reduction in parallelism

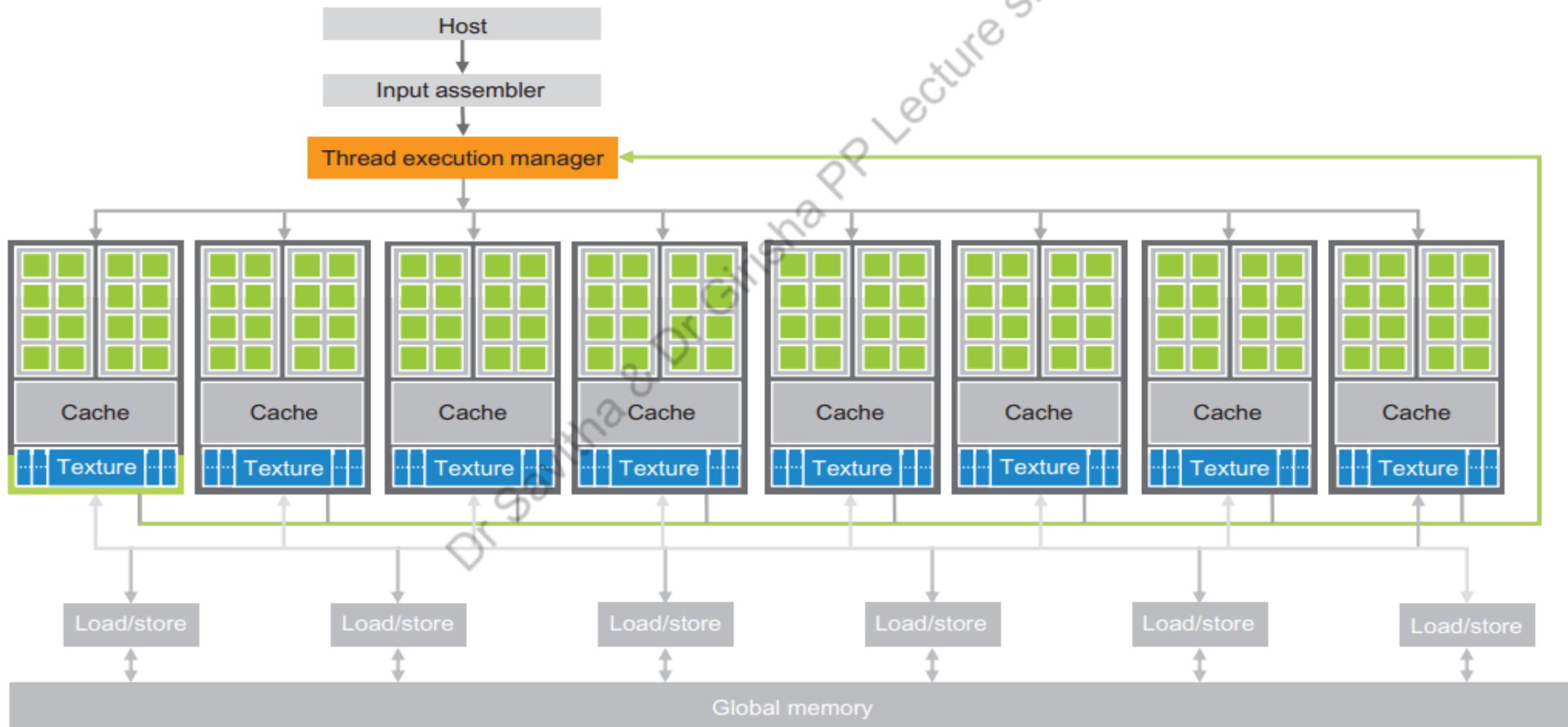
ARCHITECTURE OF A MODERN GPU

- It is organized into an array of highly threaded streaming multiprocessors (SMs)
- The number of SMs in a building block can vary from one generation of CUDA GPUs to another generation
- SM has a number of **streaming processors** (SPs) that share **control logic and an instruction cache**
- Each GPU currently comes with multiple gigabytes of Graphic Double Data Rate (GDDR) DRAM, referred to as global memory

ARCHITECTURE OF A MODERN GPU

- These GDDR DRAMs differ from the system DRAMs on the CPU motherboard
 - They are essentially the **frame buffer** memory that is used for graphics
 - For graphics applications, they hold **video images and texture information** for 3D rendering
 - But for computing, they function as very high bandwidth off-chip memory, though with somewhat longer latency than typical system memory
 - For massively parallel applications, the higher bandwidth makes up for the longer latency

