Matrix Multiplication Beyond Auto-Tuning: Rewrite Based GPU Code Generation

Michel Steuwer

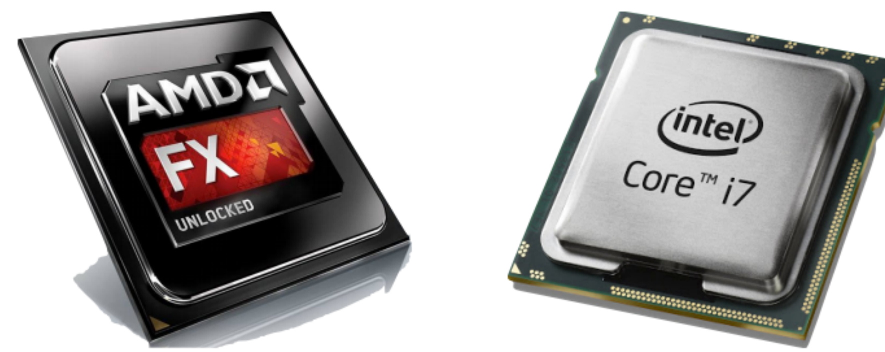
Toomas Remmelg

Christophe Dubach

THE UNIVERSITY *of* EDINBURGH

Motivation

- Tuning parallel programs is complex
- The diversity of heterogeneous accelerators makes it even harder
- Auto-tuning has been successfully applied to ease this
- **Problem of Performance-Portability:**
    - Parametric kernels are complex
    - Still falls short on new architectures!

