INTRODUCTION TO PARALLEL COMPUTING

OPENCL 2.0 UNIVERSITY TOOLKIT

Zhongliang Chen and Yash Ukidave,
Northeastern University Computer Architecture Research Lab
with
Perhaad Mistry and Dana Schaa, AMD

Instructor Notes

AMD

- These slides provide an introduction to parallel computing
- Decomposition is relevant to GPU computing since we split up tasks into kernels and decompose kernels into threads
- The topics then shift to parallel computing hardware and software models that progress into how these models combine on the GPU

Topics

AMD

- Introduction to types of parallelism
- Task and data decomposition
- Parallel computing
  - Software models
  - Hardware architectures
- Challenges using parallelism

Parallelism

AMD

- *Parallelism* describes the potential to complete multiple parts of a problem at the same time
- In order to exploit parallelism, we have to have the physical resources (i.e. hardware) to work on more than one thing at a time

Parallelism

- Amdahl's law tells us: The maximum theoretical speedup we can achieve from exploiting inherent parallelism in a problem is proportional to the ratio of serial to parallel portions, and the number of processors we have

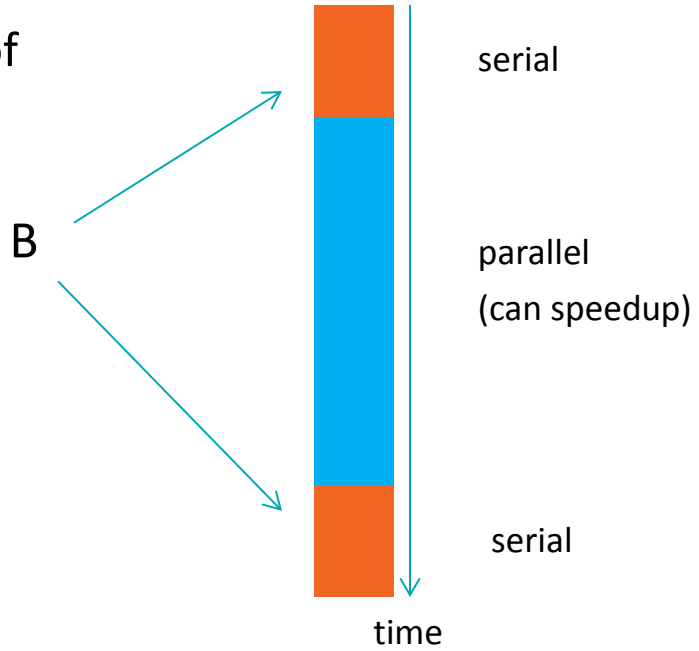
$$S = \frac{1}{B + \left(\frac{1}{n}\right)(1 - B)}$$

S = speedup

B = serial fraction of the algorithm

n = number of processors

- If an algorithm is 95% parallel (B = .05), then with a large enough n, we can approach a 20X speedup



Parallelism

AMD

- For traditional CPU architectures, we often think about *Instruction-level Parallelism (ILP)*
  - High-performance CPUs often have a large amount logic dedicated to superscalar and out-of-order hardware to exploit ILP
- For GPU computing with OpenCL, we often focus on other types of parallelism:
  - *Task parallelism* – the ability to execute different tasks within a problem at the same time
  - *Data parallelism* – the ability to execute parts of the same task (i.e. different data) at the same time
- In OpenCL, we'll see that tasks can often correspond to different kernels, and data parallelism is exploited by multiple software threads within the kernels

