



Parallel Programming Principle and Practice

Lecture 5 —logical parallel programming models



Outline

- ❑ MIMD programming models
 - shared-memory model
 - distributed-memory model
- ❑ SIMD programming models
 - vector computing model
 - CUDA model
 - *OpenCL model
- ❑ * Mapreduce
- ❑ * Tensorflow

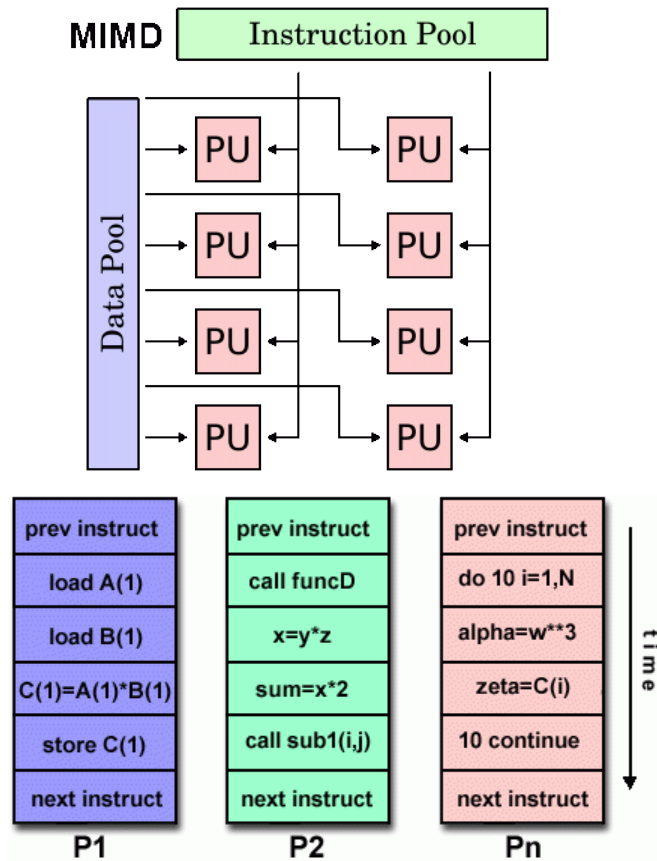
Logical parallel programming models

MIMD programming models

What is MIMD?

Multiple Instruction, Multiple Data (MIMD)

- ❑ **Multiple Instruction:** Every processor may be executing a different instruction stream
- ❑ **Multiple Data:** Every processor may be working with a different data stream
- ❑ Execution can be synchronous or asynchronous, deterministic or non-deterministic 执行可以是同步的或异步的、确定性的或非确定性的



What is MIMD?

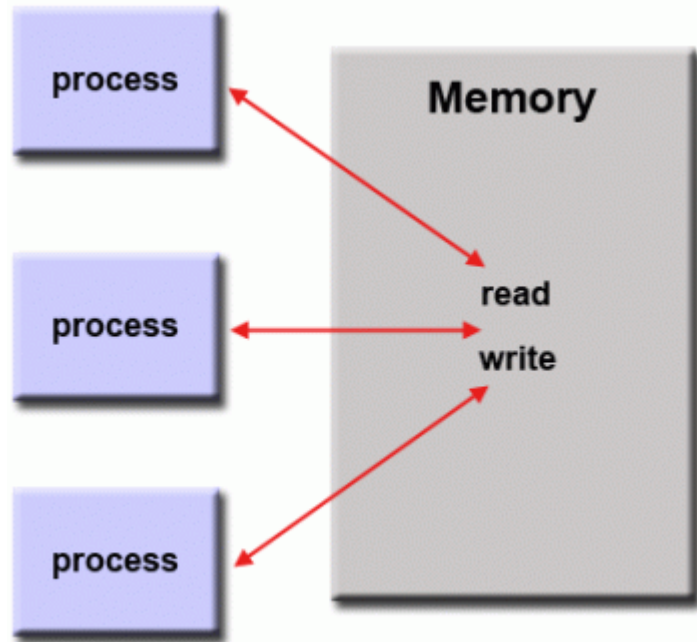
- ❑ Multiple Instruction, Multiple Data (MIMD)
- ❑ Currently, the most common type of parallel computer - most modern supercomputers fall into this category
- ❑ Examples
 - most current supercomputers
 - networked parallel computer clusters and "grids"
 - multi-processor SMP computers
 - multi-core PCs

Classifications of MIMD

- MIMD machines can be of either shared memory or distributed memory categories
 - These classifications are based on how MIMD processors access memory
 - **Shared memory machines** may be of the bus-based, extended, or hierarchical type
共享内存机器可以是基于总线的、扩展的或分层的类型
 - **Distributed memory machines** may have hypercube or mesh interconnection schemes
分布式内存机器可能具有超立方体或网状互连方案

Shared memory programming model

- In this programming model, **processes/tasks share a common address space**, which they read and write to asynchronously
- Various mechanisms such as **locks** / **semaphores** are used to control access to the shared memory, resolve contentions and to prevent race conditions and deadlocks



Examples

- ❑ On **stand-alone shared memory machines**, native operating systems, compilers and/or hardware provide support for shared memory programming.
- ❑ For example,
 - ❑ the POSIX (可移植操作系统接口, Portable Operating System Interface of UNIX) standard provides an API for using shared memory
 - ❑ UNIX provides **shared memory segments** (shmget, shmat, shmctl, etc.)