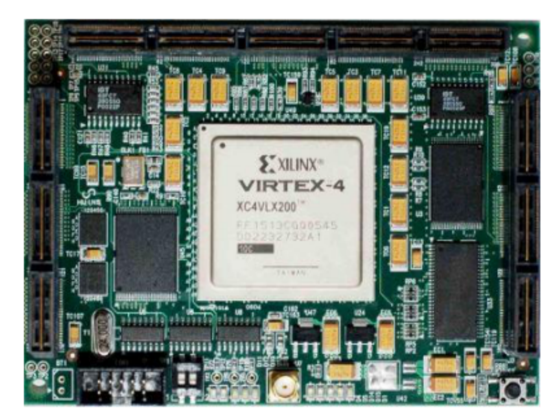


CPU

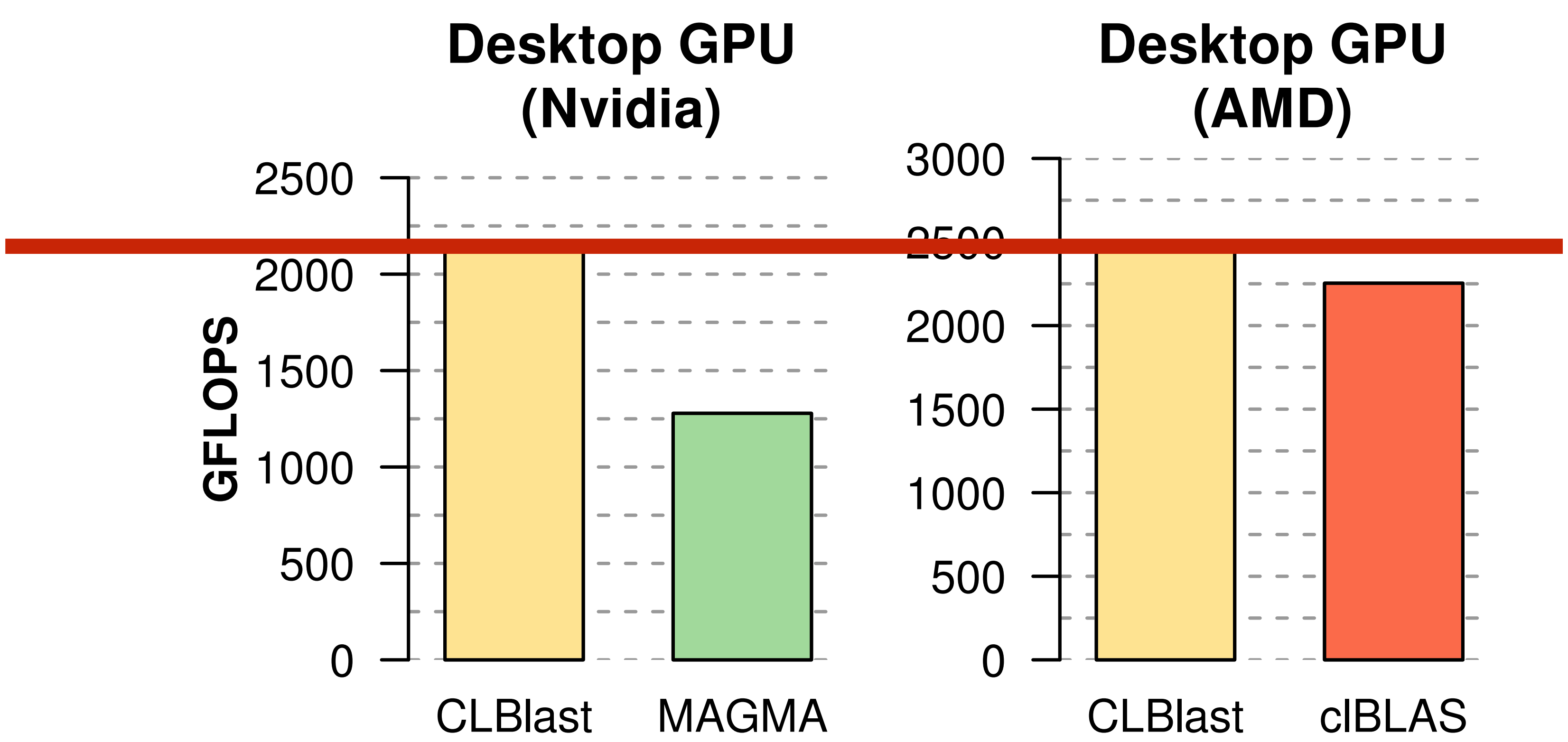
GPU

Accelerator

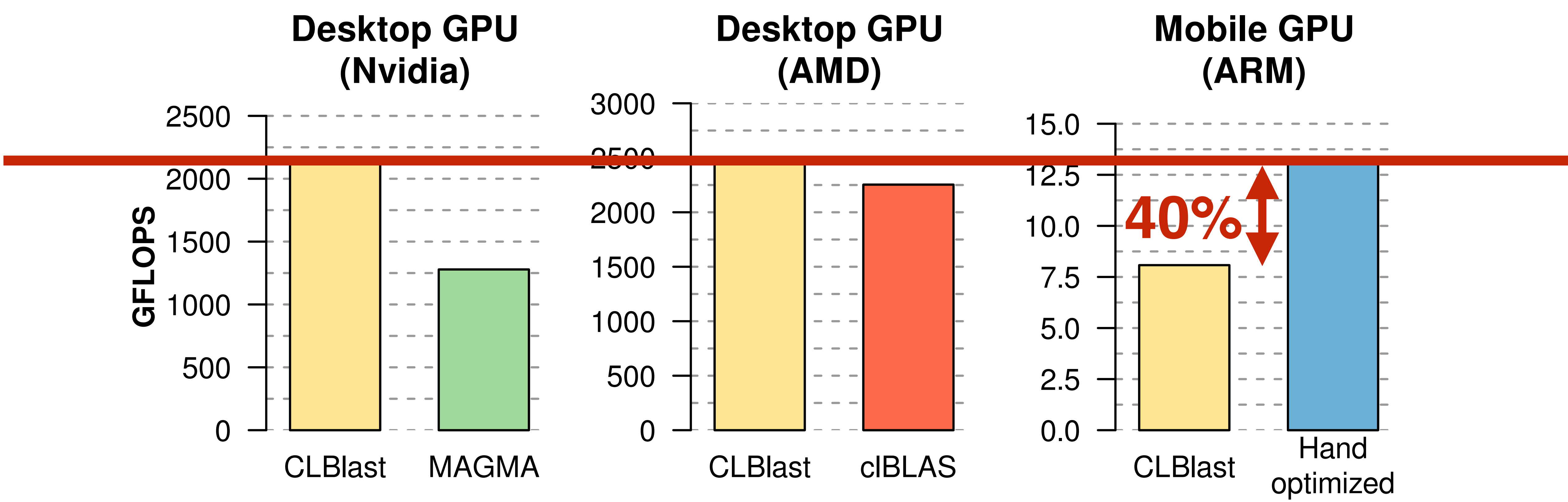
FPGA



The Problem of Performance-Portability
Matrix-Multiplication

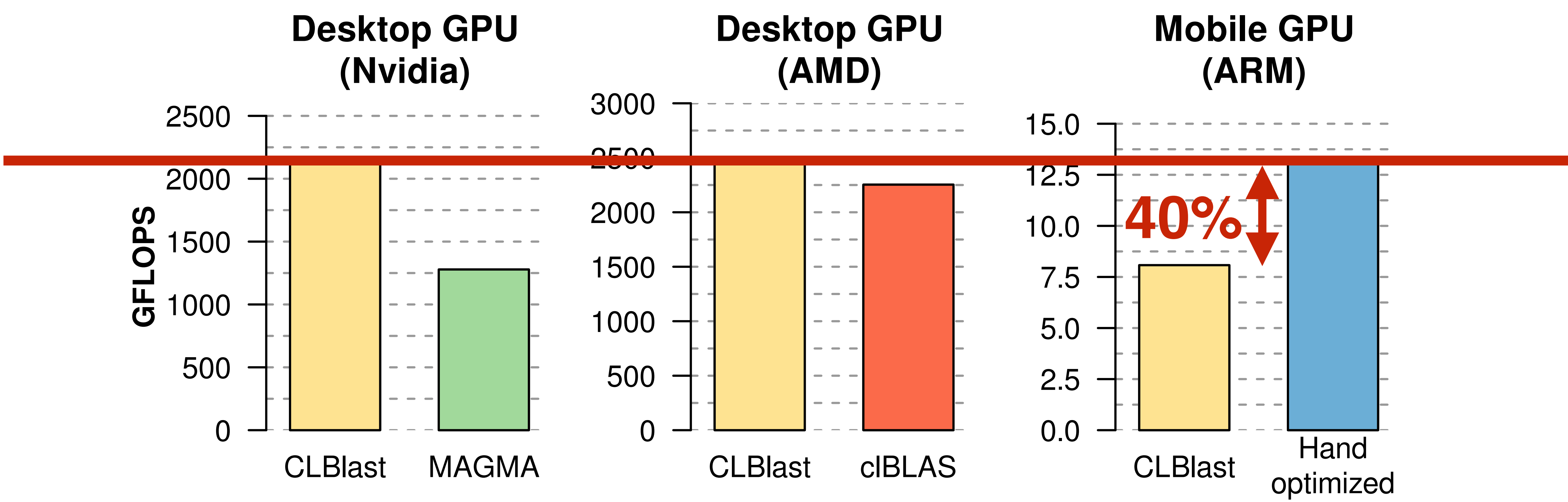


The Problem of Performance-Portability Matrix-Multiplication



- Auto-tuning fails to deliver on a significantly different architecture

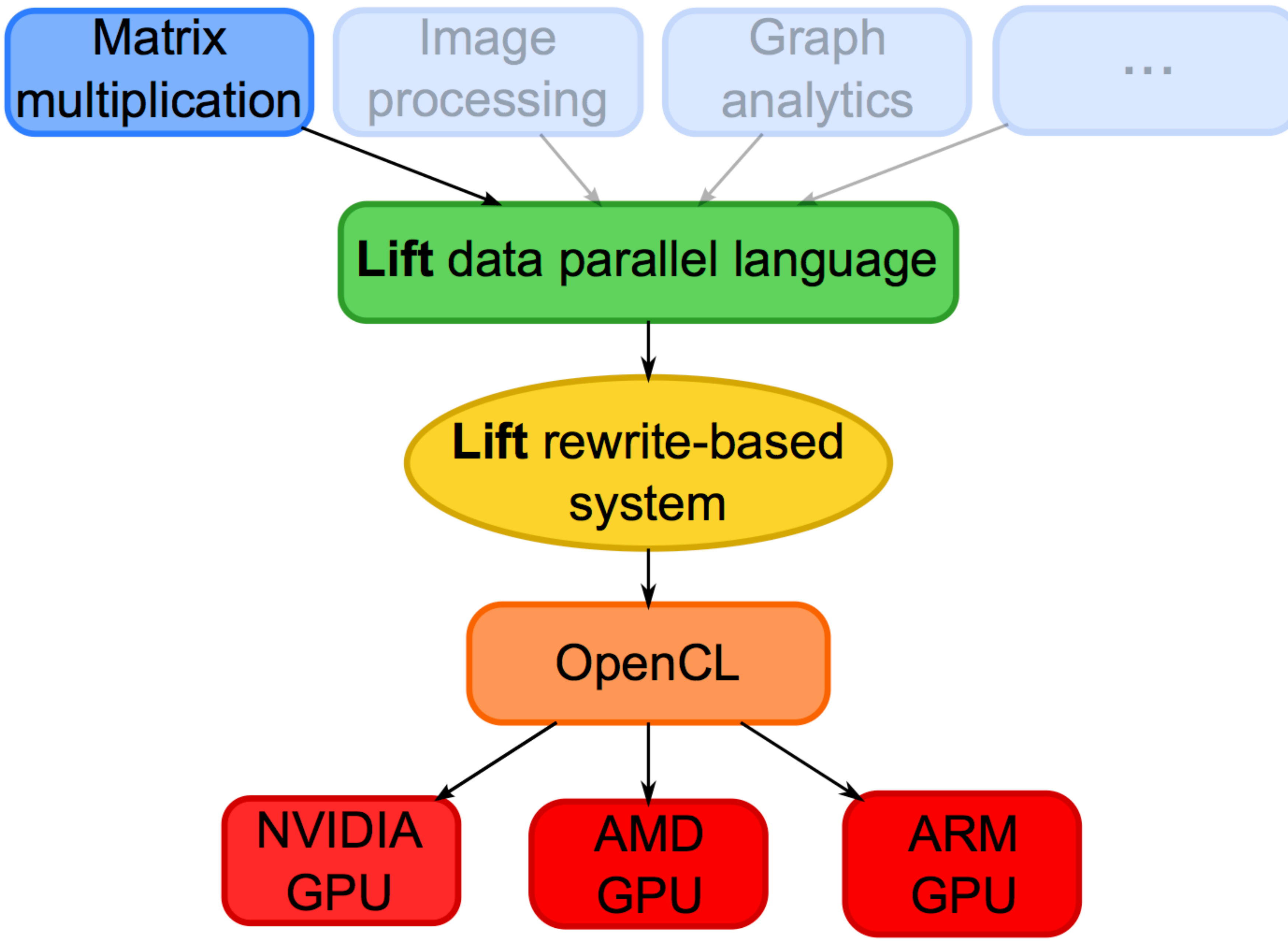
The Problem of Performance-Portability
Matrix-Multiplication



- Auto-tuning fails to deliver on a significantly different architecture
  - **Limitation of Auto-tuning:**
    Fixed set of parameters and optimisations

THE UNIVERSITY *of* EDINBURGH
informatics

Our Approach to Performance Portability



THE UNIVERSITY of EDINBURGH informatics

Lift Data Parallel Language

$$\mathbf{map}(f, \boxed{x_1 \mid x_2 \mid \cdots \mid x_n}) = \boxed{f(x_1) \mid f(x_2) \mid \cdots \mid f(x_n)}$$

$$\mathbf{reduce}(z, f, \boxed{x_1 \mid x_2 \mid \cdots \mid x_n}) = \boxed{f(\cdots(f(f(z, x_1), x_2)\cdots), x_n)}$$

