The Evolution of High Performance Computers

David Chisnall

February 8, 2011

Outline

Classic Computer Architecture

Execute?

1. Read value from memory
2. Store in register

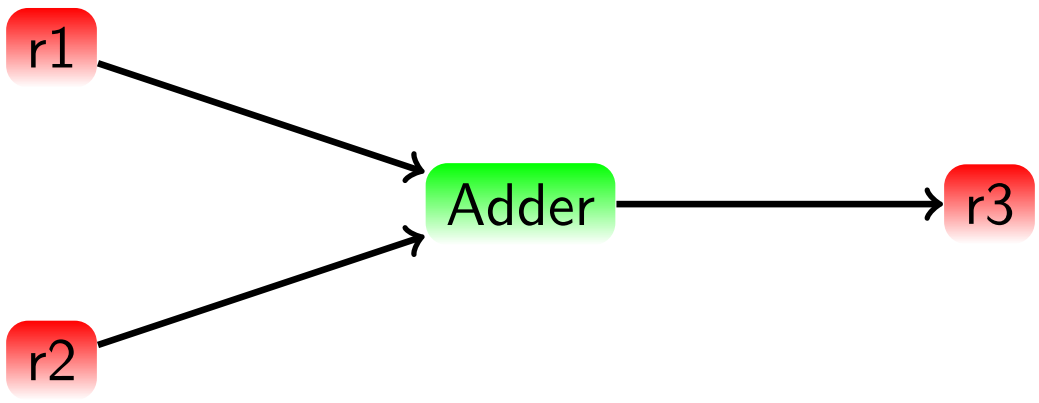
1. Read value from register
2. Store in memory

1. Read value(s) from register(s)
2. Perform calculation
3. Store result in register

Calculation?

- Add or subtract?
- Shift?
- Multiply?
- Divide?
- Evaluate polynomial?
- Discrete Cosine Transform?
- Exact set of operations depends on the computer

A Simple Case



Making it Faster: Vectors

- Each register stores multiple values
- Operations on components

Example Problem

- Pixel values: 32 bits per channel, red, green, blue, alpha
- Lighten operation scales each value by a constant factor
- Same operation on each of the colour channels

Scalar Solution

```
for (int i=0 ; i<imageSize ; i++)
{
    pixels[i].red *= 1.1;
    pixels[i].green *= 1.1;
    pixels[i].blue *= 1.1;
}
```

- Three loads per pixel
- Three multiplies per pixel
- Three stores per pixel

Vector Solution

```
pixel_t scale = {1.1, 1.1, 1.1, 1};
for (int i=0 ; i<imageSize ; i++)
{
    pixel[i] *= scale;
}
```

- One (vector) load per pixel
- One (vector) multiply per pixel
- One (vector) store per pixel
- Redundant multiply ($alpha \times 1$) costs nothing, but makes potential speedup only 3, not 4 for a 4-element vector unit

Why is this faster?

- One fetch, one decode, four executes
- Fewer instructions, less space used for storing the program

Memory Layout

- For vectors to be fast, you must be able to load vectors from memory
- If you separate the channels, this would not be possible
- Modern compilers can *auto-vectorise* code with a usable memory layout

Even Faster: Parallel Streams

- Each loop iteration is independent
- We can do them in parallel

Bigger Vectors?

```
pixel_pair_t scale = {1.1, 1.1, 1.1, 1, 1.1,
    1.1, 1.1, 1};
for (int i=0 ; i<imageSize ; i+=2)
{
     pixels[i] *= scale;
}
```

- Now we can process two pixels at once
- Some modern systems have 256-bit vectors
- Really hard to map most problems to them
- Most vector units are only 128-bit

Independent Processors?

- Split loop into two or more loops
- Run each on a different processor

Hybrid Solution: Semi-independent Cores

- Each core runs a group of threads
- When threads are executing the same thing, all run
- When they branch, only one runs
- Typical GPU architecture

What the Programmer Sees

- One program (kernel) per input (e.g. per pixel)
- Conceptually all run in parallel
- In practice, 1–128 run in parallel

How it Works

- Processor is $n \times m$ element vector processor
- Programmer sees n element vector processor
- Processor starts m elements at once
- Linear code Just Work^{TM}
- Special case for branches:
- All threads take the same branch, all continue to run
- Some take a different branch, paused, resumed later

Bad Example

```
__kernel void stripe(const float4 *input
                     global float4 *output)
{
    int i = get_global_id(0);
    // Lighten even pixels, darken odd pixels
    if (i % 2)
    {
        output[i] = input[i] * 1.1;
    }
    else
    {
        output[i] = input[i] * 0.9;
    }
}
```

- Each pair of threads will take different branches
- Only half will actually be running in parallel

Why it's Useful

- Good for code with highly independent elements
- Higher ratio of execution units to other components than a general purpose CPU

When is it not Useful?

- Problems that don't have independent inputs
- Algorithms that have lots of branches

So, Branches are Fast on CPUs?

- Fetch, decode, execute, is an oversimplification
- Modern pipelines are 7–20 stages
- Can't fetch instruction after the branch until after executing the branch!
- P4 can have 140 instructions executing at once
- Can have none executing if it doesn't know what to execute though...

Not So Much Then?

- Typical CPUs do *branch prediction*
- Correct prediction is very fast
- Incorrect prediction is very slow
- Accuracy is about 95%
- So 5% of branches cause a pipeline stall (bad!)

Another Work-Around: Predicated Instructions

- Found in ARM and Itanium
- Instruction predicated on condition register
- Executes anyway
- Result is only stored if the condition was set
- No pipeline stall, but some redundant work
- Much faster for short branches

Avoiding Branches: Loop Unrolling

What are funroll loops?

```
for (int i=0 ; i<imageSize ; i++)
{
    pixels[i] *= scale;
}
```

- One branch per pixel
- Should be correctly predicted...
- But still wastes some time

Avoiding Branches: Loop Unrolling

What are funroll loops?

```
for (int i=0 ; i<imageSize ; i++)
{
    pixels[i] *= scale;
    pixels[i++] *= scale;
    pixels[i++] *= scale;
    pixels[i++] *= scale;
}
```

- One branch per four pixels
- Code is bigger, but should be faster

Don't do this yourself! The compiler will do it for you!

Gotcha: Short-Circuit Evaluation

```
if (a() || b() || c())
    doStuff();
```

What this really means:

```
if (a())
    doStuff();
else if (b())
    doStuff();
else if (c())
    doStuff();
```

One `if` statement, three branches!

Avoiding Short-Circuiting

```
int condition = a();
condition = b() || condition;
condition = c() || condition;
if (condition)
doStuff();
```

- All of the subexpressions are evaluated.

Questions?