Vector and Array Processors

Moving on from distributed: the next major architecture to consider is SIMD
Moving on from distributed: the next major architecture to consider is SIMD

Recall: these have many processors all executing the same thing on different data
Vector and Array Processors

Moving on from distributed: the next major architecture to consider is SIMD

Recall: these have many processors all executing the same thing on different data

First we need to recall the SIMD architecture and go through the issues it brings
Vector and Array Processors

All processors are controlled by just one Control unit, so are all executing the same instruction.
Vector and Array Processors

All processors are controlled by just one Control unit, so are all executing the same instruction

This is data parallelism
Each processor has its own chunk of *private memory*, and a shared chunk of *global memory*.
Vector and Array Processors

Each processor has its own chunk of *private memory*, and a shared chunk of *global memory*

Processors can be strung linearly in a *vector* or in a square mesh as an *array*
Of course, you can use an array as a vector or a vector as an array, with a modest loss of efficiency.
Of course, you can use an array as a vector or a vector as an array, with a modest loss of efficiency
Vector and Array Processors

Vector processors appeared quite early on in computer architectures (1960s) and were a mainstay in 1980s supercomputers (Crays), as they are a relatively simple extension of the uniprocessor.
Vector and Array Processors

Vector processors appeared quite early on in computer architectures (1960s) and were a mainstay in 1980s supercomputers (Crays), as they are a relatively simple extension of the uniprocessor.

Array processors have come into fashion and gone away again several times.
Vector and Array Processors

Vector processors appeared quite early on in computer architectures (1960s) and were a mainstay in 1980s supercomputers (Crays), as they are a relatively simple extension of the uniprocessor.

Array processors have come into fashion and gone away again several times.

GPUs owe a lot to array processor design: more on this later.
Vector and Array Processors

Vector processors appeared quite early on in computer architectures (1960s) and were a mainstay in 1980s supercomputers (Crays), as they are a relatively simple extension of the uniprocessor.

Array processors have come into fashion and gone away again several times.

GPUs owe a lot to array processor design: more on this later.

Similarly, SWAR instructions owe something to vector design.
The basic idea of SIMD is that we can parallelise loops like

```c
for (i = 0; i < 1024; i++) {
    c[i] = a[i] + b[i];
}
```

as

```c
in parallel do c[i] = a[i] + b[i];
```
Vector and Array Processors

The basic idea of SIMD is that we can parallelise loops like

\[
\text{for } (i = 0; i < 1024; i++) \{ \\
\quad c[i] = a[i] + b[i]; \\
\}
\]

as

\[
\text{in parallel do } c[i] = a[i] + b[i];
\]

Exercise. Go back and look at SWAR; and OpenMP
Vector and Array Processors

The important points being

• all elements in the arrays are being treated identically
• there is no interference between any of the operations
• there are no dependencies across iterations of the loop

So no races, thus no serialisation of the operations is needed
Vector and Array Processors

The important points being

- all elements in the arrays are being treated identically
The important points being

- all elements in the arrays are being treated identically
- there is no interference between any of the operations
The important points being

- all elements in the arrays are being treated identically
- there is no interference between any of the operations
- there are no dependencies across iterations of the loop

So no races, thus no serialisation of the operations is needed
What if there are conflicts? For example

```c
for (i = 1; i < 1024; i++) {
    a[i] = a[i] + a[i-1];
}
```

Here, the new value of \(a[i]\) depends on the value of \(a[i-1]\); which will have been updated in the previous iteration of the loop.
What if there are conflicts? For example

```c
for (i = 1; i < 1024; i++) {
    a[i] = a[i] + a[i-1];
}
```

Here, the new value of $a[i]$ depends on the value of $a[i-1]$; which will have been updated in the previous iteration of the loop.

In comparison

```
in parallel do a[i] = a[i] + a[i-1];
```

takes the original value of $a[i-1]$
Vector and Array Processors

Starting with $a = 1, 1, 1, 1$; the sequential loop gives
Vector and Array Processors

Starting with \( a = 1, 1, 1, 1 \); the sequential loop gives
1 2 1 1

While the parallel version gives
1 1 1 1
1 1 1 +
1 2 2 2
This is due to the nature of the original loop: it is actually a prefix scan operation.
Prefix scans can be done SIMD, but when parallelising code you have to be aware that is what is happening!
Vector and Array Processors

Starting with $a = 1, 1, 1, 1$; the sequential loop gives

```
1  2  1  1
1  2  3  1
```

This is due to the nature of the original loop: it is actually a prefix scan operation.
Prefix scans can be done SIMD, but when parallelising code you have to be aware that is what is happening!
Vector and Array Processors

Starting with $a = 1, 1, 1, 1$; the sequential loop gives

1 2 1 1
1 2 3 1
1 2 3 4

While the parallel version gives

1 1 1 1
1 1 1 +
1 2 2 2

This is due to the nature of the original loop: it is actually a prefix scan operation.