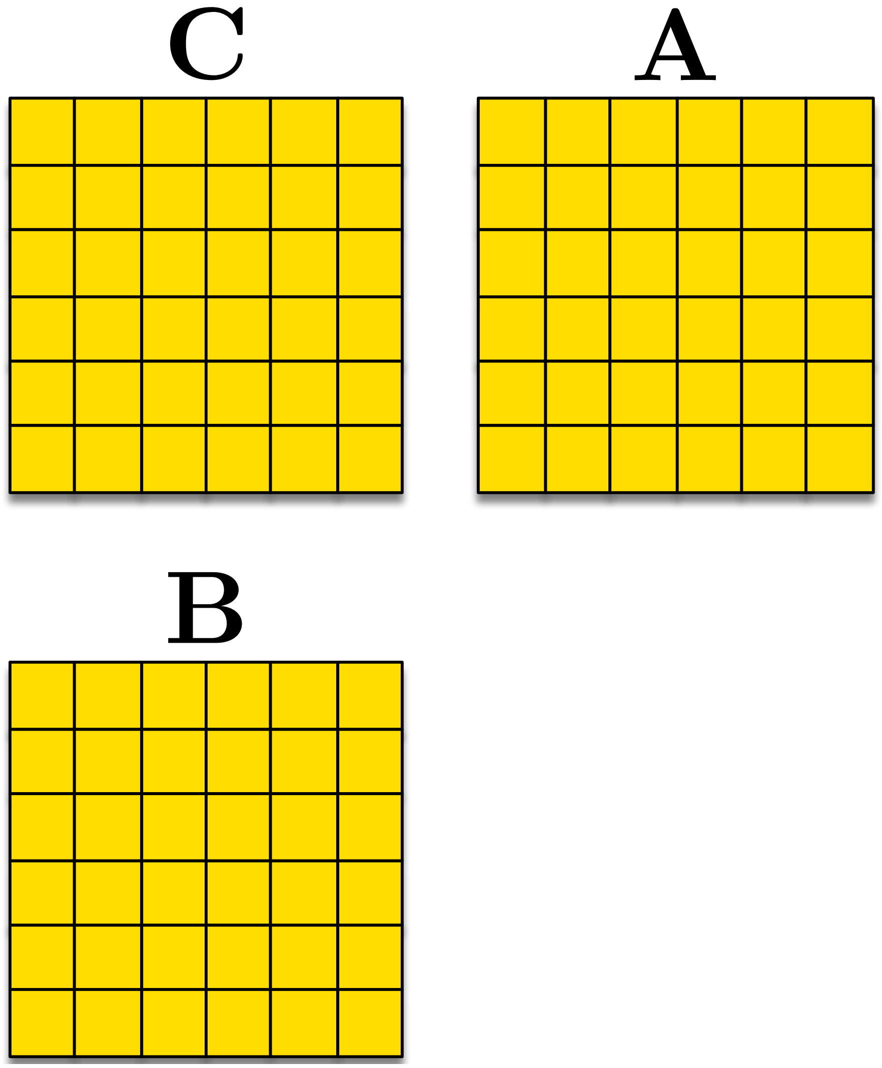
$$\mathbf{zip}(\boxed{x_1 \mid x_2 \mid \cdots \mid x_n}, \boxed{y_1 \mid y_2 \mid \cdots \mid y_n}) = \boxed{(x_1, y_1) \mid (x_2, y_2) \mid \cdots \mid (x_n, y_n)}$$

$$\mathbf{id}(\boxed{x_1 \mid x_2 \mid \cdots \mid x_n}) = \boxed{x_1 \mid x_2 \mid \cdots \mid x_n}$$

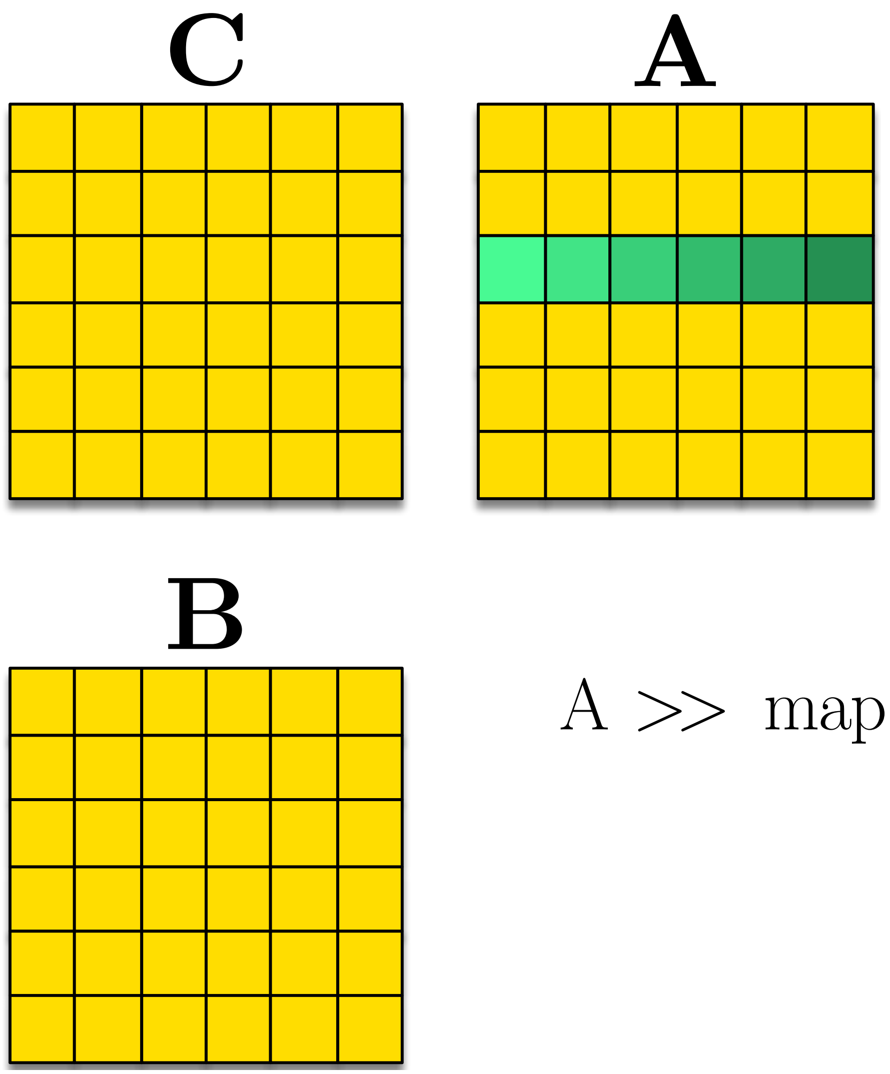
$$\mathbf{split}^m(\boxed{x_1 \mid x_2 \mid \cdots \mid \cdots \mid \cdots \mid \cdots \mid \cdots \mid \cdots \mid \cdots \mid x_n}) = \underbrace{\boxed{\boxed{x_1 \mid x_2 \mid \cdots \mid \cdots}}}_{m}\boxed{\boxed{\cdots \mid \cdots \mid \cdots \mid \cdots}} \mid \cdots \mid \boxed{\boxed{\cdots \mid \cdots \mid \cdots \mid x_n}}$$

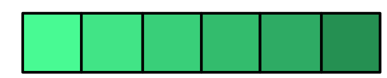
$$\mathbf{join}(\boxed{\boxed{x_1 \mid x_2 \mid \cdots \mid \cdots}}\boxed{\boxed{\cdots \mid \cdots \mid \cdots \mid \cdots}} \mid \cdots \mid \boxed{\boxed{\cdots \mid \cdots \mid \cdots \mid x_n}}) = \boxed{x_1 \mid x_2 \mid \cdots \mid \cdots \mid \cdots \mid \cdots \mid \cdots \mid \cdots \mid \cdots \mid x_n}$$