Examples

- shmget() 得到一个共享内存标识符或创建一个共享内存对象

int shmget(key_t key, int size, int flag);

- **shmget()** returns a shared memory ID if OK, -1 on error
 - **Key** is typically the constant “IPC_PRIVATE”, which lets the kernel choose a new key – keys are non-negative integer identifier
 - **Size** is the size of shared memory segment in bytes
 - **Flag**: IPC_CREAT|IPC_EXCL (If there is no shared memory whose key value is equal to the key in the kernel, a new shared memory will be created; if such a shared memory exists, an error will be reported)

Examples

- ❑ `shmat()`把共享内存区对象映射到调用进程的地址空间

Once a shared memory segment has been created, a process attaches it to its address space by calling `shmat()`:

```
void *shmat(int shmid, void* addr, int flag);
```

- `shmat()` returns a **pointer** to shared memory segment if OK, -1 on error
- The recommended technique is to set `addr` and `flag` to zero,

```
i.e.: char* buf=(char*)shmat(shmid,0,0);
```

- The UNIX commands “`ipcs`” and “`ipcrm`” are used to list and remove
- shared memory segments on the current machine

Examples

□ shmctl() 共享内存管理

shmctl() performs various shared memory operations:

*int shmctl (int shmid, int cmd, struct shmid_ds *buf);*

- **cmd** can be one of IPC_STAT, IPC_SET, or IPC_RMID:
 - **IPC_STAT** fills the buf data structure
 - **IPC_SET** can change the uid, gid, and mode of the shmid
 - **IPC_RMID** sets up the shared memory segment to be removed from the system once the last process using the segment terminates or detaches from it – a process detaches from a shared memory segment using shmdt(void* addr), which is similar to free()
- shmctl() returns 0 if OK, -1 on error

Shared Memory Example

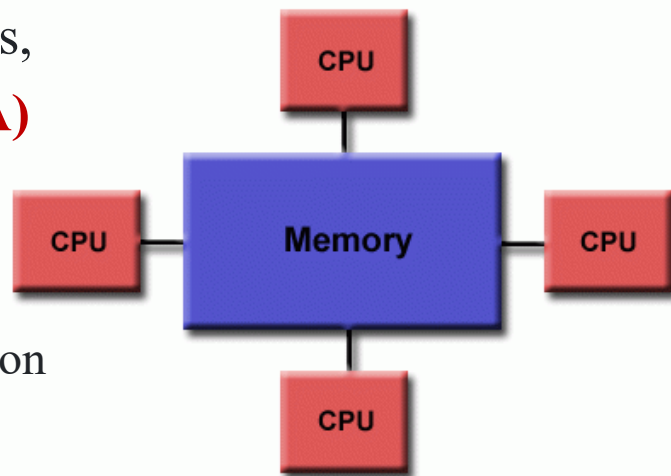
```
char* ShareMalloc(int size)
{
    int shmId;
    char* returnPtr;
    //得到一个共享内存标识符或创建一个共享内存对象
    if ((shmId=shmget(IPC_PRIVATE, size, (SHM_R | SHM_W)) < 0)
        Abort("Failure on shmget\n");
    //把共享内存区对象映射到调用进程的地址空间,这里0让内核自己决定一个合适的地址位置
    if (returnPtr=(char*)shmat(shmId,0,0)) == (void*) -1)
        Abort("Failure on shmat\n");
    //进程结束后删除这片共享内存
    shmctl(shmId, IPC_RMID, (struct shmid_ds *) NULL);
    return (returnPtr);
}
```

Shared memory

- ❑ Shared memory parallel computers vary widely
 - generally have in common the ability for all processors to access all memory as global address space
 - Multiple processors can operate independently but share the same memory resources
 - Changes in a memory location effected by one processor are visible to all other processors
- ❑ Historically, shared memory machines have been classified as **UMA** and **NUMA**, based upon memory access times

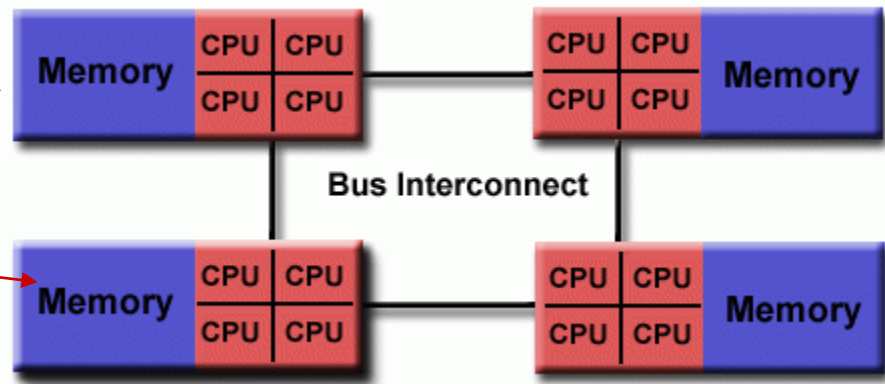
Uniform Memory Access (UMA) 统一内存访问

- Most commonly represented today by **Symmetric Multiprocessor (SMP)** (对称多处理器) machines, sometimes called **CC-UMA (Cache Coherent UMA)**
 - Identical processors
 - Equal access and access times to memory
 - Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update
 - Cache coherency is accomplished at the hardware level



Non-Uniform Memory Access (NUMA)非均匀内存访问

- ❑ Often made by physically linking two or more SMPs
- ❑ One SMP ~~can directly~~ access memory of another SMP
- ❑ Not all processors have equal access time to all memories
 - Memory access across link is slower



Implementations

- ❑ **Native compilers and/or hardware** translate user program variables into actual memory addresses, which are global
 - On **stand-alone SMP machines**, this is straightforward
 - On **distributed shared memory machines**, memory is physically distributed across a network of machines, but made global through specialized hardware and software

Evaluation: shared memory

Advantages

- ❑ Global address space provides a user-friendly programming perspective to memory
- ❑ Data sharing between tasks is **both fast and uniform** due to the proximity of memory to CPUs