ARCHITECTURE OF A MODERN GPU



END

Dr Savitha & Dr Girisha PP Lecture slides

OpenMP Performance considerations

Dr Savitha & Dr Girishan P Lecture Slides

Introduction

- It may be possible to quickly write a correctly functioning OpenMP program
- But not so easy to create a program that provides the **desired level of performance**
 - It is often because some **basic programming rules** have not been adhered to
- Programmers have developed some rule of thumb for writing efficient sequential code
 - Guarantees certain base level performance
 - This can also be extended to OpenMP programs
- Best practice: **Write an efficient sequential program. Then introduce OpenMP constructs**

Performance Considerations for Sequential Programs

- Poor single-processor performance is often caused by the suboptimal usage of cache memory
- **Cache-miss** on highest level of memory hierarchy is expensive
 - **5-10 times** more expensive than fetching the data from the cache
 - **Higher frequency- poor program performance**
- In a shared memory systems:
 - The adverse effect is more
 - More number of threads are involved
 - **A cache-miss: results in additional traffic on the system interconnect**
 - No systems in the market has interconnect with sufficient bandwidth

Memory Access Patterns

- **Memory Hierarchy:**
 - The **largest**, and also **slowest**, part of memory is known as **main memory**
 - Main memory is organized into pages, a subset of which will be available to a given application
 - The memory levels closer to the processor are **successively smaller and faster** and are collectively known as **cache**
- When the program is compiled, the compiler will arrange its data objects to be stored in the main memory
- They will be transferred to cache when needed

Memory Access Patterns

- If the data requested is not present in cache, its known as Cache-miss
- It must be **retrieved** from higher levels of the memory hierarchy
- Program data is brought into cache in chunks called **blocks**
- Data that is already in cache may need to be removed, or “**evicted**”, to make space for a new block of data
- **Memory hierarchy cannot be programmed by the programmer or the compiler**
- **We can only control the data fetched into the cache and evicted from the cache**
 - Reduce the frequency with which this situation occurs

Memory Access Patterns

- A major goal is to organize data accesses so that values are used as often as possible while they are still in cache
- Example: Let's consider Arrays
 - C typically specify that the elements of arrays be stored contiguously in memory
 - Thus, if an array element is fetched into cache, “nearby” elements of the array will be in the same cache block and will be fetched as part of the same transaction
 - If a computation that uses any of these values can be performed while they are still in cache, it will be beneficial for performance

Loop Optimizations

- If we were to encounter the first implementation of the loop in a piece of code, we could simply exchange the order of the loop headers and most likely experience a significant performance benefit
- This strategy is called **loop interchange** (or loop exchange)

```
for (int j=0; j<n; j++)
    for (int i=0; i<n; i++)
        sum += a[i] [j];
```



```
for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
        sum += a[i] [j];
```

Loop Optimizations

- Since many programs spend much of their time executing loops
 - Array access
- A suitable reorganization of the computation in loop nests to exploit cache can significantly improve a program's performance
- A programmer should consider transforming a loop
 - If accesses to arrays in the loop nest do not occur in the order in which they are stored in memory
 - If a loop has a large body and the references to an array element or its neighbors are far apart

Loop Optimizations

- They can be applied if the changes to the code do not affect correct execution of the program

If any memory location is referenced more than once in the loop nest and if at least one of those references modifies its value, then their relative ordering must not be changed by the transformation

Loop Optimizations

- Loop transformations have other purposes:
 - They may help the compiler to better utilize the instruction pipeline or may increase the amount of exploitable parallelism
 - They can also be applied to increase the size of parallel regions
- **Loop unrolling**
 - Loop unrolling, also known as loop unwinding, is a loop transformation technique that attempts to optimize a program's execution speed at the expense of its binary size, which is an approach known as space–time tradeoff.
 - Powerful technique to effectively reduce the overheads of loop execution
 - Loop unrolling can help to improve cache line utilization by improving data reuse
 - It can also help to increase the instruction-level parallelism

Loop Optimizations

```
for (int i=1; i<n; i++) {  
    a[i] = b[i] + 1;  
    c[i] = a[i] + a[i-1] + b[i-1];  
}
```



```
for (int i=1; i<n; i+=2) {  
    a[i] = b[i] + 1;  
    c[i] = a[i] + a[i-1] + b[i-1];  
    a[i+1] = b[i+1] + 1;  
    c[i+1] = a[i+1] + a[i] + b[i];  
}
```