Matrix Multiplication **Expressed Functionally**



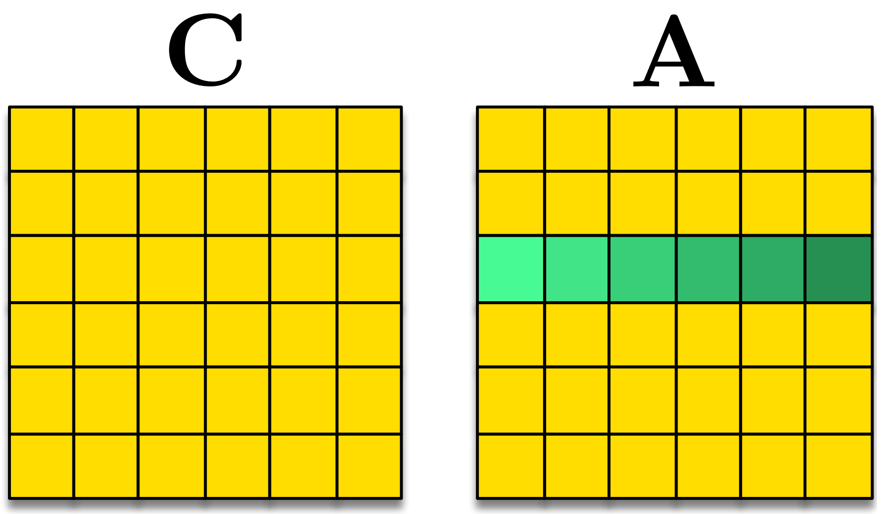
Matrix Multiplication
Expressed Functionally

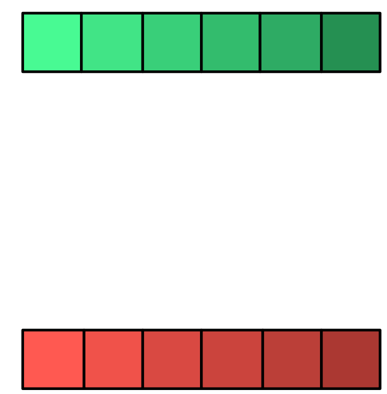


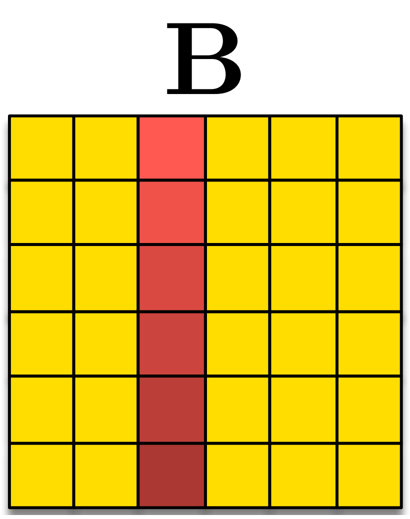


$$A \gg \mathrm{map}(\lambda\ \mathit{rowOfA} \mapsto \qquad)$$

Matrix Multiplication
Expressed Functionally



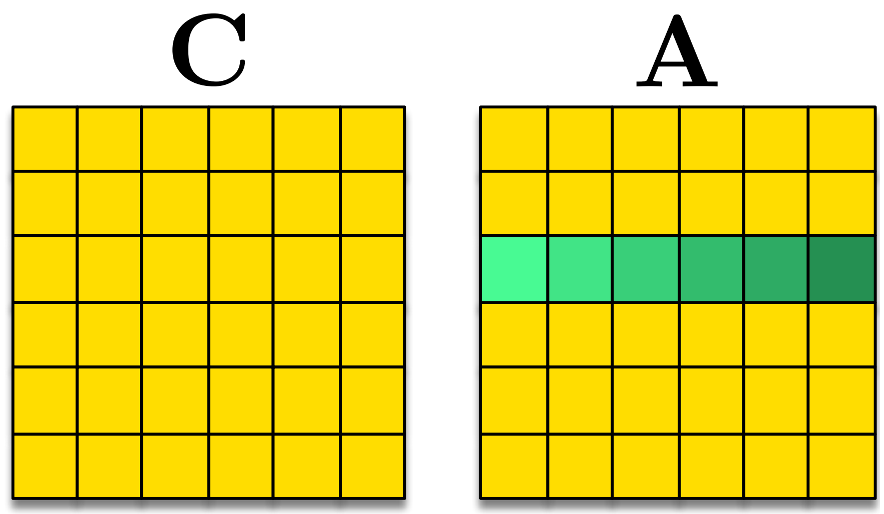


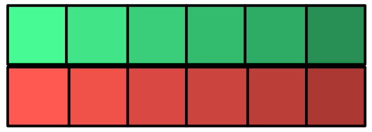


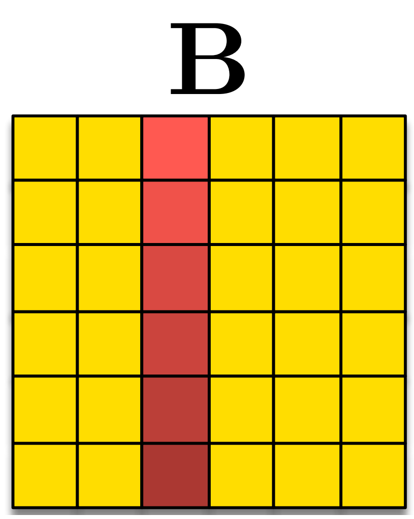
$$A \gg \text{map}(\lambda\ \mathit{rowOfA} \mapsto$$
$$B \gg \text{map}(\lambda\ \mathit{colOfB} \mapsto$$
$$)\ \ )$$

THE UNIVERSITY of EDINBURGH informatics

Matrix Multiplication
Expressed Functionally

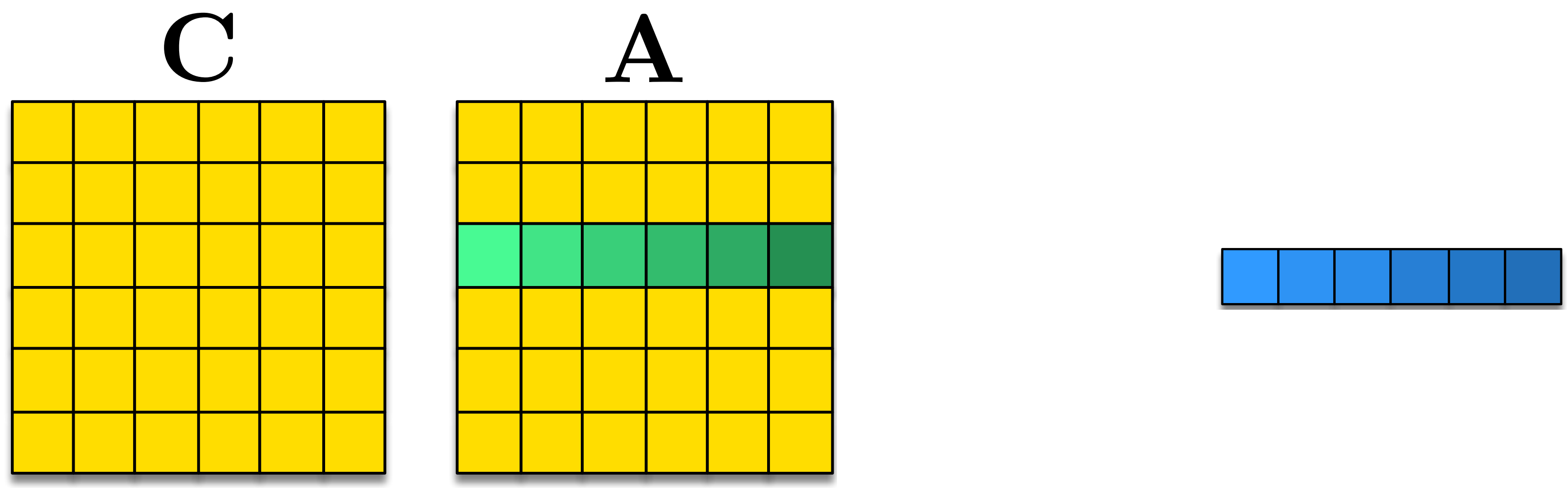


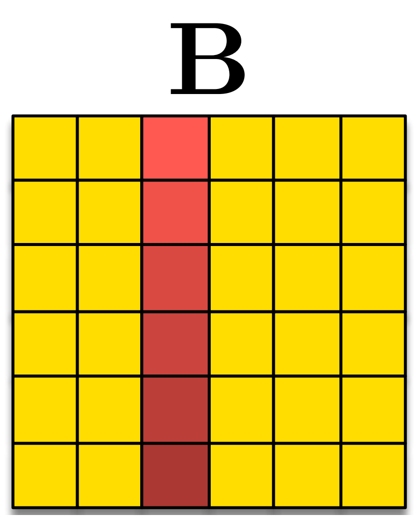




$$A \gg \mathrm{map}(\lambda\ rowOfA \mapsto$$
$$B \gg \mathrm{map}(\lambda\ colOfB \mapsto$$
$$\mathrm{zip}(rowOfA,\ colOfB) \gg$$
$$)\ )$$