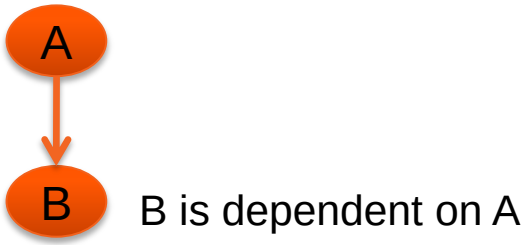
Decomposition

AMD

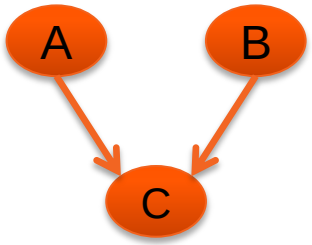
- For non-trivial problems, it helps to have more formal concepts for determining parallelism
- When we think about how to parallelize a program we use the concepts of decomposition:
  - *Task decomposition*: dividing the algorithm into individual tasks (don't focus on data)
  - *Data decomposition*: dividing a data set into discrete chunks that can be operated on in parallel

Task Decomposition

- Task decomposition reduces an algorithm to functionally independent parts
- Tasks may have dependencies on other tasks
  - If the input of task B is dependent on the output of task A, then task B is dependent on task A
  - Tasks that don't have dependencies (or whose dependencies are completed) can be executed at any time to achieve parallelism
  - *Task dependency graphs* are used to describe the relationship between tasks



B is dependent on A



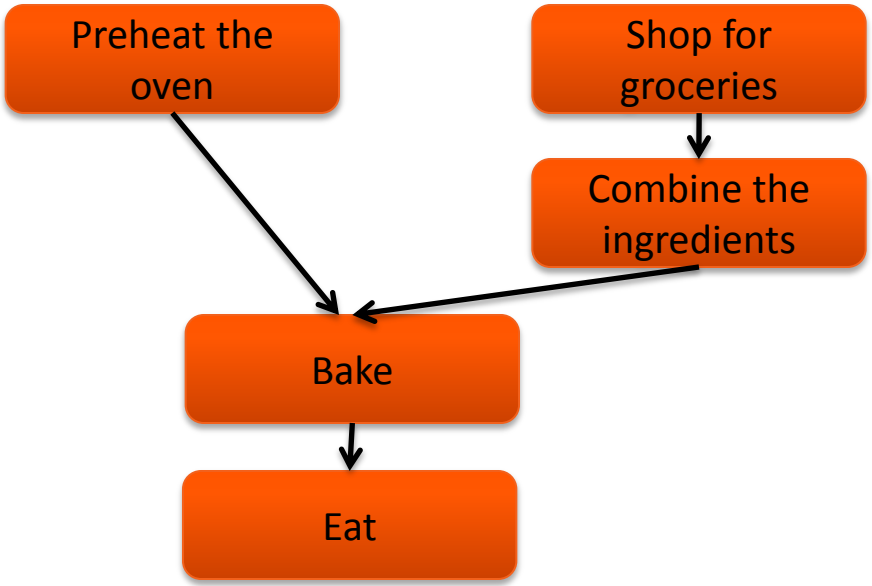
A and B are independent of each other

C is dependent on A and B

Task Dependency Graphs

AMD

- We can create a simple task dependency graph for baking cookies
  - Any tasks that are not connected via the graph can be executed in parallel (such as preheating the oven and shopping for groceries)



Data Decomposition

AMD

- Data decomposition is a technique for breaking work into multiple independent tasks, but where each task has the same responsibility
  - Each task can be thought of as processing a different piece of data
- Using data decomposition allows us to exploit data parallelism
- Using OpenCL, data decomposition allows us to easily map data parallel problems onto data parallel hardware

Output Data Decomposition

AMD

- For most scientific and engineering applications, data is decomposed based on the output data
- Examples of output decomposition
  - Each output pixel of an image convolution is obtained by applying a filter to a region of input pixels
  - Each output element of a matrix multiplication is obtained by multiplying a row by a column of the input matrices
- This technique is valid any time the algorithm is based on one-to-one or many-to-one functions

Input Data Decomposition

AMD

- Input data decomposition is similar, except that it makes sense when the algorithm is a one-to-many function
- Examples of input decomposition
  - A histogram is created by placing each input datum into one of a fixed number of bins
  - A search function may take a string as input and look for the occurrence of various substrings
- For these types of applications, each thread creates a “partial count” of the output, and synchronization, atomic operations, or another task are required to compute the final result