- In this example, the loop body executes 2 iterations in one pass
- This number is called the “**unroll factor**”
- A higher value tends to give better performance but also increases the number of registers needed
- **If the unroll factor does not divide the iteration count, the remaining iterations must be performed outside this loop nest**
- **This is implemented through a second loop, the “cleanup” loop**

Loop Optimizations

- **Unroll and jam** is an extension of loop unrolling that is appropriate for some loop nests with multiple loops

```
for (int j=0; j<n; j++)
    for (int i=0; i<n; i++)
        a[i][j] = b[i][j] + 1;
```



```
for (int j=0; j<n; j+=2){
    for (int i=0; i<n; i++)
        a[i][j] = b[i][j] + 1;
    for (int i=0; i<n; i++)
        a[i][j+1] = b[i][j+1] + 1;
}
```



```
for (int j=0; j<n; j+=2)
    for (int i=0; i<n; i++) {
        a[i][j] = b[i][j] + 1;
        a[i][j+1] = b[i][j+1] + 1;
    }
```

Loop Optimizations

- Can this loops be optimized using **loop Interchange**?

```
for (int j=0; j<n; j++)
    for (int i=0; i<m; i++)
        a[i][j+1] = a[i+1][j] + b;
```

Loop Optimizations

- **Loop fusion** merges two or more loops to create a bigger loop
- This might enable data in cache to be reused more frequently
- May increase the amount of computation per iteration in order to improve the instruction-level parallelism

```
for (int i=0; i<n; i++)
    a[i] = b[i] * 2;
for (int i=0; i<n; i++)
{
    x[i] = 2 * x[i];
    c[i] = a[i] + 2;
}
```



```
for (int i=0; i<n; i++)
{
    a[i] = b[i] * 2;
    c[i] = a[i] + 2;
    x[i] = 2 * x[i];
}
```

Loop Optimizations

- **Loop fission** is a transformation that breaks up a loop into several loops
- Sometimes, we may be able to improve use of cache this way
- Isolate a part that inhibits full optimization of the loop
- This technique is likely to be most useful if a loop nest is large and its data does not fit into cache or if we can optimize parts of the loop in different ways

Loop Optimizations

```
for (int i=0; i<n; i++)
{
    c[i] = exp(i/n) ;
    for (int j=0; j<m; j++)
        a[j][i] = b[j][i] + d[j] * e[i];
}
```

```
for (int i=0; i<n; i++)
    c[i] = exp(i/n) ;

for (int j=0; j<m; j++)
    for (int i=0; i<n; i++)
        a[j][i] = b[j][i] + d[j] * e[i];
```