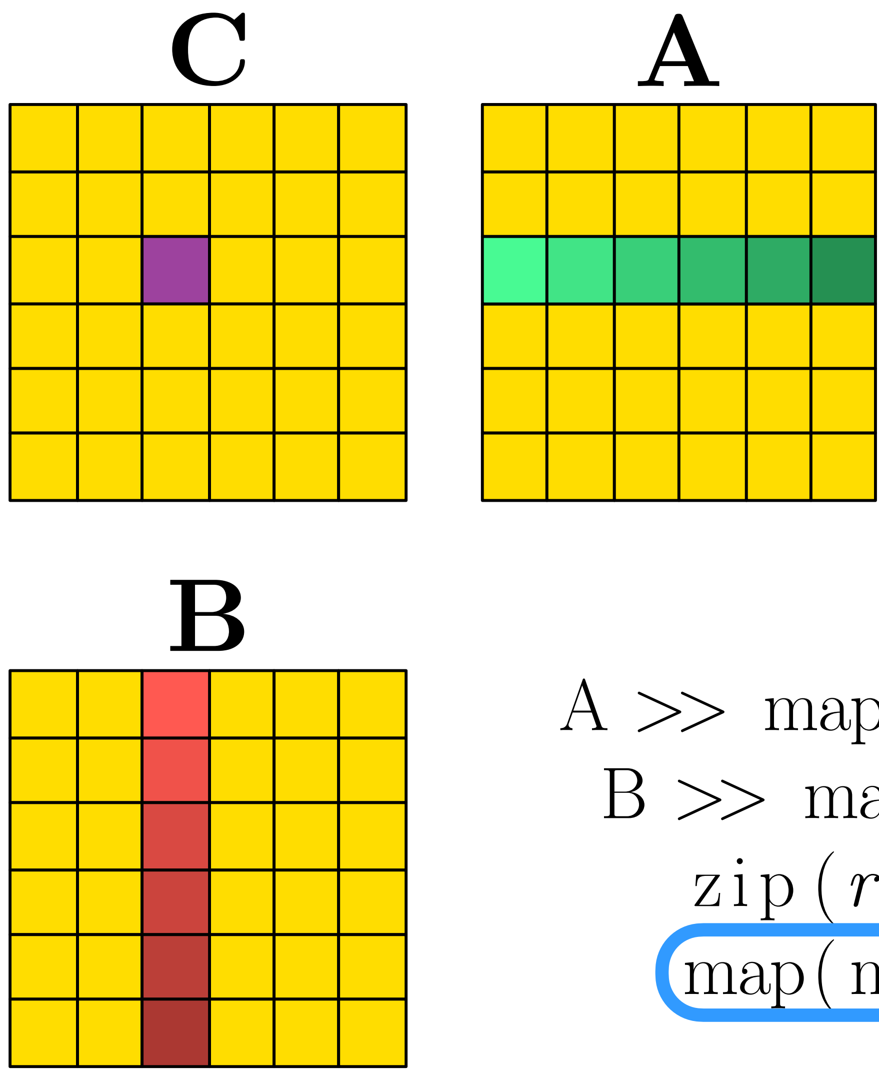
Matrix Multiplication
Expressed Functionally

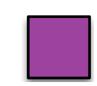




$$
\begin{array}{l}
\mathrm{A} \gg \mathrm{map}(\lambda\ \mathit{rowOfA} \mapsto \\
\quad \mathrm{B} \gg \mathrm{map}(\lambda\ \mathit{colOfB} \mapsto \\
\qquad \mathrm{zip}(\mathit{rowOfA},\ \mathit{colOfB}) \gg \\
\qquad \mathrm{map}(\mathrm{mult}) \qquad\qquad) \ )
\end{array}
$$

Matrix Multiplication
Expressed Functionally





```
A >> map(λ rowOfA ↦
  B >> map(λ colOfB ↦
    zip(rowOfA, colOfB) >>
    map(mult) >> reduce(0.0f, add) ) )
```

```
A >> map(λ rowOfA ↦
  B >> map(λ colOfB ↦
    zip(rowOfA, colOfB) >>
    map(mult) >> reduce(0.0f,add)
  )
)
```

```
A >> map(λ rowOfA ↦
  B >> map(λ colOfB ↦
      zip(rowOfA, colOfB) >>
      map(mult) >> reduce(0.0f, add)
  )
)
```

Naive code generation

```
for (int i = 0; i<M; i++) {
  for (int j = 0; j<N; j++) {
    for (int k = 0; k<K; k++) {
      temp[k + K*N i + K*j] =
        mult(A[k + K*i], B[k + K*j]);
    }
    for (int k = 0;k<K;k++) {
      C[j + N*i] +=
        temp[k + K*N*i + K*j];
    }
  }
}
```

THE UNIVERSITY of EDINBURGH
informatics

```
A >> map(λ rowOfA ↦
  B >> map(λ colOfB ↦
      zip(rowOfA, colOfB) >>
      map(mult) >> reduce(0.0f,add)
  )
)
```

Naive code generation

```
for (int i = 0; i<M; i++) {
  for (int j = 0; j<N; j++) {
    for (int k = 0; k<K; k++) {
      temp[k + K*N i + K*j] =
        mult(A[k + K*i], B[k + K*j]);
    }
    for (int k = 0;k<K;k++) {
      C[j + N*i] +=
        temp[k + K*N*i + K*j];
    }
  }
}
```

THE UNIVERSITY of EDINBURGH
informatics

```
A >> map(λ rowOfA ↦
  B >> map(λ colOfB ↦
      zip(rowOfA, colOfB) >>
      map(mult) >> reduce(0.0f,add)
  )
)
```

Naive code generation

```
for (int i = 0; i<M; i++) {
  for (int j = 0; j<N; j++) {
    for (int k = 0; k<K; k++) {
      temp[k + K*N i + K*j] =
        mult(A[k + K*i], B[k + K*j]);
    }
    for (int k = 0;k<K;k++) {
      C[j + N*i] +=
        temp[k + K*N*i + K*j];
    }
  }
}
```

THE UNIVERSITY of EDINBURGH
informatics

```
A >> map(λ rowOfA ↦
  B >> map(λ colOfB ↦
      zip(rowOfA, colOfB) >>
     map(mult) >> reduce(0.0f, add)
  )
)
```

Naive code generation

```
for (int i = 0; i<M; i++){
  for (int j = 0; j<N; j++){
    for (int k = 0; k<K; k++){
      temp[k + K*N i + K*j] =
        mult(A[k + K*i], B[k + K*j]);
    }
    for (int k = 0;k<K;k++) {
      C[j + N*i] +=
        temp[k + K*N*i + K*j];
    }
  }
}
```

THE UNIVERSITY of EDINBURGH
informatics

```
A >> map(λ rowOfA ↦
  B >> map(λ colOfB ↦
     zip(rowOfA, colOfB) >>
    map(mult) >> reduce(0.0f, add)
  )
)
```

Naive code generation

```
for (int i = 0; i<M; i++){
 for (int j = 0; j<N; j++){
   for (int k = 0; k<K; k++){
     temp[k + K*N i + K*j] =
       mult(A[k + K*i], B[k + K*j]);
   }
   for (int k = 0;k<K;k++) {
     C[j + N*i] +=
       temp[k + K*N*i + K*j];
   }
 }
}
```

How to generate high performance code?

?