Disadvantages

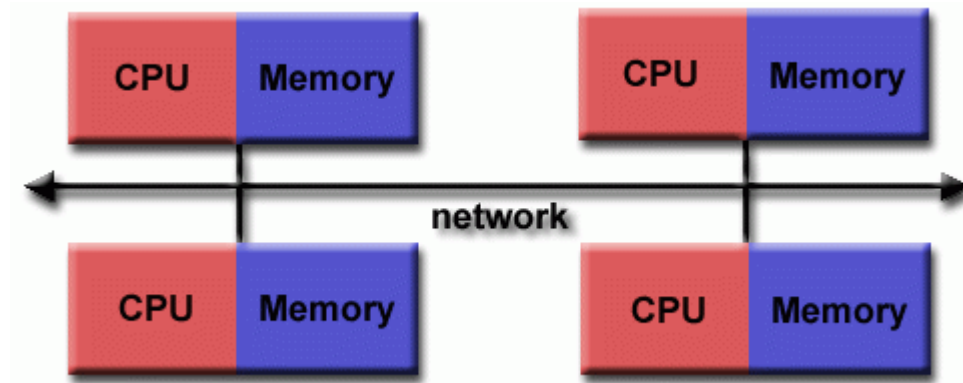
- ❑ Primary disadvantage is the **lack of scalability between memory and CPUs**
 - ❑ **Adding more CPUs** can geometrically increase traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management
- ❑ Programmer responsibility for synchronization constructs that ensure "correct" access of global memory

Shared memory programming model

- An **advantage** of this model from the programmer's point of view is that the notion of data "ownership" is lacking
 - there is no need to specify explicitly the communication of data between tasks
 - all processes see and have equal access to shared memory
 - program development can often be simplified
- An important **disadvantage** in terms of performance is that it becomes more difficult to understand and manage *data locality*
 - **Keeping data local to the process** that works on it conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processes use the same data
 - Unfortunately, controlling data locality is hard to understand and may be beyond the control of the average user

Distributed-memory

- ❑ In distributed memory MIMD(multiple instruction, multiple data) machines, each processor has its own individual memory location
- ❑ Each processor has no direct knowledge about other processor's memory
 - For data to be shared, it must be passed from one processor to another as a message



Distributed-memory

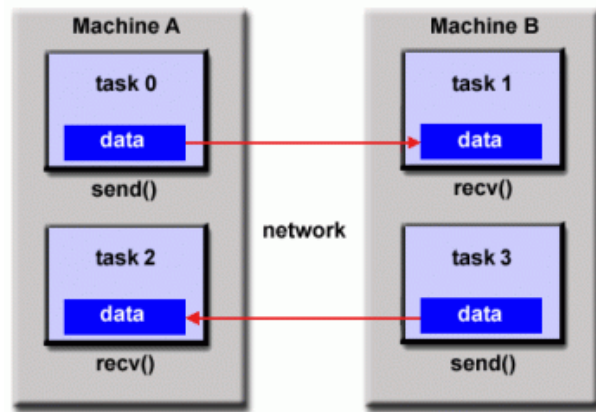
- Like shared memory systems, distributed memory systems vary widely but share a common characteristic
 - Distributed memory systems require a communication network to **connect inter-processor memory**
 - Processors have their own local memory
 - Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors

Distributed-memory

- Because each processor has its own local memory, it operates independently
 - Changes it makes to its local memory have no effect on the memory of other processors
 - the concept of **cache coherency** does not apply
- When a processor needs access to data in another processor, it is usually the task of the programmer to **explicitly define how and when data is communicated**
 - Synchronization between tasks is likewise the programmer's responsibility
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet

Distributed Memory / Message Passing programming model

- This model demonstrates the following characteristics:
 - A set of tasks that use their own local memory during computation
 - Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines
 - Tasks **exchange data** through communications by sending and receiving messages
 - Data transfer usually requires **cooperative operations** to be performed by each process
 - For example, a **send** operation must have a matching **receive** operation



Implementations

- ❑ From a programming perspective
 - **Message passing implementations** usually comprise a library of subroutines
 - Calls to these subroutines are imbedded in source code
 - The programmer is responsible for determining all parallelism
- ❑ Historically, **a variety of message passing libraries** have been available since the 1980s
 - These implementations differed substantially from each other making it difficult for programmers to develop portable applications
- ❑ In 1992, the **MPI** Forum was formed with the primary goal of establishing a standard interface for message passing implementations

Implementations

- ❑ Part 1 of the **Message Passing Interface (MPI)** was released in 1994. Part 2 (MPI-2) was released in 1996. Both MPI specifications are available on the web at <http://www-unix.mcs.anl.gov/mpi/>
- ❑ MPI is now the *de facto industry standard* for message passing, replacing virtually all other message passing implementations used for production work
- ❑ MPI implementations exist virtually for all popular parallel computing platforms.
Not all implementations include everything in both MPI-1 and MPI-2

Example: MPI (Message Passing Interface)

- ❑ MPI defines a standard library for message-passing that can be used to develop portable message-passing programs using either C or Fortran
- ❑ The MPI standard defines both the syntax as well as the semantics of a core set of library routines
- ❑ Vendor implementations of MPI are available on almost all commercial parallel computers
- ❑ It is possible to write fully-functional message-passing programs by using only the six routines

Example: MPI (Message Passing Interface)

- ❑ `MPI_Init` is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment
- ❑ `MPI_Finalize` is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment
- ❑ The prototypes of these two functions are:

```
int MPI_Init(int *argc, char ***argv)
```

```
int MPI_Finalize()
```

- ❑ `MPI_Init` also strips off any MPI related command-line arguments
- ❑ All MPI routines, data-types, and constants are prefixed by “MPI_”
 - The return code for successful completion is `MPI_SUCCESS`