Loop Optimizations

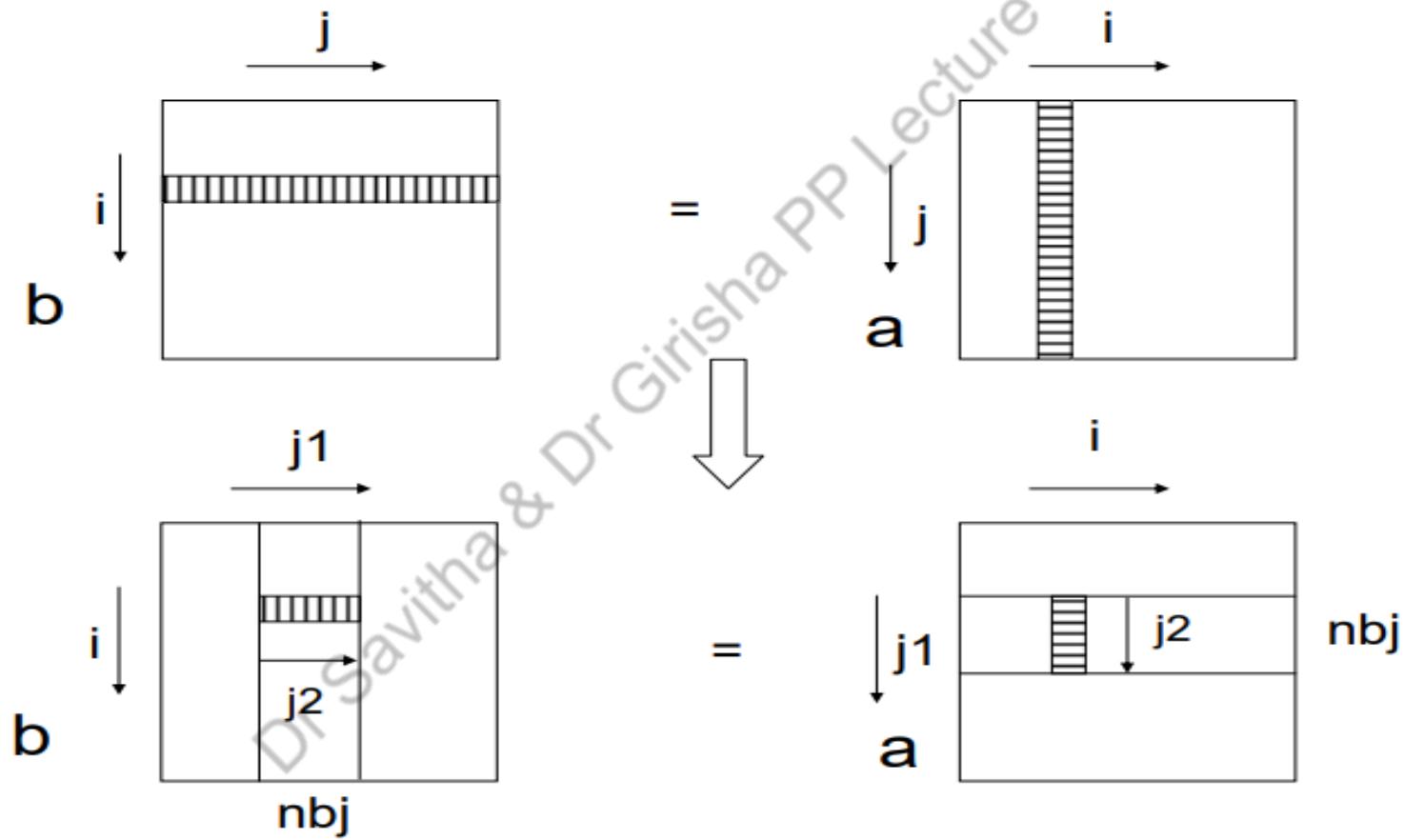
- **Loop tiling or blocking**
 - It is a powerful transformation designed to tailor the number of memory references inside a loop iteration so that they fit into cache
 - If data sizes are large and memory access is bad
 - If there is data reuse in the loop
- Loop tiling replaces the original loop by a pair of loops

```
for (int i=0; i<n; i++)
  for (int j=0; j<m; j++)
    b[i][j] = a[j][i];
```



```
for (int j1=0; j1<n; j1+=nbj)
  for (int i=0; i<n; i++)
    for (int j2=0; j2 < MIN(n-j1,nbj); j2++)
      b[i][j1+j2] = a[j1+j2][i];
```

Loop Optimizations



Use of Pointers and Contiguous Memory in C

- The memory model in C is such that, without additional information, one must assume that all pointers may reference any memory address
 - This is generally referred to as the **pointer aliasing problem**
 - It prevents a compiler from performing many program optimizations
- If pointers are guaranteed to point to portions of nonoverlapping memory, optimizations can be applied

Using Compilers

- Modern compilers implement most, if not all, of the loop optimizations
- They perform a variety of analyses to determine whether they may be applied
 - The main one is known as data dependence analysis
- They also apply a variety of techniques to reduce the number of operations performed and reorder code to better exploit the hardware
- It is worthwhile to experiment with compiler options to squeeze the maximum performance out of the application

Amdahl's law

- Amdahl's Law is a principle used in parallel computing to predict the maximum potential speedup when using multiple processors. It's named after Gene Amdahl, a computer architect, who formulated it in 1967.
- If we denote by T_1 the execution time of an application on 1 processor, then in an ideal situation, the execution time on P processors should be T_1/P
- If T_P denotes the execution time on P processors, then the ratio
$$S = T_1/T_P$$
- Parallel speedup is a measure of the success of the parallelization

Amdahl's law

- Virtually all programs contain some regions that are suitable for parallelization and other regions that are not
- By using an increasing number of processors, the time spent in the parallelized parts of the program is reduced, but the sequential section remains the same
- Eventually the execution time is completely dominated by the time taken to compute the sequential portion, which puts an upper limit on the expected speedup

$$S = 1 / (f_{\text{par}}/P + (1 - f_{\text{par}}))$$

- f_{par} is the parallel fraction of the code and P is the number of processors

Amdahl's law

- Suppose that 70% of a program execution can be speeded up if the program is parallelized and run on 16 processing units instead of one. What is the **maximum speedup** that can be achieved by the whole program?
- What is the **maximum speedup** if we increase the number of processing units to 32, then to 64, and then to 128

