Vector Machines

Vectorization

\[
\mathbf{a} \cdot \mathbf{v} = a \cdot \left( \begin{array}{c} x \\ y \\ z \end{array} \right) = \left( \begin{array}{c} a \cdot x \\ a \cdot y \\ a \cdot z \end{array} \right)
\]

**Type Declarations**

- `type Real_Precision = Float`
- `type Scalar = Real_Precision`
- `type Vector = [Real_Precision]`

**Function Definitions**

- `scale :: Scalar -> Vector -> Vector`
- `scale scalar vector = map (scalar *) vector`

Executed sequentially.

Potentially concurrent, yet:
Vector Machines

Vectorization

\[ \mathbf{a} \cdot \mathbf{v} = a \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a \cdot x \\ a \cdot y \\ a \cdot z \end{pmatrix} \]

**Buzzword collection:** AltiVec, SPE, MMX, SSE, NEON, SPU, AVX, ...

**Translates into CPU-level vector operations**

\[ \mathbf{a} \cdot \mathbf{v} = a \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a \cdot x \\ a \cdot y \\ a \cdot z \end{pmatrix} \]

Functions are "promoted"

**Combined with in-lining, loop unrolling and caching, this is as fast as a single CPU will get.**

```
import Control.Parallel.Strategies

type Real = Float

type Vector = [Real]

scale :: Vector -> Vector
scale vector = parMap rpar (scalar *) vector
```

```
const Index = {1 .. 10000000},
Vector_1 : [Index] real = 1.0,
Scale    : real = 5.1,
Scaled   : [Vector] real = Scale * Vector_1;
```

```
type Real = digits 15;
type Vectors is array (Positive range <>) of Real;

function Scale (Scalar : Real; Vector : Vectors) return Vectors is
    Scaled_Vector : Vectors (Vector'Range);
    begin
        for i in Vector'Range loop
            Scaled_Vector (i) := Scalar * Vector (i);
        end loop;
        return Scaled_Vector;
    end Scale;
```

```
executed in parallel.
```

```
This may be faster or slower than a sequential execution.
```
Data Parallelism

Vector Machines

Vectorization

\[ \mathbf{a} \cdot \mathbf{v} = a \cdot (x, y, z) = (a \cdot x, a \cdot y, a \cdot z) \]

\[
\text{const} \ \text{Index} = \{1 \ldots 100000000\},
\text{Vector}_1 : [\text{Index}] \ \text{real} = 1.0,
\text{Scale} : \text{real} = 5.1,
\text{Scaled} : [\text{Vector}] \ \text{real} = \text{Scale} \times \text{Vector}_1;
\]

Functions are "promoted"

Executed lazy sequentially.

Translates into CPU-level vector operations as well as multi-core or fully distributed operations.

Data Parallelism

Vector Machines

Reduction

\[
\mathbf{v}_1 = \mathbf{v}_2 \Rightarrow \left( \begin{array}{c} x_1 \\ y_1 \\ z_1 \end{array} \right) \Rightarrow (x_1 = x_2) \land (y_1 = y_2) \land (z_1 = z_2)
\]

\[
\text{type} \ \text{Real Precision} = \text{Float}
\text{type} \ \text{Vector} = [\text{Real Precision}]
\text{equal} :: \text{Vector} \rightarrow \text{Vector} \rightarrow \text{Bool}
\text{equal \ v}_1 \ \text{v}_2 = \text{foldr} (\&\&) \text{True} \ \text{zipWith} (\==) \ \text{v}_1 \ \text{v}_2
\]

Data Parallelism

Vector Machines

Reduction

\[
\mathbf{v}_1 = \mathbf{v}_2 \Rightarrow \left( \begin{array}{c} x_1 \\ y_1 \\ z_1 \end{array} \right) \Rightarrow (x_1 = x_2) \land (y_1 = y_2) \land (z_1 = z_2)
\]

\[
\text{type} \ \text{Real Precision} = \text{Float}
\text{type} \ \text{Vector} = [\text{Real Precision}]
\text{equal} :: \text{Vector} \rightarrow \text{Vector} \rightarrow \text{Bool}
\text{equal} = (\==)
\]

Potentially concurrent, yet:

Executed lazy sequentially.
Vector Machines

Reduction

\[ v_1 = v_2 \Rightarrow \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \Rightarrow (x_1 = x_2) \land (y_1 = y_2) \land (z_1 = z_2) \]

**Type**

- **Real** is digits 15;
- **Vectors** is array (Positive range <>) of Real;