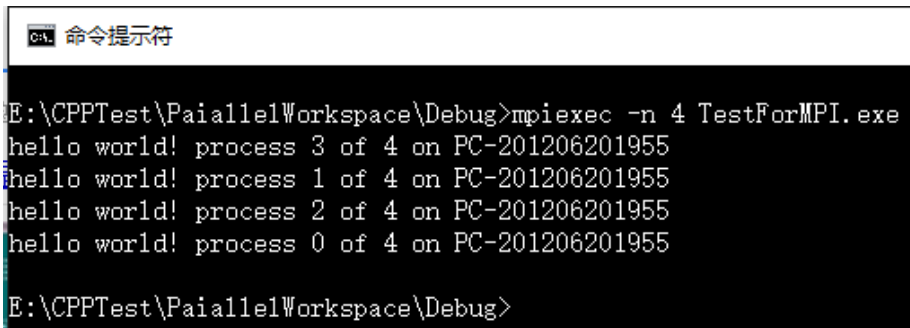
Example: MPI (Message Passing Interface)

- ❑ The `MPI_Comm_size` and `MPI_Comm_rank` functions are used to determine the number of processes(进程) and the label of the calling process, respectively
- ❑ The calling sequences of these routines are as follows:

```
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```
- ❑ The rank of a process is **an integer that ranges from zero up to the size of the communicator minus one**

Example: MPI (Message Passing Interface)

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char* argv[])
{
    int rank, numprocs;
    int namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // 获得进程号
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs); // 返回通信子进程的数量
    MPI_Get_processor_name(processor_name, &namelen);
    fprintf(stderr, "hello world! process %d of %d on %s\n", rank, numprocs,
processor_name);
    MPI_Finalize();
    return 0;
}
```



```
命令提示符
E:\CPPTest\PaiallelWorkspace\Debug>mpiexec -n 4 TestForMPI.exe
hello world! process 3 of 4 on PC-201206201955
hello world! process 1 of 4 on PC-201206201955
hello world! process 2 of 4 on PC-201206201955
hello world! process 0 of 4 on PC-201206201955
E:\CPPTest\PaiallelWorkspace\Debug>
```

Sending and Receiving Messages

- The basic functions for sending and receiving messages in MPI are the `MPI_Send` and `MPI_Recv`, respectively