Prefix scans can be done SIMD, but when parallelising code you have to be aware that is what is happening!
Vector and Array Processors

Starting with $a = 1, 1, 1, 1$; the sequential loop gives

\[
\begin{array}{cccc}
1 & 2 & 1 & 1 \\
1 & 2 & 3 & 1 \\
1 & 2 & 3 & 4 \\
\end{array}
\]

While the parallel version gives

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
\hline
1 & 1 & 1 & + \\
\hline
1 & 2 & 2 & 2 \\
\end{array}
\]
Vector and Array Processors

Starting with $a = 1, 1, 1, 1$; the sequential loop gives

\[
\begin{align*}
1 & \quad 2 & \quad 1 & \quad 1 \\
1 & \quad 2 & \quad 3 & \quad 1 \\
1 & \quad 2 & \quad 3 & \quad 4
\end{align*}
\]

While the parallel version gives

\[
\begin{align*}
1 & \quad 1 & \quad 1 & \quad 1 \\
1 & \quad 1 & \quad 1 & \quad + \\
\underline{1} & \underline{1} & \underline{1} & \underline{+} \\
1 & \quad 2 & \quad 2 & \quad 2
\end{align*}
\]

This is due to the nature of the original loop: it is actually a \textit{prefix scan} operation
Vector and Array Processors

Starting with $a = 1, 1, 1, 1$; the sequential loop gives

\[
\begin{array}{cccc}
1 & 2 & 1 & 1 \\
1 & 2 & 3 & 1 \\
1 & 2 & 3 & 4 \\
\end{array}
\]

While the parallel version gives

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & + \\
1 & 2 & 2 & 2 \\
\end{array}
\]

This is due to the nature of the original loop: it is actually a *prefix scan* operation.

Prefix scans can be done SIMD, but when parallelising code you have to be aware that is what is happening!
Vector and Array Processors

Having given a warning, SIMD processing is very powerful
Vector and Array Processors

Having given a warning, SIMD processing is very powerful.

Vectors and arrays with thousands of processors are common.
Vector and Array Processors

Having given a warning, SIMD processing is very powerful.

Vectors and arrays with thousands of processors are common.

If your problem is data parallel, it can get huge speedups by running SIMD.
Vector and Array Processors

Having given a warning, SIMD processing is very powerful.

Vectors and arrays with thousands of processors are common.

If your problem is data parallel, it can get huge speedups by running SIMD.

If you can get your data to the individual processors fast enough.
Vector and Array Processors

In SIMD the processing power is not the problem: it’s the data movement
Vector and Array Processors

In SIMD the processing power is not the problem: it’s the data movement

With thousands of processors, CPU is essentially free
In SIMD the processing power is not the problem: it’s the data movement.

With thousands of processors, CPU is essentially free.

The major way to lose efficiency is through data movement.
Vector and Array Processors

As usual, the bus bandwidths between the processors and between the global memory and the processors is much less than you might wish.
Vector and Array Processors

As usual, the bus bandwidths between the processors and between the global memory and the processors is much less than you might wish.

The total *aggregate* bandwidth, adding together all the individual bandwidths of all the buses can be huge, but this is a useless statistic (thus is given by marketing).
Vector and Array Processors

As usual, the bus bandwidths between the processors and between the global memory and the processors is much less than you might wish.

The total *aggregate* bandwidth, adding together all the individual bandwidths of all the buses can be huge, but this is a useless statistic (thus is given by marketing).

Careful overlapping of communications and processing is the way to make these systems work at their best efficiency.
Vector and Array Processors

As usual, the bus bandwidths between the processors and between the global memory and the processors is much less than you might wish.

The total *aggregate* bandwidth, adding together all the individual bandwidths of all the buses can be huge, but this is a useless statistic (thus is given by marketing).

Careful overlapping of communications and processing is the way to make these systems work at their best efficiency.

Thus, for example, rather than waiting for a read from memory to return a value, go away and do some other computation while the read is being processed.
Vector and Array Processors

As usual, the bus bandwidths between the processors and between the global memory and the processors is much less than you might wish.

The total aggregate bandwidth, adding together all the individual bandwidths of all the buses can be huge, but this is a useless statistic (thus is given by marketing).

Careful overlapping of communications and processing is the way to make these systems work at their best efficiency.

Thus, for example, rather than waiting for a read from memory to return a value, go away and do some other computation while the read is being processed.

This kind of asynchronous programming improves efficiency but is much harder to do and to get right.
Vector and Array Processors

Programming a vector or array machine can be interesting
Programming a vector or array machine can be interesting

The Fortran language turns out to be fairly well suited to compilation for SIMD
Vector and Array Processors

Programming a vector or array machine can be interesting

The Fortran language turns out to be fairly well suited to compilation for SIMD

Because the memory model of (early) Fortran is so simple (no pointers, etc.) it is fairly straightforward for a compiler to analyse a Fortran program
Vector and Array Processors

Programming a vector or array machine can be interesting.

The Fortran language turns out to be fairly well suited to compilation for SIMD.

Because the memory model of (early) Fortran is so simple (no pointers, etc.) it is fairly straightforward for a compiler to analyse a Fortran program.