Parallel Computing

AMD

- The choice of how to decompose a problem is based solely on the algorithm
- However, when actually implementing a parallel algorithm, both hardware and software considerations must be taken into account

Parallel Computing

AMD

- There are hardware and software approaches to parallelism
- Much of the 1990s was spent on getting CPUs to *automatically* take advantage of Instruction Level Parallelism (ILP)
  - Multiple instructions (without dependencies) are issued and executed in parallel
  - Automatic hardware parallelization will not be considered for the remainder of the lecture
- Higher-level parallelism (e.g. threading) cannot be done automatically, so software constructs inserted by programmers or compilers tell the hardware where parallelism exists
  - In parallel programming, the programmer must choose a programming model and parallel hardware that are suited for the problem

Parallel Hardware

AMD

- Hardware is generally better suited for some types of parallelism more than others

Hardware type	Examples	Parallelism
Multi-core superscalar processors	Phenom II CPU	Task
Vector or SIMD processors	SSE units (x86 CPUs)	Data
Multi-core SIMD processors	Radeon R9 290X GPU	Data

- Currently, GPUs are comprised of many independent “processors” that have SIMD processing elements
  - One task is run at a time on the GPU*
  - *Loop strip mining* (next slide) is used to split a data parallel task between independent processors
  - Every instruction must be data parallel to take full advantage of the GPU’s SIMD hardware
    - SIMD hardware is discussed later in the lecture

*if multiple tasks are run concurrently, no inter-task communication is possible

Extracting Parallelism: Loop Strip Mining

AMD

- *Loop strip mining* is a loop-transformation technique that partitions the iterations of a loop so that multiple iterations can be:
  - executed at the same time (vector/SIMD units),
  - split between different processing units (multi-core CPUs),
  - or both (GPUs)
- An example with loop strip mining is shown in the following slides

Parallel Software – SPMD

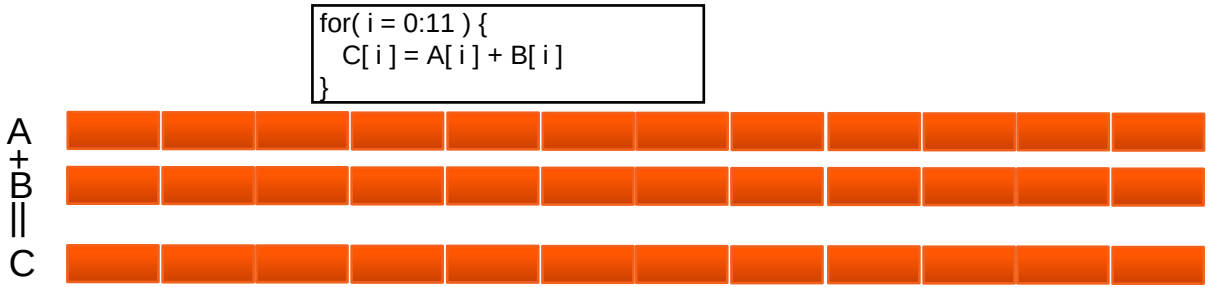
AMD

- GPU programs are called *kernels*, and are written using the Single Program Multiple Data (SPMD) programming model
  - SPMD executes multiple instances of the same program independently, where each program works on a different portion of the data
- For data-parallel scientific and engineering applications, combining SPMD with loop strip mining is a very common parallel programming technique
  - Message Passing Interface (MPI) is used to run SPMD on a distributed cluster
  - POSIX threads (pthreads) are used to run SPMD on a shared-memory system
  - Kernels run SPMD within a GPU