- The calling sequences of these routines are as follows:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm)
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int
tag, MPI_Comm comm, MPI_Status *status)
```

- MPI provides equivalent datatypes for all C datatypes. This is done for portability reasons
- The datatype `MPI_BYTE` corresponds to a byte (8 bits) and `MPI_PACKED` corresponds to a collection of data items that has been created by packing non-contiguous data
- The **message-tag** can take values ranging from zero up to the MPI defined constant `MPI_TAG_UB`

Evaluation: distributed-memory

Advantages

- **Memory is scalable** with the number of processors
 - Increase the number of processors and the size of memory increases proportionately (增加处理器，相应的也增加内存。能够无可限扩展么？)
 - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain global cache coherency
- **Cost effectiveness**: can use commodity, off-the-shelf processors and networking (成熟、商用的处理器和网络)

Evaluation: distributed-memory

Disadvantages

- The **programmer** is responsible for many of the details associated with data communication between processors (能够无限扩展么? 怎么办?)
 - It may be difficult to map existing data structures, based on global memory, to this memory organization
- Non-uniform memory access times - **data residing (驻留) on a remote node takes longer to access than node local data**

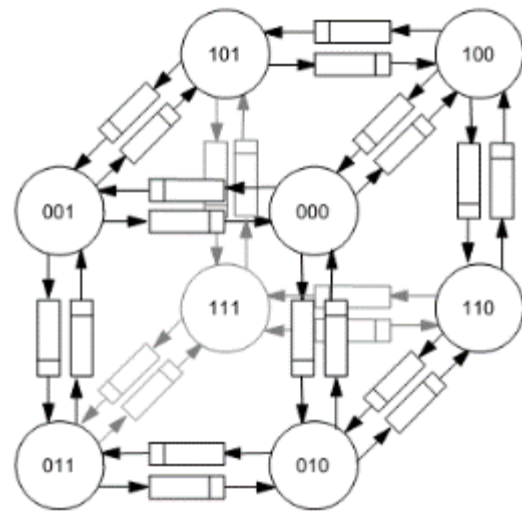
Evolution of Message Passing

❑ Early machines: FIFO on each link

- Hardware **close** to programming model
 - synchronous ops
- Replaced by DMA, enabling non-blocking ops
 - **Buffered by system** at destination until recv

❑ Diminishing role of topology

- Store & forward routing: topology important
- Introduction of pipelined routing made it less so important
- **Cost is in node network interface**
- Simplifies programming



Toward Architectural Convergence

- ❑ Evolution and role of software have blurred boundary
 - **Send/recv** supported on SAS machines via buffers
 - Can **construct global address space** on MP using hashing
 - **Page-based (or fine-grained) shared virtual memory**

- ❑ Programming models distinct, but organizations converging (会聚)
 - **Nodes connected by general network and communication assists**
 - Implementations also converging, at least in high-end machines



SIMD programming models

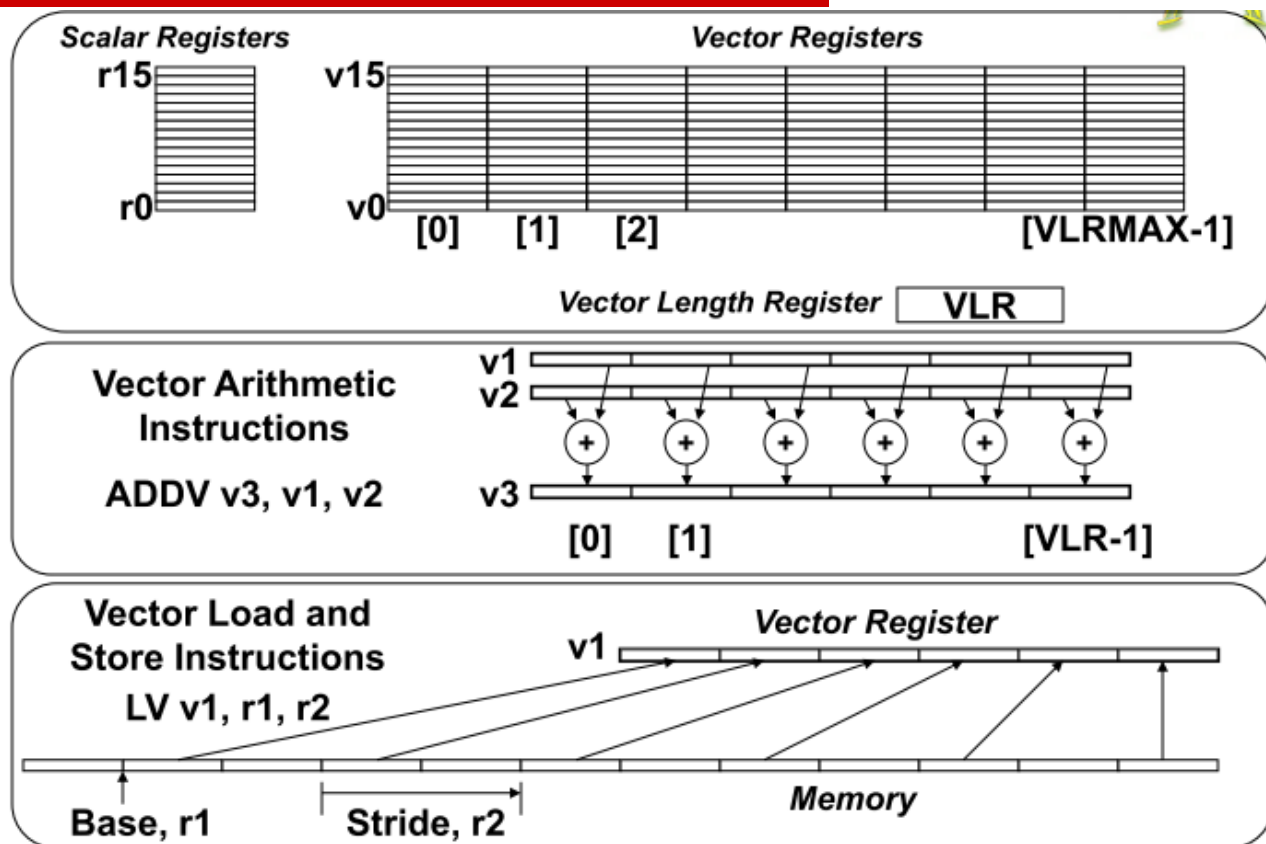
Vector computing model(矢量计算模型)

Vector Code Example

# C code	# Scalar Code	# Vector Code
<pre>for (i=0; i<64; i++) C[i] = A[i] + B[i];</pre>	<pre>LI R4, 64 loop: L.D F0, 0(R1) L.D F2, 0(R2) ADD.D F4, F2, F0 S.D F4, 0(R3) DADDIU R1, 8 DADDIU R2, 8 DADDIU R3, 8 DSUBIU R4, 1 BNEZ R4, loop</pre>	<pre>LI VLR, 64 LV V1, R1 LV V2, R2 ADDV.D V3, V1, V2 SV V3, R3</pre>

- ❑ Require programmer (or compiler) to identify parallelism
 - Hardware does not need to re-extract parallelism
- ❑ Many multimedia/HPC applications are natural consumers of vector processing

Vector Programming Model





SIMD programming models

CUDA model

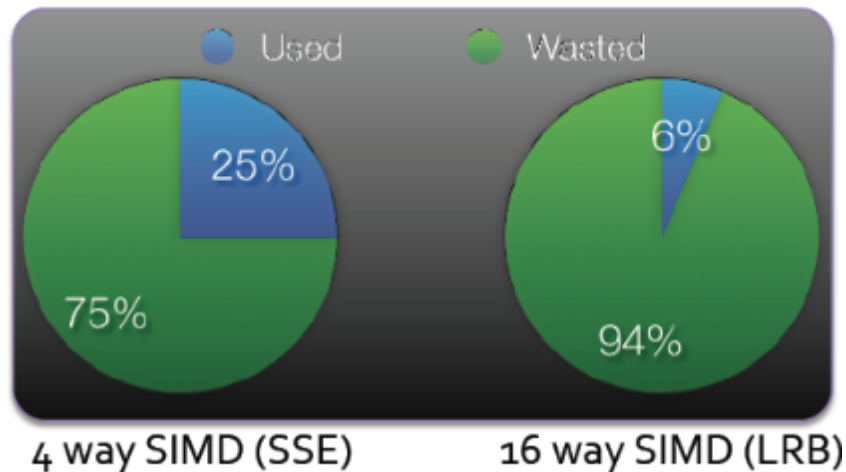
CUDA(Compute Unified Device Architecture:统一计算设备架构)

- ❑ Software layer
- ❑ A parallel computing platform and application programming interface
- ❑ Allows software to use certain types of GPU for general purpose processing (GPGPU)
- ❑ Created by Nvidia

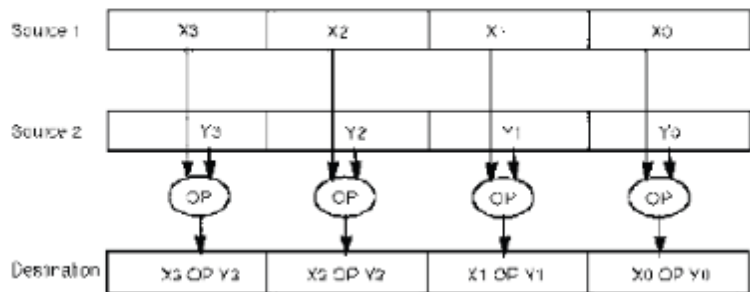


Developer(s)	Nvidia
Initial release	June 23, 2007; 14 years ago
Stable release	11.6.1 / February 22, 2022; 22 days ago
Operating system	Windows, Linux
Platform	Supported GPUs
Type	GPGPU
