GPU Microarchitecture Note Set 1—Basic Concepts

Parallel Computation

Performance Measures

Big Cores v. Little Cores

Latency v. Throughput

Parallel Computation

The Idea:

One computer takes t seconds to run a program, which is not fast enough . . .
. . . so try to use n computers to get the program to run in t/n seconds . . .
. . . choose n to fit your performance goal and budget.

Easier said than done.

Parallel Computation:
The use of multiple processor cores to speed the execution of a program.

A *parallel program* consists of multiple *threads* that will execute on a *parallel system* consisting multiple *cores*.

The goal is to lower execution time by using multiple cores.

Realizing this goal is often **frustrated** by the **difficulty** of parallel programming.

Parallel Computation and GPUs

Modern multi-core CPUs and GPUs are both parallel computers.

GPUs are **much more parallel** than CPUs.

For reasons to be described. . .
. . . this makes GPUs **more efficient** for an important **class of problems**.

But it also makes them **more difficult to programn**.

Coverage of Topics In Parallel Computation

These notes will only consider single-chip parallel systems:

Multi-core CPUs.

Many-core CPUs.

GPUs.

The following types of parallel systems are beyond the scope of this course:

Multi-chip multiprocessors.

Multi-node computing clusters.

Including LSU's *Tezpur* and *SuperMike II* clusters. . .
. . . and LONI's *Queen Bee* cluster in downtown Baton Rouge.

Definitions

Thread:

A path through the program defined by the programmer, compiler, or some piece of support software.

The first program you wrote probably consisted of a single thread.

Programs start execution with a single thread, then create additional threads to share the work.

A program with multiple threads is a parallel program.

Core:

A processor, what was once called a computer CPU. A core consists of registers, caches, hardware for fetching and decoding instructions, functional units for executing instructions, etc. Certain resources, such as fetch and decode logic might be shared with other cores.

A multi-core chip, of course, contains multiple cores.

A core occupies a certain area, A, on a chip …
… the area is an important factor when considering efficiency.

A core dissipates a certain power, P …
… power consumption is also important.

A chip has a certain area and power limit…
… these limit the number of cores on a chip.

Heavy Weight Core:
A core designed to execute a single thread quickly.

Heavy weight cores have large area and high power consumption.

Energy per instruction is high.

General-purpose CPUs, such as those found in home computers, consist of heavy weight cores.

Light Weight Core:
A core designed for efficiency.

Light weight cores have small area.

Energy per instruction is low.

Execution of Multithreaded Programs

Consider a system with c cores and a program with r threads.

Typically the OS will distribute the r threads evenly over the c cores.

If $c < r$ then $c - r$ cores will sit idle.

If $c > r$ then a core may have more than on thread assigned.

Performance Characterization

Consider

A parallel program that can spawn any number of threads, as needed.

A computer consisting of c cores.

Let $t(1)$ denote the execution time on 1 core.

The value is determined by the single-thread performance of the core.

Let $t(c)$ denote the execution time on c cores.

The value is determined by the parallel program and by $t(1)$.

Speedup:

[of a parallel program on parallel system]. The ratio of execution time on one core to the time on the entire system.

Using the notation above:

$$S = \frac{t(1)}{t(c)}.$$

For example:

A program runs in 10 s on one core and 3 s on 5 cores.

The speedup is then $S = \frac{10\,\mathrm{s}}{3\,\mathrm{s}} = 3.33$.

Speedup Special Cases

Speedup Case: Linear Speedup— $S = c$.

This occurs when $t(c) = t(1)/c$.

This indicates no duplication of effort by threads, no time lost to communication.

There are some programs with linear speedup...
... but for many others the speedup is lower.

Example:

A program runs in 10 s on one core and is to be run on 5 cores.

To achieve linear speedup it would need to run in $10\,\mathrm{s}/5 = 2\,\mathrm{s}$.

Speedup Case: No Speedup— $S = 1$.

This occurs when $t(c) = t(1)$ (for $c > 1$).

This might be the programmer's fault, or an inherent property of the problem.

Speedup Case: Serial Limiter— $S = c/(cf + 1 - f)$

This is sometimes referred to as *Amdahl's Law*.

This applies to a program that can be split into two parts…
… a part with linear speedup…
… and a part with no speedup (the *serial* portion).

Symbol f is the fraction of the program with linear speedup.

When $f = 1$, all of the program enjoys linear speedup;…
… when $f = 0$, none.