Algorithmic Rewrite Rules

Split-join rule:

$$\text{map}(f) \implies \text{split}(k) >> \text{map}(\text{map}(f)) >> \text{join}$$

Map fusion rule:

$$\text{map}(f) >> \text{map}(g) \implies \text{map}(f >> g)$$

Map-reduce fusion rule:

$$\text{mapSeq}(f) >> \text{reduceSeq}(z, \oplus) \implies \text{reduceSeq}(z, \lambda\ (acc, x) \mapsto \oplus(acc, f(x)))$$

Algorithmic Rewrite Rules

Split-join rule:

$$\mathrm{map}(f) \implies \mathtt{split}(k) >> \mathrm{map}(\mathrm{map}(f)) >> \mathtt{join}$$

Map fusion rule:

$$\mathrm{map}(f) >> \mathrm{map}(g) \implies \mathrm{map}(f >> g)$$

Map-reduce fusion rule:

$$\mathrm{mapSeq}(f) >> \mathtt{reduceSeq}(z, \oplus) \implies \mathtt{reduceSeq}(z, \lambda\ (acc, x) \mapsto \oplus(acc, f(x)))$$

- *Rewrite rules* are provably correct
- Express algorithmic and optimisation choices

How to apply rewrite rules?

```
A >> map(λ rowOfA ↦
  B >> map(λ colOfB ↦
     zip(rowOfA, colOfB) >>
     map(mult) >> reduce(0.0f,add)
  )
)
```

```
for (int i  = 0; i<M; i++) {
  for (int j  = 0; j<N; j++) {
    for (int k = 0; k<K; k++) {
      temp[k + K∗N i + K∗j] =
        mult(A[k + K∗i], B[k + K∗j]);
    }
    for (int k = 0;k<K;k++) {
      C[j  + N∗i] +=
        temp[k + K∗N∗i + K∗j];
    }
  }
}
```

```
A >> map(λ rowOfA ↦
  B >> map(λ colOfB ↦
     zip(rowOfA, colOfB) >>
     map(mult) >> reduce(0.0f,add)
  )
)
```

```
for (int i = 0; i<M; i++) {
  for (int j = 0; j<N; j++) {
    for (int k = 0; k<K; k++) {
      temp[k + K*N i + K*j] =
        mult(A[k + K*i], B[k + K*j]);
    }
    for (int k = 0;k<K;k++) {
      C[j + N*i] +=
        temp[k + K*N*i + K*j];
    }
  }
}
```

$\text{map}(f) \implies \text{split}(k) >> \text{map}(\text{map}(f)) >> \text{join}$

```
A >> map(λ rowOfA ↦
  B >> map(λ colOfB ↦
    zip(rowOfA, colOfB) >>
    map(mult) >> reduce(0.0f, add)
  )
)
```

```
for (int i = 0; i<M; i++) {
  for (int j = 0; j<N; j++) {
    for (int k = 0; k<K; k++) {
      temp[k + K*N i + K*j] =
        mult(A[k + K*i], B[k + K*j]);
    }
    for (int k = 0;k<K;k++) {
      C[j + N*i] +=
        temp[k + K*N*i + K*j];
    }
  }
}
```

$\text{map}(f) \implies \text{split}(k) \gg \text{map}(\text{map}(f)) \gg \text{join}$

```
A >> split(m) >> map(λ rowsOfA ↦
  rowsOfA >> map(λ rowOfA ↦
    B >> map(λ colOfB ↦
      zip(rowOfA, colOfB) >>
      map(mult) >> reduce(0.0f, add)
    )
  )
) >> join
```

```
A >> map(λ rowOfA ↦
  B >> map(λ colOfB ↦
    zip(rowOfA, colOfB) >>
    map(mult) >> reduce(0.0f, add)
  )
)
```

```
for (int i = 0; i<M; i++) {
  for (int j = 0; j<N; j++) {
    for (int k = 0; k<K; k++) {
      temp[k + K∗N i + K∗j] =
        mult(A[k + K∗i], B[k + K∗j]);
    }
    for (int k = 0;k<K;k++) {
      C[j + N∗i] +=
        temp[k + K∗N∗i + K∗j];
    }
  }
}
```

$\text{map}(f) \implies \text{split}(k) \gg \text{map}(\text{map}(f)) \gg \text{join}$

```
A >> split(m) >> map(λ rowsOfA ↦
  rowsOfA >> map(λ rowOfA ↦
    B >> map(λ colOfB ↦
      zip(rowOfA, colOfB) >>
      map(mult) >> reduce(0.0f, add)
    )
  )
) >> join
```

```
for (int i = 0; i<M/2; i++) {
  for (int l = 0; l<2; l++) {
    for (int j = 0; j<N; j++) {
      for (int k = 0; k<K; k++) {
        temp[k + 2∗K∗N∗i + K∗N∗l + K∗j] =
          mult(A[k + K∗l + 2∗K∗i], B[k + K∗j]);
      }
      for (int k = 0;k<K;k++) {
        C[j + N∗l + 2∗N∗i] +=
          temp[k + 2∗K∗N∗i + K∗N∗l + K∗j];
      }
    }
  }
}
```

```
A >> split(m) >> map(λ rowsOfA ↦
  rowsOfA >> map(λ rowOfA ↦
    B >> map(λ colOfB ↦
      zip(rowOfA, colOfB) >>
      map(mult) >> reduce(0.0f,add)
    )
  )
) >> join
```

```
for (int i = 0; i<M/2; i++) {
  for (int l = 0; l<2; l++) {
    for (int j = 0; j<N; j++) {
      for (int k = 0; k<K; k++) {
        temp[k + 2*K*N*i + K*N*l + K*j] =
          mult(A[k + K*l + 2*K*i], B[k + K*j]);
      }
      for (int k = 0;k<K;k++) {
        C[j + N*l + 2*N*i] +=
          temp[k + 2*K*N*i + K*N*l + K*j];
      }
    }
  }
}
```

```
A >> split(m) >> map(λ rowsOfA ↦
  rowsOfA >> map(λ rowOfA ↦
    B >> map(λ colOfB ↦
        zip(rowOfA, colOfB) >>
        map(mult) >> reduce(0.0f,add)
    )
  )
) >> join
```

```
for (int i = 0; i<M/2; i++) {
  for (int l = 0; l<2; l++) {
    for (int j = 0; j<N; j++) {
      for (int k = 0; k<K; k++) {
        temp[k + 2*K*N*i + K*N*l + K*j] =
          mult(A[k + K*l + 2*K*i], B[k + K*j]);
      }
      for (int k = 0;k<K;k++) {
        C[j + N*l + 2*N*i] +=
          temp[k + 2*K*N*i + K*N*l + K*j];
      }
    }
  }
}
```

$$X \gg \mathrm{map}(\lambda\ x \mapsto\ Y \gg \mathrm{map}(\lambda\ y \mapsto\ f))$$
$$\Longrightarrow$$
$$Y \gg \mathrm{map}(\lambda\ y \mapsto\ X \gg \mathrm{map}(\lambda\ x \mapsto\ f)) \gg \mathrm{transpose}$$

```
A >> split(m) >> map(λ rowsOfA ↦
  rowsOfA >> map(λ rowOfA ↦
    B >> map(λ colOfB ↦
      zip(rowOfA, colOfB) >>
      map(mult) >> reduce(0.0f,add)
    )
  )
) >> join
```

```
X >> map(λ x ↦ Y >> map(λ y ↦ f))
⟹
Y >> map(λ y ↦ X >> map(λ x ↦ f)) >> transpose
```

```
A >> split(m) >> map(λ rowsOfA ↦
  B >> map(λ colOfB ↦
    rowsOfA >> map(λ rowOfA ↦
      zip(rowOfA, colOfB) >>
      map(mult) >> reduce(0.0f,add)
    )
  ) >> transpose
) >> join
```

```
for (int i = 0; i<M/2; i++) {
  for (int l = 0; l<2; l++) {
    for (int j = 0; j<N; j++) {
      for (int k = 0; k<K; k++) {
        temp[k + 2*K*N*i + K*N*l + K*j] =
          mult(A[k + K*l + 2*K*i], B[k + K*j]);
      }
      for (int k = 0;k<K;k++) {
        C[j + N*l + 2*N*i] +=
          temp[k + 2*K*N*i + K*N*l + K*j];
      }
    }
  }
}
```

```
A >> split(m) >> map(λ rowsOfA ↦
  rowsOfA >> map(λ rowOfA ↦
   B >> map(λ colOfB ↦
        zip(rowOfA, colOfB) >>
        map(mult) >> reduce(0.0f,add)
      )
    )
  ) >> join
```

```
for (int i = 0; i<M/2; i++) {
  for (int l = 0; l<2; l++) {
    for (int j = 0; j<N; j++) {
      for (int k = 0; k<K; k++) {
        temp[k + 2*K*N*i + K*N*l + K*j] =
          mult(A[k + K*l + 2*K*i], B[k + K*j]);
      }
      for (int k = 0;k<K;k++) {
        C[j + N*l + 2*N*i] +=
          temp[k + 2*K*N*i + K*N*l + K*j];
      }
    }
  }
}
```