License	Proprietary
Website	developer.nvidia.com/cuda-zone

CUDA Goals: SIMD Programming



- ❑ Hardware architects love SIMD, since it permits a very space and energy-efficient implementation
- ❑ However, standard SIMD instructions on CPUs are inflexible, and difficult to use, difficult for a compiler to target
- ❑ CUDA thread abstraction will **provide programmability at the cost of additional hardware**



Example : Pytorch 调用 cuda

#定义device为cuda

device='cuda'

#模型加载到cuda

model = model.to(device)

#调用交叉熵损失函数

criterion = torch.nn.CrossEntropyLoss()

#损失函数加载到cuda

criterion = criterion.to(device)

#图片数据加载到cuda

image = image.float().to(device)

#标签数据加载到cuda

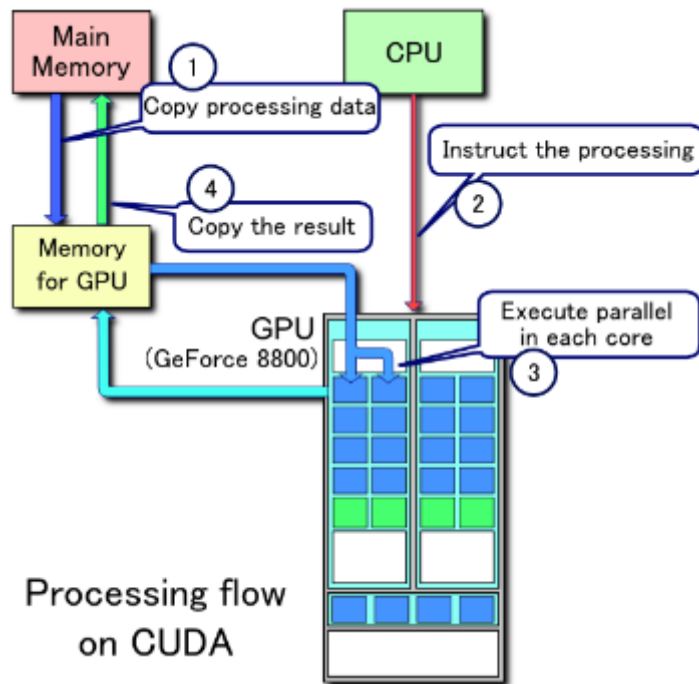
label = label.long().to(device)

Code semantics:

- ❑ pytorch调用cuda api, 将神经网络模型, 损失函数, 数据加载到GPU上, 届时相关数据会在GPU上计算
- ❑ 若不调用相关cuda api, 则相应数据只会在CPU上运算
- ❑ 调用cuda api以便使用GPU会极大节省计算时间
 - 如一个5层的神经网络, 跑完20万次迭代
 - 在Core i5上大概需要24天
 - 在RTX980上大概需要24小时

Processing Flow

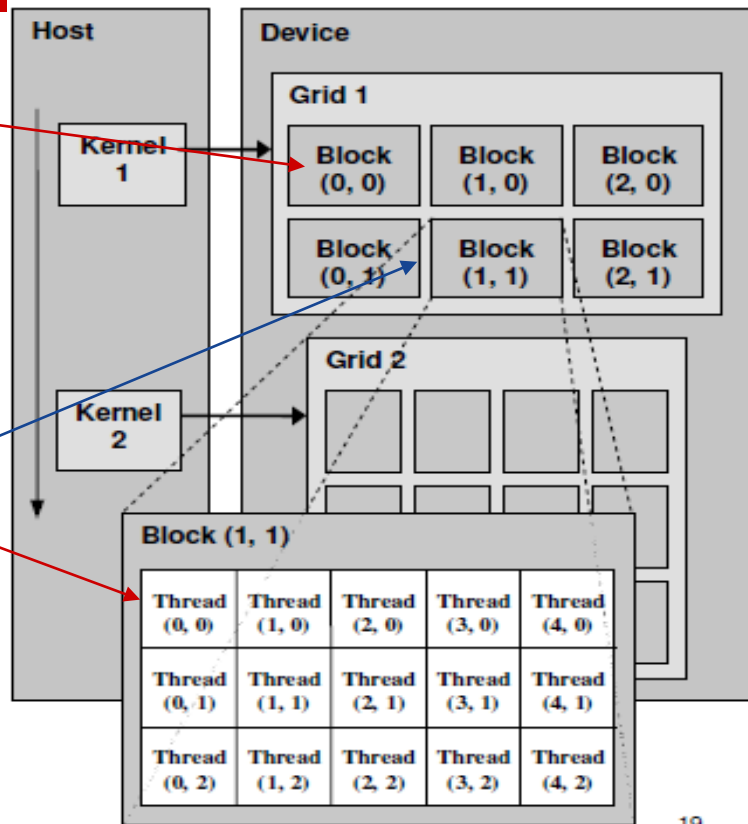
- ① Copy data from main memory to GPU memory
- ② CPU initiates the GPU compute kernel
- ③ GPU's CUDA cores execute the kernel in parallel
- ④ Copy the resulting data from GPU memory to main memory



CUDA Programming Model

A **kernel** is executed by a **grid** of **thread blocks**

- A thread block is a batch of threads that can cooperate with each other by:
 - Sharing data through shared memory
 - Synchronizing their execution
- Threads from different blocks cannot cooperate



Advantages of CUDA

- ❑ CUDA has several **advantages** over traditional general purpose computation on GPUs
 - Scattered reads (分散读取)— code can read from **arbitrary addresses in memory**
 - Shared memory (共享内存) - CUDA exposes **a fast shared memory region** (16KB in size) that can be shared amongst threads

Limitations of CUDA

- ❑ CUDA has several **limitations** over traditional general purpose computation on GPUs
 - A single process must run spread across **multiple disjoint memory spaces**, unlike other C language runtime environments
 - The bus bandwidth and latency between the CPU and the GPU may be a bottleneck
 - CUDA-enabled GPUs are only available from NVIDIA

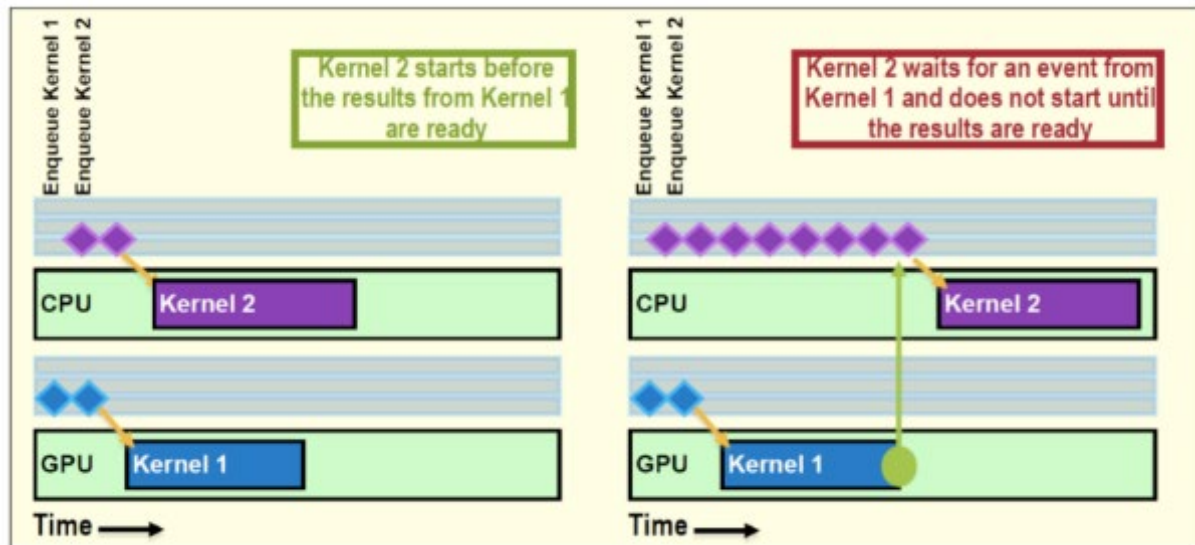


SIMD programming models

***OpenCL model**

OpenCL Programming Model

- ❑ OpenCL is **a framework for writing programs**
- ❑ Execute across heterogeneous platforms consisting of CPUs, GPUs, DSPs, FPGAs and other processors or hardware accelerators
- ❑ **Concurrency Control with OpenCL Event-Queueing**



*Functions executed on an OpenCL device are called kernels

Example

❑ OpenCL kernel language

- The programming language that is used to write compute kernels is called **kernel language**
- OpenCL adopts C/C++-based languages to specify the kernel computations performed on the device with some restrictions and additions to facilitate efficient mapping to the heterogeneous hardware resources of accelerators.

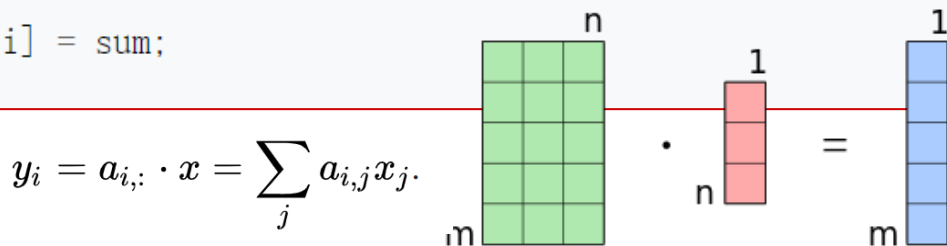
❑ OpenCL C language

- OpenCL C is a C99-based language dialect adapted to fit the device model in OpenCL
- **Memory buffers** reside in specific levels of the memory hierarchy, and pointers are annotated with the region qualifiers **__global**, **__local**, **__constant**, and **__private**, reflecting this.
- Instead of a device program having a main function, OpenCL C functions are marked **__kernel** to signal that they are entry points into the program to be called from the host program

Example

□ matrix-vector multiplication algorithm in OpenCL C