Parallel Software – SPMD

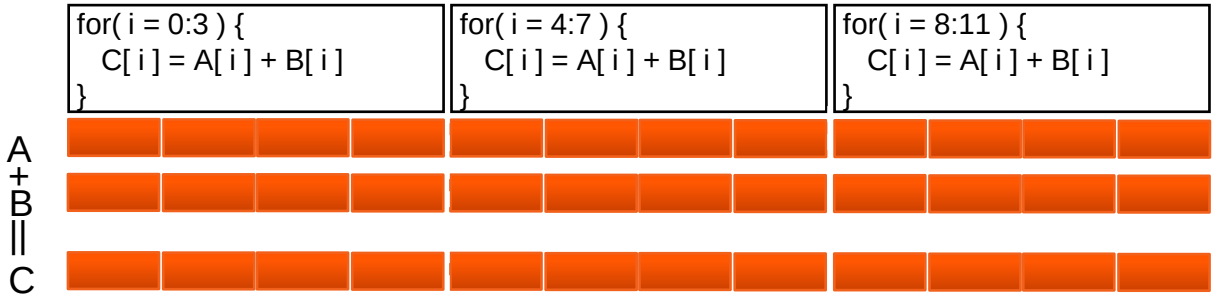
- Consider the following vector addition example

Serial program:
one program completes the entire task



- Combining SPMD with loop strip mining allows multiple copies of the same program execute on different data in parallel

SPMD program:
multiple copies of the same program run on different chunks of the data



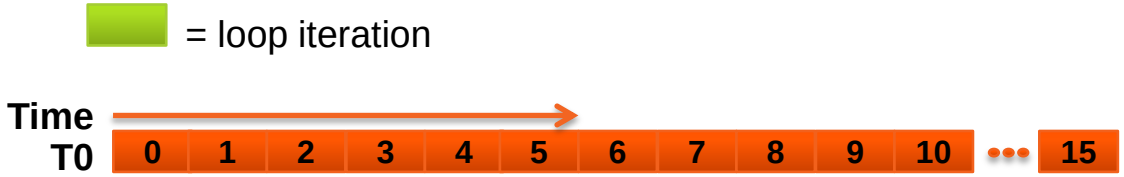
Parallel Software – SPMD

- In the vector addition example, each chunk of data could be executed as an independent thread
- On modern CPUs, the overhead of creating threads is so high that the chunks need to be large
  - In practice, usually a few threads (about as many as the number of CPU cores) and each is given a large amount of work to do
- For GPU programming, there is low overhead for thread creation, so we can create one thread per loop iteration

Parallel Software – SPMD

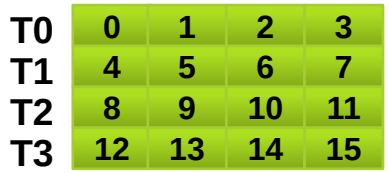
Single-threaded (CPU)

```
// there are N elements
for(i = 0; i < N; i++)
  C[i] = A[i] + B[i]
```



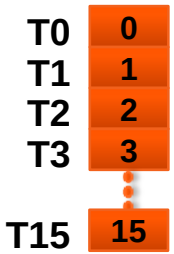
Multi-threaded (CPU)

```
// tid is the thread id
// P is the number of cores
for(i = tid*(N/P); i < (tid+1)*N/P; i++)
  C[i] = A[i] + B[i]
```



Massively Multi-threaded (GPU)

```
// tid is the thread id
C[tid] = A[tid] + B[tid]
```