Amdahl's law

- Obstacles along the way to perfect linear speedup are the overheads introduced by **forking and joining threads, thread synchronization, and memory accesses**
- A measure of a program's ability to decrease the execution time of the code with an increasing number of processors is referred to as **parallel scalability**

Measuring OpenMP Performance

- How to measure and identify what factors determine overall program performance
- On Unix systems if we use **:/bin/time ./a.out**

```
$ /bin/time ./program.exe  
real 5.4  
user 3.2  
sys 1.0
```

Measuring OpenMP Performance

- **Real:** Program took 5.4 seconds from beginning to end
- **User:** The time the program spent executing outside any operating system services
- **Sys:** The time spent on operating system services, such as input/output routines

```
$ /bin/time ./program.exe
```

real	5.4
user	3.2
sys	1.0

- **CPU time:** The sum of user and system time
- The real time is also referred to as **wall-clock time** or **elapsed time**

Measuring OpenMP Performance

- There is a difference between real time and CPU time
 - The application did not get a full processor to itself, because of a high load on the system
- OpenMP program has additional overheads
 - These overheads are collectively called the ***parallel overhead***
 - It includes the time to
 - Create, start, and stop threads
 - The extra work needed to figure out what each task is to perform
 - The time spent waiting in barriers and at critical sections and locks
 - The time spent computing some operations redundantly

```
$ /bin/time ./program.exe
real      5.4
user      3.2
sys       1.0
```

Measuring OpenMP Performance

$$T_{CPU}(P) = (1 + O_P \cdot P)T_{Serial}$$

$$T_{Elapsed}(P) = ((\frac{f}{p}) + 1 - f + O_P \cdot P)T_{Serial}$$

- T_{Serial} is the CPU time of the original serial version of the application
- P is the number of processors
- O_P is the parallel overhead
- P with O_P assumed to be a constant percentage
- $f \in [0, 1]$ is the fraction of execution time that has been parallelized

Measuring OpenMP Performance

- If the original program takes $T_{Serial} = 10.20$ seconds to run and code corresponding to 95% of the execution time has been parallelized. Assume that each additional processor adds a 2% overhead to the total CPU time. Compute Speedup and Efficiency of the parallel program with 4 processors. Also estimate the T_{CPU} and $T_{Elapsed}$ of the given program.

Solution

- To compute the speedup and efficiency of the parallel program with 4 processors, need to calculate the total execution time with and without parallelization, and then use these values to find the speedup and efficiency.
 - T_{Serial} = Original serial execution time = 10.20 seconds
 - P = Number of processors = 4
 - S_P = Speedup achieved by parallelization
 - E_P = Efficiency achieved by parallelization
 - $T_{Parallel}$ = Total execution time with parallelization
 - T_{CPU} = Total CPU time
 - $T_{Elapsed}$ = Total elapsed time

Given that 95% of the execution time is parallelized, calculate the total execution time with parallelization:

$$T_{\text{Parallel}} = (1 - 0.95) \times T_{\text{Serial}} + \frac{0.95 \times T_{\text{Serial}}}{P} + (P - 1) \times 0.02 \times T_{\text{Serial}}$$

This equation represents the total execution time of the program when parallelized. The first term represents the sequential portion of the program that cannot be parallelized. The second term represents the parallel portion of the program that can be divided among the processors, with $\frac{0.95 \times T_{\text{Serial}}}{P}$ being the time taken when 95% of the program is parallelized and executed across P processors. The third term accounts for the overhead incurred due to additional processors, where $(P - 1) \times 0.02 \times T_{\text{Serial}}$ is the additional time taken with P processors, each incurring a 2% overhead.