Loops like the first example above can be spotted and converted automatically into SIMD code.
Vector and Array Processors

Programming a vector or array machine can be interesting

The Fortran language turns out to be fairly well suited to compilation for SIMD

Because the memory model of (early) Fortran is so simple (no pointers, etc.) it is fairly straightforward for a compiler to analyse a Fortran program

Loops like the first example above can be spotted and converted \textit{automatically} into SIMD code

For other languages, particularly C, this is harder due to \textit{aliasing}
In $c[i] = a[i] + b[i]$; we can’t in general be sure that $a$, $b$ and $c$ all refer to distinct areas of memory.
In $c[i] = a[i] + b[i]$; we can’t in general be sure that $a$, $b$ and $c$ all refer to distinct areas of memory.

If they are not distinct, e.g., $c = a$ makes $c$ an alias for $a$, we can get hidden interference.
In $c[i] = a[i] + b[i]$; we can’t in general be sure that $a$, $b$ and $c$ all refer to distinct areas of memory.

If they are not distinct, e.g., $c = a$ makes $c$ an alias for $a$, we can get hidden interference.

If the arrays do not overlap, we can do all the assignments simultaneously and get the same results as a sequential execution.
In $c[i] = a[i] + b[i]$; we can’t in general be sure that $a$, $b$ and $c$ all refer to distinct areas of memory

If they are not distinct, e.g., $c = a$ makes $c$ an alias for $a$, we can get hidden interference

If the arrays do not overlap, we can do all the assignments simultaneously and get the same results as a sequential execution

If they do overlap, we would get different results; e.g., $c=a$ and $b=a-1$ gives us the prefix scan we saw earlier
Automatic Parallelism Extraction

In $c[i] = a[i] + b[i]$; we can’t in general be sure that $a$, $b$ and $c$ all refer to distinct areas of memory.

If they are not distinct, e.g., $c = a$ makes $c$ an alias for $a$, we can get hidden interference.

If the arrays do not overlap, we can do all the assignments simultaneously and get the same results as a sequential execution.

If they do overlap, we would get different results; e.g., $c = a$ and $b = a - 1$ gives us the prefix scan we saw earlier.

In Fortran, being a simpler language, this kind of interference is usually impossible.
Early supercomputers were programmed in Fortran
Automatic Parallelism Extraction

Early supercomputers were programmed in Fortran

If they had a vector processor, the compiler could automatically extract the parallelism and generate SIMD code
Automatic Parallelism Extraction

Early supercomputers were programmed in Fortran

If they had a vector processor, the compiler could automatically extract the parallelism and generate SIMD code

More modern languages, by adding pointers (or equivalent), are much harder to analyse
Early supercomputers were programmed in Fortran

If they had a vector processor, the compiler could automatically extract the parallelism and generate SIMD code

More modern languages, by adding pointers (or equivalent), are much harder to analyse

Note that later Fortran standards add pointer types...
A C compiler cannot correctly automatically compile

```c
void add(double a[], double b[], double c[], int n) {
    int i;
    for (i = 0; i < n; i++) a[i] = b[i] + c[i];
}
```

to run in parallel (SIMD or MIMD or otherwise), as it cannot tell if the arrays overlap or not
A C compiler cannot correctly automatically compile

```c
void add(double a[], double b[], double c[], int n)
{
    int i;
    for (i = 0; i < n; i++) a[i] = b[i] + c[i];
}
```

to run in parallel (SIMD or MIMD or otherwise), as it cannot tell if the arrays overlap or not

(C.f., `memcpy` and `memmove` in the C library)
So, C (C99 standard) has added the `restrict` keyword to mean “there are no aliases to this name”: this is purely as an aid to the compiler to help it create more efficient code.
So, C (C99 standard) has added the `restrict` keyword to mean “there are no aliases to this name”: this is purely as an aid to the compiler to help it create more efficient code.

```c
void add(double * restrict a, double * restrict b,
         double * restrict c, int n)
{
    int i;
    for (i = 0; i < n; i++) a[i] = b[i] + c[i];
}
```

can be compiled to run in parallel.
Automatic Parallelism Extraction

Note that the `restrict` keyword is promise from the programmer to the compiler that they won’t use the given function with aliased arguments.
Automatic Parallelism Extraction

Note that the *restrict* keyword is promise from the programmer to the compiler that they won’t use the given function with aliased arguments.

It is very difficult or impossible for the compiler to enforce this: so if the programmer gets it wrong, and calls the function on aliased arguments, the result will likely be incorrect.
Automatic Parallelism Extraction

Note that the `restrict` keyword is a promise from the programmer to the compiler that they won’t use the given function with aliased arguments.

It is very difficult or impossible for the compiler to enforce this: so if the programmer gets it wrong, and calls the function on aliased arguments, the result will likely be incorrect.

Again, there is little support in C to stop the programmer being stupid.
Automatic Parallelism Extraction

Note that the `restrict` keyword is promise from the programmer to the compiler that they won’t use the given function with aliased arguments.

It is very difficult or impossible for the compiler to enforce this: so if the programmer gets it wrong, and calls the function on aliased arguments, the result will likely be incorrect.

Again, there is little support in C to stop the programmer being stupid.

There do exist explicitly SIMD languages where the programmer indicates the parallelism: we shall look at CUDA later.