```
// Multiplies A*x, leaving the result in y.  
// A is a row-major matrix, meaning the (i,j) element is at A[i*ncols+j].  
__kernel void matvec(__global const float *A, __global const float *x,  
                    uint ncols, __global float *y)  
{  
    size_t i = get_global_id(0);           // Global id, used as the row index  
    __global float const *a = &A[i*ncols]; // Pointer to the i'th row  
    float sum = 0.f;                       // Accumulator for dot product  
    for (size_t j = 0; j < ncols; j++) {  
        sum += a[j] * x[j];  
    }  
    y[i] = sum;  
}
```





logical parallel programming models

***Mapreduce**

Programming Model for **Batch Data Processing**

- ❑ Inspired by primitives of Lisp and other **functional languages**
- ❑ Most of our computations can be represented by a *map* and *reduce* operator to each logical record in the input dataset
 - The *map* and *reduce* operator are interfaces for programmers
- ❑ We can develop a general framework **MapReduce** for batch data processing
 - Provide the powerful map/reduce interface for programmers
 - Deal with parallelization details transparently and automatically

THE SEQUENCE ADT

- ❑ Collect together all the values for a given key
- ❑ Key-value pairs
- ❑ Group items that share a **common key**

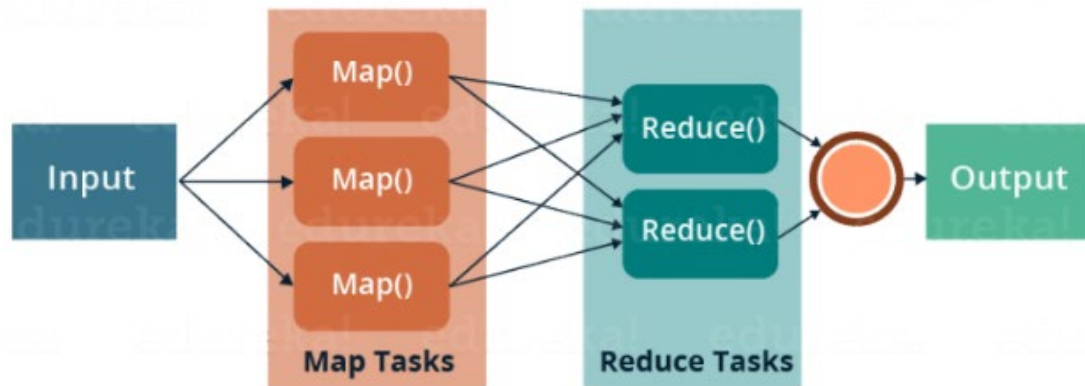
```
val Data = <("jack sprat", "15-210"),  
            ("jack sprat", "15-213"),  
            ("mary contrary", "15-210"),  
            ("mary contrary", "15-251"),  
            ("mary contrary", "15-213"),  
            ("peter piper", "15-150"),  
            ("peter piper", "15-251"), ... >
```

↓