$$T_{Parallel} = (1 - 0.95) \times T_{Serial} + \frac{0.95 \times T_{Serial}}{P} + (P - 1) \times 0.02 \times T_{Serial}$$

$$T_{Parallel} = 0.05 \times 10.20 + \frac{0.95 \times 10.20}{4} + 3 \times 0.02 \times 10.20$$

$$T_{Parallel} \approx 0.51 + \frac{9.69}{4} + 3 \times 0.204$$

$$T_{Parallel} \approx 0.51 + 2.4225 + 0.612 \approx 3.5445$$

Now, let's calculate the speedup:

$$S_P = \frac{T_{Serial}}{T_{Parallel}}$$

$$S_P = \frac{10.20}{3.5445} \approx 2.878$$

Next, let's calculate the efficiency:

$$E_P = \frac{S_P}{P}$$

$$E_P = \frac{2.878}{4} \approx 0.7195$$

Now, let's estimate the total CPU time (T_{CPU}) and total elapsed time ($T_{Elapsed}$):

- Total CPU time (T_{CPU}) can be approximated as the total execution time with parallelization:

$$T_{CPU} = T_{Parallel}$$

- Total elapsed time ($T_{Elapsed}$) can be taken as the same as the total execution time since we assume no other significant overheads: $T_{Elapsed} = T_{Parallel}$

So, $T_{CPU} \approx 3.5445$ seconds and $T_{Elapsed} \approx 3.5445$ seconds.

To summarize:

- Speedup (S_P) ≈ 2.878
- Efficiency (E_P) ≈ 0.7195
- Total CPU time (T_{CPU}) ≈ 3.5445 seconds
- Total elapsed time ($T_{Elapsed}$) ≈ 3.5445 seconds

Measuring OpenMP Performance

- If the original program takes $T_{Serial} = 18.35$ seconds to run and code corresponding to 72% of the execution time has been parallelized. Assume that each additional processor adds a 6% overhead to the total CPU time. Compute Speedup and Efficiency of the parallel program with 8 processors. Also estimate estimate the T_{CPU} and $T_{Elapsed}$ of the given program.

Measuring OpenMP Performance

- The observable performance of OpenMP programs is influenced by at least the following factors
 - The manner in which memory is accessed by the individual threads
 - The fraction of the work that is sequential, or replicated (**Sequential overheads**)
 - The amount of time spent handling OpenMP constructs (**Parallelization overheads**)
 - When a work-sharing directive is implemented, the work to be performed by each thread is usually determined at run time
 - The load imbalance between synchronization points (**Load imbalance overheads**)
 - Threads perform different amounts of work in a work-shared region
 - Threads might have to wait for a member of their team to carry out the work of a single construct
 - Other synchronization costs (**Synchronization overheads**)

Measuring OpenMP Performance

- Suppose that 65% of program execution can be sped up if the program is parallelized and run on 8 processing units instead of one. What is the maximum speedup that can be achieved by the whole program? What is the maximum speedup if we increase the number of processing units to 16.
- If the original program takes $T_{Serial} = 9.4$ seconds to run and code corresponding to 68% of the execution time has been parallelized. Assume that 6 processors incur a total overhead of 0.24 units to the total CPU time. Compute Speedup and Efficiency of the parallel program with 6 processors. Also estimate estimate the T_{CPU} and $T_{Elapsed}$ of the given program.

Best Practices

- Optimize Barrier Use
 - Barriers are expensive operations
 - Can use Nowait clause to ignore the barriers

```
#pragma omp parallel
{
    .....
#pragma omp for
    for (i=0; i<n; i++)
    .....
#pragma omp for nowait
    for (i=0; i<n; i++)

} /*-- End of parallel region - barrier is implied --*/
```

Best Practices

- Optimize Barrier Use

```
#pragma omp parallel default(none) \
    shared(n,a,b,c,d,sum) private(i)
{
    #pragma omp for nowait
    for (i=0; i<n; i++)
        a[i] += b[i];

    #pragma omp for nowait
    for (i=0; i<n; i++)
        c[i] += d[i];

    #pragma omp barrier

    #pragma omp for nowait reduction(+:sum)
    for (i=0; i<n; i++)
        sum += a[i] + c[i];
} /*-- End of parallel region --*/
```

Best Practices

- Avoid the Ordered Construct
 - The ordered construct ensures that the corresponding block of code within a parallel loop is executed in the order of the loop iterations
 - The runtime system has to keep track which iterations have finished and possibly keep threads in a wait state until their results are needed
- Avoid Large Critical Regions
 - The more code contained in the critical region, however, the greater the likelihood that threads have to wait to enter it, and the longer the potential wait times

```
#pragma omp parallel shared(a,b) private(c,d)
{
    .....
#pragma omp critical
{
    a += 2 * c;
    c = d * d;
}
} /*-- End of parallel region --*/
```