Parallel Hardware – SIMD

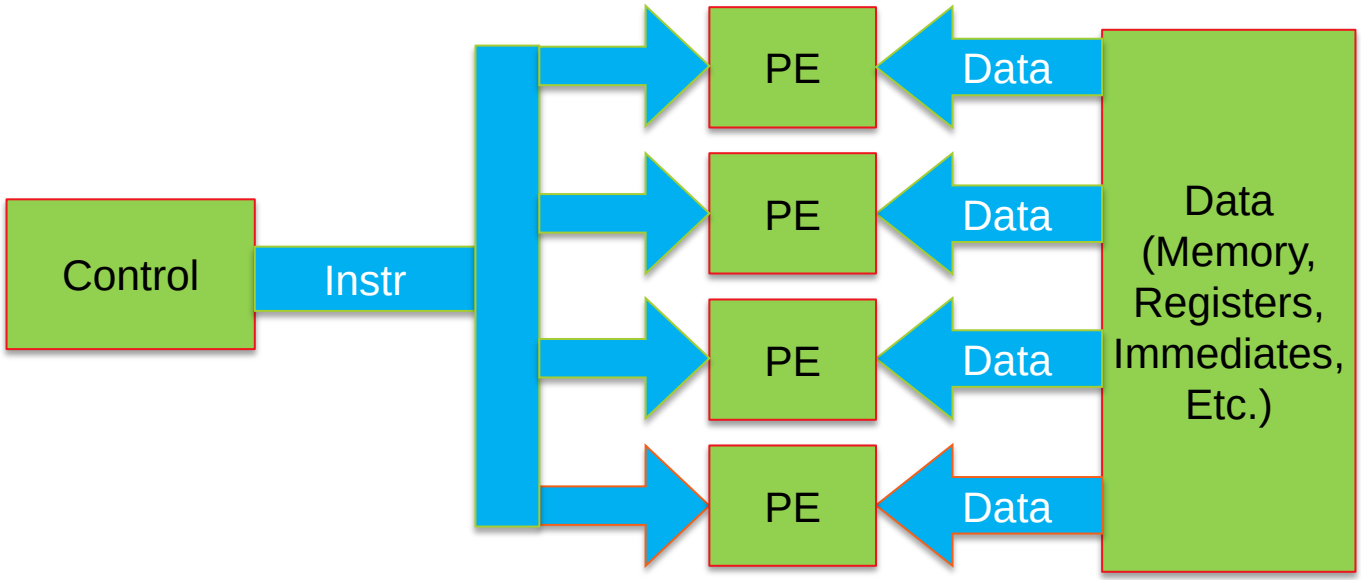
AMD

- Each processing element of a Single Instruction Multiple Data (SIMD) processor executes the same instruction with different data at the same time
  - A single instruction is issued to be executed simultaneously on many ALU units
  - We say that the number of ALU units is the *width* of the SIMD unit
- SIMD processors are efficient for data parallel algorithms
  - They reduce the amount of control flow and instruction hardware in favor of ALU hardware

Parallel Hardware – SIMD

AMD

- A SIMD hardware unit



Parallel Hardware – SIMD

AMD

- In the vector addition example, a SIMD unit with a width of four could execute four iterations of the loop at once
- All current GPUs are based on SIMD hardware
  - The GPU hardware implicitly maps each SPMD thread to a SIMD “core”
    - The programmer does not need to consider the SIMD hardware for correctness, just for performance
  - This model of running threads on SIMD hardware is often referred to as Single Instruction Multiple Threads (SIMT)

Challenges of Parallelization

AMD

- On CPUs, hardware-supported atomic operations enable concurrency
  - Atomic operations allow data to be read and written without intervention from another thread

- GPUs support some system-wide atomic operations, but with a large performance trade-off
  - Usually code that requires global synchronization is not well suited for GPUs (or should be restructured)
  - Any problem that is decomposed using input data partitioning (i.e., requires results to be combined at the end) will likely need to be restructured to execute well on a GPU

Summary

AMD

- Choosing appropriate parallel hardware and software models is highly dependent on the problem we are trying to solve
  - Problems that fit the output data decomposition model are usually mapped fairly easily to data-parallel hardware

- Naively, OpenCL's parallel programming model is easy because it is simplified SPMD programming
  - We can often map iterations of a for-loop directly to OpenCL work-items
  - However, we will see that obtaining high performance requires thorough understanding of hardware (incorporating hardware parallelism + memory subsystem), and complicates the programming model