Heavy Cores v. Light Cores

Single-Thread Performance

Why not avoid parallel programming by having a really fast core?

Short answer: diminishing returns.

First, consider the budget:

A chip has a certain area.

Limited by dollar cost and technology.

A chip has a power limit.

Determined by ability to cool chip.

Label either one as "cost."

Performance of Core v. Cost

Curve drops from linear early.

For linear portion of curve, bigger core is an easy choice.

The distance between the linear reference and performance curve...
... is the penalty for avoiding parallel programming.

Implications

Parallel programming can't be avoided.

Program Needs and System Capabilities

Our Goal:

Determine whether we should be **satisfied**. . .
. . . with **the performance** of our parallel program. . .
. . . on our system.

We are not satisfied when the program runs **more slowly** then we **expect**.

Knowing when to be satisfied is a **key skill**.

The Idea:

Estimate the computation needs of our problem, algorithm, or program.

For example, 10^{20} floating-point operations.

Determine the computation capabilities of our system.

For example, 10^{17} FLOPS.

Use these to estimate execution time.

For example, $10^{20}/10^{17} = 1000\,\mathrm{s}$.

If the estimate does not match the measured value, find the cause.

Computation Needs

These refer to a program, algorithm, or problem.

Determined by analyzing a problem, algorithm, or program.

Common Computation Needs:

Floating-Point Operations

Data Transfer (bytes read from and written to memory).

Instruction Count (Number of executed instructions)

Computation Capabilities

These refer to the capabilities of a system (CPU, GPU, etc.)

Common Computation Capabilities

Floating Point Execution Rate - FLOPS

Number of floating-point operations divided by amount of time.

Instruction Execution Rate - IPC

Number of executed instructions divided by amount of time.

Data Transfer Bandwidth - B/s

Number of bytes crossing some line, divided by amount of time.

Floating Point Operation Need and Capability

Commonly used for scientific programs…

… because such operations are considered essential (can't be avoided) and…

… because of the historic high cost of FP hardware.

Consider:

```
double *x, *a, *b, *c;
for ( i=0; i<1e9; i++ ) x[i] = a[i] + b[i] * c[i];
```

Uses $10^9 \times 2 = 2 \times 10^9$ FP operations.

Consider a system capable of 100 GFLOPS (10^{11} FLOPS).

Execution time bound is 20 ms.

If measured execution time was 100 ms we would **not be satisfied**.

Data Transfer Need and Capability

Used for all kinds of programs.

Easy to measure when each data item read exactly once.

Otherwise, need to account for cache and scratchpad performance.

Consider:

```
double *x, *a, *b;
for ( i=1; i<1e9-1; i++ ) x[i] = a[i-1] * l + a[i] + a[i+1] * r;
```

The size of a double is 8 bytes.

In an iteration, three elements of `a` are accessed. . .
. . . two were accessed in a prior iteration. . .
. . . and so shouldn't need to be read from memory. . .
. . . leaving one new element of `a` accessed per iteration.

Total data transfer needs: $10^9 \times 8 \times 2 = 16\,\mathrm{GB}$.

Suppose system can transfer 30 GB/s.

Performance bound: $16/30 = 533\,\mathrm{ms}$.

Data Transfer Example, Continued.

Suppose measured execution time were just 1.07 s. We are not satisfied.

Maybe we were wrong about each element of `a` being read once.

Maybe we should look at number of FP operations.

Maybe we should look at number of instructions.

Instruction Count and Execution Rate

Considered a less fundamental bound than FLOPS and data transfer.

Number of instructions determined by many factors:

- How program is written.
- Quality of compiler.
- Instruction set of processor.

Performance Estimation

With above needs and capabilities can compute three performance bounds.

Actual performance will be no greater than the smallest of these.

Latency and Throughput

Latency and Throughput

Commonly used types of performance measure...

... for describing components or whole systems.

Latency:

The time needed to do something from start to finish.

In discussion of GPUs, latency often refers...

... to the execution time of a single instruction or of a single thread.

A processor is called *latency oriented* ...

... if it has low latency (runs a single thread quickly).

Throughput:
The number of things completed per unit time.

Throughput can refer to many possible things, including:

Instructions per cycle (or second).

FLOPS.

Vertices per second. (For graphics.)

A processor is called *throughput oriented* if it has high throughput.

Latency v. Throughput

Can’t Have Both