Best Practices

- Maximize Parallel Regions
 - Overheads are associated with starting and terminating a parallel region
 - Large parallel regions offer more opportunities for using data in cache and provide a bigger context for other compiler optimizations
 - Minimize the number of parallel regions

```
#pragma omp parallel for
for (.....)
{
    /**- Work-sharing loop 1 --*/
}

#pragma omp parallel for
for (.....)
{
    /**- Work-sharing loop 2 --*/
}
.....
#pragma omp parallel for
for (.....)
{
    /**- Work-sharing loop N --*/
}
```

Best Practices

```
#pragma omp parallel
{
    #pragma omp for /*-- Work-sharing loop 1 --*/
    { ..... }

    #pragma omp for /*-- Work-sharing loop 2 --*/
    { ..... }

    .....

    #pragma omp for /*-- Work-sharing loop N --*/
    { ..... }
}
```

Best Practices

- Avoid Parallel Regions in Inner Loops
 - We repeatedly experience the overheads of the parallel construct
 - the overheads of the #pragma omp parallel for construct are incurred n² times

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        #pragma omp parallel for
        for (k=0; k<n; k++)
            { .....
```

```
#pragma omp parallel
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            #pragma omp for
            for (k=0; k<n; k++)
                { .....
```

Best Practices

- Address Poor Load Balance
 - In some parallel algorithms, threads have different amounts of work to do
 - The threads wait at the next synchronization point until the slowest one completes
 - Solution is to use schedule clause
 - The dynamic and guided workload distribution schedules have higher overheads than does the static scheme

```
for (i=0; i<N; i++) {  
    ReadFromFile(i,...);  
    for (j=0; j<ProcessingNum; j++ )  
        ProcessData(); /* lots of work here */  
    WriteResultsToFile(i);  
}
```

Best Practices

```
#pragma omp parallel
{
    /* preload data to be used in first iteration of the i-loop */
    #pragma omp single
        {ReadFromFile(0,...);}

    for (i=0; i<N; i++) {

        /* preload data for next iteration of the i-loop */
        #pragma omp single nowait
            {ReadFromFile(i+1...);}

        #pragma omp for schedule(dynamic)
            for (j=0; j<ProcessingNum; j++)
                ProcessChunkOfData(); /* here is the work */
        /* there is a barrier at the end of this loop */

        #pragma omp single nowait
            {WriteResultsToFile(i);}

    } /* threads immediately move on to next iteration of i-loop */
} /* one parallel region encloses all the work */
```

Additional Performance Considerations

- The Single Construct Versus the Master Construct
 - A **single region** can be executed by any thread, typically the first to encounter it
 - A single construct has an implicit barrier
 - Whereas this is not the case for the master region
 - A master construct can be more efficient: **Single construct requires more work in the OpenMP library**
 - The single construct might be more efficient if the master thread is not likely to be the first one to reach it and the threads need to synchronize at the end of the block

Additional Performance Considerations

- Avoid False Sharing
 - One of the factors limiting scalable performance is false sharing
 - State bits are used by cache coherence protocols to track the state of the cache line
 - If a single byte is updated in the cache, the entire cache line needs to be fetched from the memory
 - Threads update different data elements in the same cache line, they interfere with each other
 - This effect is known as false sharing

```
#pragma omp parallel for shared(Nthreads,a) schedule(static,1)
    for (int i=0; i<Nthreads; i++)
        a[i] += i;
```

Additional Performance Considerations

- Avoid False Sharing
 - Array padding can be used to eliminate the problem
 - Changing the indexing from $a[i]$ to $a[i][0]$ eliminates the false sharing
- False sharing is likely to significantly impact performance under the following conditions:
 - Shared data is modified by multiple threads
 - The access pattern is such that multiple threads modify the same cache line(s)
 - These modifications occur in rapid succession

Additional Performance Considerations

- Private Versus Shared Data
 - The programmer may often choose whether data should be shared or private
 - For example, if threads need unique read/write access to a one dimensional array, one could declare a two-dimensional shared array with one row
 - Alternatively, each thread might allocate a one dimensional private array within the parallel region
 - In general, the latter approach is to be preferred over the former
 - If there are frequent modification to the data, it may result in false sharing
 - Degrades performance
 - If data is read but not written in a parallel region, it could be shared, ensuring that each thread has (read) access to it.

END

Dr Savitha & Dr Girisha PP Lecture Slides