**Function**

- function `=` (Vector_1, Vector_2 : Vectors) return Boolean is
  - (for all i in Vector_1’Range => Vector_1 (i) = Vector_2 (i));

**Translates into**

CPU-level vector operations

**∧-chain is evaluated lazy sequentially.**
Vector Machines

Reduction

\[
\mathbf{v}_1 = \mathbf{v}_2 \Rightarrow \begin{bmatrix}
\mathbf{x}_1 \\
\mathbf{y}_1 \\
\mathbf{z}_1
\end{bmatrix} = \begin{bmatrix}
\mathbf{x}_2 \\
\mathbf{y}_2 \\
\mathbf{z}_2
\end{bmatrix} \Rightarrow (\mathbf{x}_1 = \mathbf{x}_2) \land (\mathbf{y}_1 = \mathbf{y}_2) \land (\mathbf{z}_1 = \mathbf{z}_2)
\]

Translates into

CPU-level vector operations

\land \text{-chain is evaluated lazy sequentially.}

const Index = \{1 .. 100000000\};
proc Equal (v1, v2 : Index) real = 1.0;
proc Equal (v1, v2 : Vectors) return Boolean renames "\=";

Function is “promoted”

Data Parallelism

Vector Machines

Reduction

\[
\mathbf{v}_1 = \mathbf{v}_2 \Rightarrow \begin{bmatrix}
\mathbf{x}_1 \\
\mathbf{y}_1 \\
\mathbf{z}_1
\end{bmatrix} = \begin{bmatrix}
\mathbf{x}_2 \\
\mathbf{y}_2 \\
\mathbf{z}_2
\end{bmatrix} \Rightarrow (\mathbf{x}_1 = \mathbf{x}_2) \land (\mathbf{y}_1 = \mathbf{y}_2) \land (\mathbf{z}_1 = \mathbf{z}_2)
\]

Translates into

CPU-level vector operations

\land \text{-chain is evaluated lazy sequentially.}

\land \text{-operations are evaluated in a concurrent divide-and-conquer (binary tree) structure.}

\land \text{-operations are evaluated in a concurrent divide-and-conquer (binary tree) structure.}
Data Parallelism

Vector Machines

Reduction

\[ \vec v_1 = \vec v_2 \Rightarrow \begin{align*}
    x_1 &= x_2 \\
    y_1 &= y_2 \\
    z_1 &= z_2
\end{align*} \]

\[ \Rightarrow (x_1 = x_2) \land (y_1 = y_2) \land (z_1 = z_2) \]

```
const Index = {1 .. 10000000},
    Vector_1, Vector_2 : [Index] real = 1.0;
proc Equal (v1, v2) : bool
    return v1 == v2;
writeln (Equal (Vector_1, Vector_2));
```

Type mismatch

```
const Mask : [1 .. 3, 1 .. 3] real = ((0, -1, 0), (-1, 5, -1), (0, -1, 0));
```
Vector Machines

General Data-parallelism

\[
\forall px \rightarrow [-1, 5, -1] \text{ real } = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}
\]

\[
\text{const } Mask : \{1 \ldots 3, 1 \ldots 3\} \text{ real } = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix};
\]

\[
\text{proc Unsharp Mask (P, (i, j) : index (Image)) : real }
\]

\[
\{\text{return } \oplus \text{reduce} (Mask \times P[i - 1 .. i + 1, j - 1 .. j + 1]);\}
\]

\[
\text{const Sharpened Picture } = \forall px \text{ in Image do Unsharp Mask (Picture, px)};
\]

Translates into CPU-level vector operations as well as multi-core or fully distributed operations
Cellular automaton transitions from a state $S$ into the next state $S'$: $S \rightarrow S' \iff \forall c \in S: c \rightarrow c' = r(S, c)$, i.e. all cells of a state transition concurrently into new cells by following a rule $r$.

John Conway's Game of Life rule:

```plaintext
proc Rule (S, (i, j) : index (World)) : Cell {
    const Population : index ({0 .. 9}) = + reduce Count (Cell.Alive, S [i - 1 .. i + 1, j - 1 .. j + 1]);
    return (if Population == 3
       || (Population == 4 & & S [i, j] == Cell.Alive)
       then Cell.Alive else Cell.Dead);
}
```

Summary

Data Parallelism

- Data-Parallelism
  - Vectorization
  - Reduction
  - General data-parallelism

Examples

- Image processing
- Cellular automata