$$X >> \text{map}(\lambda\ x \mapsto\ Y >> \text{map}(\lambda\ y \mapsto\ f))$$
$$\Longrightarrow$$
$$Y >> \text{map}(\lambda\ y \mapsto\ X >> \text{map}(\lambda\ x \mapsto\ f)) >> \text{transpose}$$

```
A >> split(m) >> map(λ rowsOfA ↦
  B >> map(λ colOfB ↦
   rowsOfA >> map(λ rowOfA ↦
        zip(rowOfA, colOfB) >>
        map(mult) >> reduce(0.0f,add)
      )
    ) >> transpose
  ) >> join
```

```
for (int i = 0; i<M/2; i++) {
  for (int j = 0; j<N; j++) {
    for (int l = 0; l<2; l++) {
      for (int k = 0; k<K; k++) {
        temp[k + 2*K*N*i + K*N*l + K*j] =
          mult(A[k + K*l + 2*K*i], B[k + K*j]);
      }
      for (int k = 0;k<K;k++) {
        C[j + N*l + 2*N*i] +=
          temp[k + 2*K*N*i + K*N*l + K*j];
      }
    }
  }
}
```

```
A >> split(m) >> map(λ rowsOfA ↦
  B >> map(λ colOfB ↦
    rowsOfA >> map(λ rowOfA ↦
      zip(rowOfA, colOfB) >>
      map(mult) >> reduce(0.0f,add)
    )
  ) >> transpose
) >> join
```

```
for (int i = 0; i<M/2; i++) {
  for (int j = 0; j<N; j++) {
    for (int l = 0; l<2; l++) {
      for (int k = 0; k<K; k++) {
        temp[k + 2*K*N*i + K*N*l + K*j] =
          mult(A[k + K*l + 2*K*i], B[k + K*j]);
      }
      for (int k = 0;k<K;k++) {
        C[j + N*l + 2*N*i] +=
          temp[k + 2*K*N*i + K*N*l + K*j];
      }
    }
  }
}
```

```
A >> split(m) >> map(λ rowsOfA ↦
  B >> map(λ colOfB ↦
    rowsOfA >> map(λ rowOfA ↦
      zip(rowOfA, colOfB) >>
      map(mult) >> reduce(0.0f,add)
    )
  ) >> transpose
) >> join
```

```
for (int i = 0; i<M/2; i++) {
  for (int j = 0; j<N; j++) {
    for (int l = 0; l<2; l++) {
      for (int k = 0; k<K; k++) {
        temp[k + 2*K*N*i + K*N*l + K*j] =
          mult(A[k + K*l + 2*K*i], B[k + K*j]);
      }
      for (int k = 0;k<K;k++) {
        C[j + N*l + 2*N*i] +=
          temp[k + 2*K*N*i + K*N*l + K*j];
      }
    }
  }
}
```

$\text{map}(f) \implies \text{split}(k) \gg \text{map}(\text{map}(f)) \gg \text{join}$

```
A >> split(m) >> map(λ rowsOfA ↦
 B >> map(λ colOfB ↦
   rowsOfA >> map(λ rowOfA ↦
     zip(rowOfA, colOfB) >>
     map(mult) >> reduce(0.0f,add)
   )
  ) >> transpose
 ) >> join
```

```
for (int i = 0; i<M/2; i++) {
  for (int j = 0; j<N; j++) {
    for (int l = 0; l<2; l++) {
      for (int k = 0; k<K; k++) {
        temp[k + 2*K*N*i + K*N*l + K*j] =
          mult(A[k + K*l + 2*K*i], B[k + K*j]);
      }
      for (int k = 0;k<K;k++) {
        C[j + N*l + 2*N*i] +=
          temp[k + 2*K*N*i + K*N*l + K*j];
      }
    }
  }
}
```

```
map(f) ⟹ split(k) >> map(map(f)) >> join
```

```
A >> split(m) >> map(λ rowsOfA ↦
 B >> split(n) >> map(λ colsOfB ↦
  colsOfB >> map(λ colOfB ↦
   rowsOfA >> map(λ rowOfA ↦
     zip(rowOfA, colOfB) >>
     map(mult) >> reduce(0.0f,add)
   )
  )
 ) >> join >> transpose
) >> join
```

```
A >> split(m) >> map(λ rowsOfA ↦
 B >> map(λ colOfB ↦
   rowsOfA >> map(λ rowOfA ↦
     zip(rowOfA, colOfB) >>
     map(mult) >> reduce(0.0f,add)
   )
  ) >> transpose
) >> join
```

```
for (int i = 0; i<M/2; i++) {
 for (int j = 0; j<N; j++) {
  for (int l = 0; l<2; l++) {
   for (int k = 0; k<K; k++) {
    temp[k + 2*K*N*i + K*N*l + K*j] =
     mult(A[k + K*l + 2*K*i], B[k + K*j]);
   }
   for (int k = 0;k<K;k++) {
    C[j + N*l + 2*N*i] +=
     temp[k + 2*K*N*i + K*N*l + K*j];
   }
  }
 }
}
```

```
map(f) ⟹ split(k) >> map(map(f)) >> join
```

```
A >> split(m) >> map(λ rowsOfA ↦
 B >> split(n) >> map(λ colsOfB ↦
  colsOfB >> map(λ colOfB ↦
   rowsOfA >> map(λ rowOfA ↦
     zip(rowOfA, colOfB) >>
     map(mult) >> reduce(0.0f,add)
   )
  )
 ) >> join >> transpose
) >> join
```

```
for (int i = 0; i<M/2; i++) {
 for (int j = 0; j<N/2; j++) {
  for (int m = 0; m<2; m++) {
   for (int l = 0; l<2; l++) {
    for (int k = 0; k<K; k++) {
     temp[k + 4*K*N*i + 2*K*N*l + 2*K*j
        + K*m] =
       mult(A[k + K*l + 2*K*i], B[k + K*
         m + 2*K*j]);
    }
    for (int k = 0;k<K;k++) {
     C[m + 2*j + 2*N*l + 4*N*i] +=
       temp[k + 4*K*N*i + 2*K*N*l + 2*
         K*j + K*m];
    }
   }
  }
 }
}
```

```
A >> split(m) >> map(λ rowsOfA ↦
  B >> split(n) >> map(λ colsOfB ↦
    colsOfB >> map(λ colOfB ↦
      rowsOfA >> map(λ rowOfA ↦
          zip(rowOfA, colOfB) >>
          map(mult) >> reduce(0.0f,add) ) )
  ) >> join >> transpose
) >> join
```

```
for (int i = 0; i<M/2; i++) {
  for (int j = 0; j<N/2; j++) {
    for (int m = 0; m<2; m++) {
      for (int l = 0; l<2; l++) {
        for (int k = 0; k<K; k++) {
          temp[k + 4*K*N*i + 2*K*N*l + 2*K*j + K*m] =
            mult(A[k + K*l + 2*K*i], B[k + K*m + 2*K*j]);
        }
        for (int k = 0;k<K;k++) {
          C[m + 2*j + 2*N*l + 4*N*i] +=
            temp[k + 4*K*N*i + 2*K*N*l + 2*K*j + K*m];
        }
      }
    }
  }
}
```

```
A >> split(m) >> map(λ rowsOfA ↦
  B >> split(n) >> map(λ colsOfB ↦
    colsOfB >> map(λ colOfB ↦
      rowsOfA >> map(λ rowOfA ↦
          zip(rowOfA, colOfB) >>
          map(mult) >> reduce(0.0f,add) ) )
  ) >> join >> transpose
) >> join
```

```
for (int i = 0; i<M/2; i++) {
  for (int j = 0; j<N/2; j++) {
    for (int m = 0; m<2; m++) {
      for (int l = 0; l<2; l++) {
        for (int k = 0; k<K; k++) {
          temp[k + 4*K*N*i + 2*K*N*l + 2*K*j + K*m] =
            mult(A[k + K*l + 2*K*i], B[k + K*m + 2*K*j]);
        }
        for (int k = 0;k<K;k++) {
          C[m + 2*j + 2*N*l + 4*N*i] +=
            temp[k + 4*K*N*i + 2*K*N*l + 2*K*j + K*m];
        }
      }
    }
  }
}
```