```
val rosters = <("jack sprat", "15-210", "15-213", ...) )  
              ( "mary contrary", "15-210", "15-251", "15-213", ... )  
              ( "peter piper", "15-150", "15-251", ... )  
              .....>
```

Program Model

- ❑ Read a lot of data
- ❑ **Map**: extract something you care about from each data record
- ❑ Shuffle and sort the data
- ❑ **Reduce**: aggregate, summarize, filter, or transform the intermediate data
- ❑ Write the final results



Users can change the definition of *map* and *reduce* to fit different problems.



logical parallel programming models

***Tensorflow**

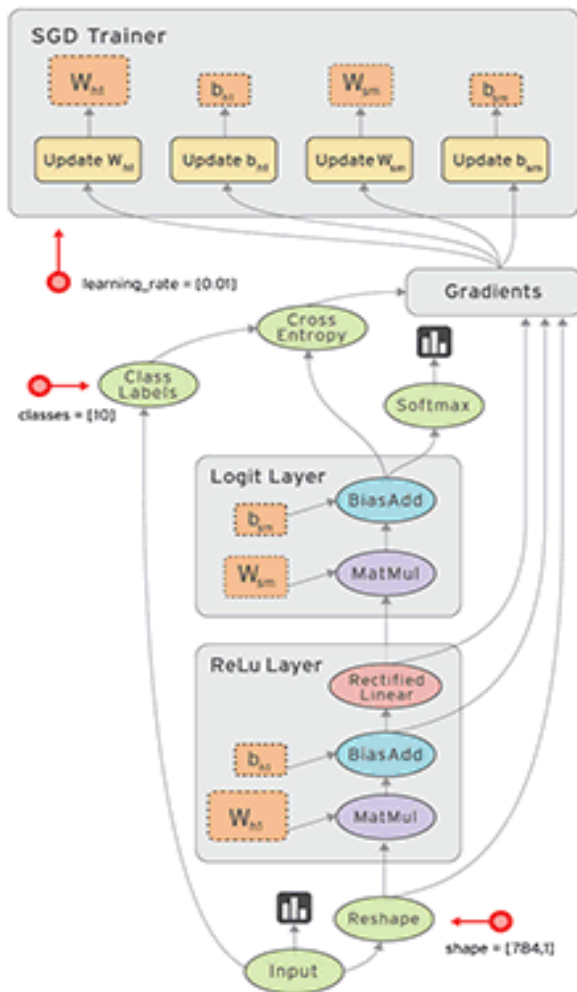
What is TensorFlow?

- ❑ From Google
- ❑ End-to-end open source platform for machine learning
- ❑ Makes it easy to create machine learning models
- ❑ **Tensor:** Data representation
- ❑ **Op:** Basic operation
- ❑ **Variable:** Input training data
- ❑ **Computational graph:** To represent computation tasks
- ❑ **Session:** To execute graph



What is **dataflow graphs**?

- A logical representation of a machine learning computation
- Represent computations as **dependencies between independent instructions**



Advantages of dataflow

- ☐ **Parallel processing** (Task partition)
- ☐ **Distributed execution** (CPU, GPU, or TPU)
- ☐ **Compile better** (XLA compiler)
- ☐ **Transportability** (independent of programming language)

Conclusion

- ❑ MIMD programming models
 - shared-memory model
 - distributed-memory model
- ❑ SIMD programming models
 - vector computing model
 - CUDA model
 - *OpenCL model
- ❑ * Mapreduce
- ❑ * Tensorflow