$$X \gg \text{map}(\lambda\ x \mapsto\ Y \gg \text{map}(\lambda\ y \mapsto\ f))$$
$$\Longrightarrow$$
$$Y \gg \text{map}(\lambda\ y \mapsto\ X \gg \text{map}(\lambda\ x \mapsto\ f)) \gg \text{transpose}$$

```
A >> split(m) >> map(λ rowsOfA ↦
 B >> split(n) >> map(λ colsOfB ↦
  colsOfB >> map(λ colOfB ↦
   rowsOfA >> map(λ rowOfA ↦
     zip(rowOfA, colOfB) >>
     map(mult) >> reduce(0.0f,add) ) )
 ) >> join >> transpose
) >> join
```

$$X \gg \text{map}(\lambda\ x \mapsto\ Y \gg \text{map}(\lambda\ y \mapsto\ f))$$
$$\Longrightarrow$$
$$Y \gg \text{map}(\lambda\ y \mapsto\ X \gg \text{map}(\lambda\ x \mapsto\ f)) \gg \text{transpose}$$

```
A >> split(m) >> map(λ rowsOfA ↦
 B >> split(n) >> map(λ colsOfB ↦
  rowsOfA >> map(λ rowOfA ↦
   colsOfB >> map(λ colOfB ↦
     zip(rowOfA, colOfB) >>
     map(mult) >> reduce(0.0f,add) )
  ) >> transpose
 ) >> join >> transpose
) >> join
```

```c
for (int i = 0; i<M/2; i++) {
  for (int j = 0; j<N/2; j++) {
    for (int m = 0; m<2; m++) {
      for (int l = 0; l<2; l++) {
        for (int k = 0; k<K; k++) {
          temp[k + 4*K*N*i + 2*K*N*l + 2*K*j
              + K*m] =
            mult(A[k + K*l + 2*K*i], B[k + K*
              m + 2*K*j]);
        }
        for (int k = 0;k<K;k++) {
          C[m + 2*j + 2*N*l + 4*N*i] +=
            temp[k + 4*K*N*i + 2*K*N*l + 2*
              K*j + K*m];
        }
      }
    }
  }
}
```

```
A >> split(m) >> map(λ rowsOfA ↦
 B >> split(n) >> map(λ colsOfB ↦
  colsOfB >> map(λ colOfB ↦
   rowsOfA >> map(λ rowOfA ↦
      zip(rowOfA, colOfB) >>
      map(mult) >> reduce(0.0f,add) ) )
 ) >> join >> transpose
) >> join
```

```
for (int i = 0; i<M/2; i++) {
  for (int j = 0; j<N/2; j++) {
    for (int m = 0; m<2; m++) {
      for (int l = 0; l<2; l++) {
        for (int k = 0; k<K; k++) {
          temp[k + 4*K*N*i + 2*K*N*l + 2*K*j
             + K*m] =
            mult(A[k + K*l + 2*K*i], B[k + K*
              m + 2*K*j]);
        }
        for (int k = 0;k<K;k++) {
          C[m + 2*j + 2*N*l + 4*N*i] +=
            temp[k + 4*K*N*i + 2*K*N*l + 2*
              K*j + K*m];
        }
      }
    }
  }
}
```

$$X \gg \text{map}(\lambda\ x \mapsto\ Y \gg \text{map}(\lambda\ y \mapsto\ f))$$
$$\Longrightarrow$$
$$Y \gg \text{map}(\lambda\ y \mapsto\ X \gg \text{map}(\lambda\ x \mapsto\ f)) \gg \text{transpose}$$

```
A >> split(m) >> map(λ rowsOfA ↦
 B >> split(n) >> map(λ colsOfB ↦
  rowsOfA >> map(λ rowOfA ↦
   colsOfB >> map(λ colOfB ↦
      zip(rowOfA, colOfB) >>
      map(mult) >> reduce(0.0f,add) )
  ) >> transpose
 ) >> join >> transpose
) >> join
```

```
for (int i = 0; i<M/2; i++) {
  for (int j = 0; j<N/2; j++) {
    for (int l = 0; l<2; l++) {
      for (int m = 0; m<2; m++) {
        for (int k = 0; k<K; k++) {
          temp[k + 4*K*N*i + 2*K*N*l + 2*K*j
             + K*m] =
            mult(A[k + K*l + 2*K*i], B[k + K*
              m + 2*K*j]);
        }
        for (int k = 0;k<K;k++) {
          C[m + 2*j + 2*N*l + 4*N*i] +=
            temp[k + 4*K*N*i + 2*K*N*l + 2*
              K*j + K*m];
        }
      }
    }
  }
}
```

After algorithmic rewrites…

Tiled Matrix Multiplication

```
λ (A, B) ↦
 A >> split(m) >> map(λ nRowsOfA ↦
  B >> split(n) >> map(λ mColsOfB ↦
    zip( transpose(nRowsOfA) >> split(k),
         transpose(mColsOfB) >> split(k) ) >>
    reduceSeq(init = make2DArray(n,m, 0.0f),
     λ (accTile, (tileOfA, tileOfB)) ↦
      zip(accTile, transpose(tileOfA)) >>
      map(λ (accRow, rowOfTileOfA) ↦
       zip(accRow, transpose(tileOfB)) >>
       map(λ (acc, colOfTileOfB) ↦
         zip(rowOfTileOfA, colOfTileOfB) >>
         map(mult) >> reduce(acc, add)
       ) >> join
      )
    ) >> transpose() >>
    map(transpose) >> transpose
  ) >> join >> transpose
) >> join
```

```
for (int i = 0;i<M/2; i++) {
  for (int j = 0;j<N/2; j++) {
    for (int k = 0;k<K/4; k++) {
      for (int l = 0;l<2; l++) {
        for (int m = 0;m<2; m++) {
          for (int n = 0;n<4; n++) {
            temp[n + 4*m + 8*N*i + 16*j + 8*l] =
              mult(
                A[n + 2*K*i + 4*k + K*l],
                B[n + 2*K*j + 4*k + K*m]
              );
          }
          for (int n = 0;n<4; n++) {
            C[m + 2*N*i + 2*j + N*l] +=
              temp[n + 4*m + 8*N*i + 16*j + 8*l];
          }
        }
      }
    }
  }
}
```

How to map to OpenCL?

OpenCL Specific Rewrite Rules

- *Rewrite rules* express mapping and optimisation choices
- *Patterns* correspond to OpenCL concepts

Examples:

OpenCL thread hierarchy:

$\mathrm{map}(f) \implies \mathrm{mapGlb}_{\{0,1,2\}}(f)$

$\mathrm{map}(f) \implies \mathrm{mapLcl}_{\{0,1,2\}}(f)$

OpenCL memory hierarchy:

$f \implies \mathrm{toPrivate}(f)$

$f \implies \mathrm{toLocal}(f)$

$f \implies \mathrm{toGlobal}(f)$

OpenCL vector types and operations:

$$\mathrm{map}(f) \implies \mathrm{asVector}(n, b) \gg \mathrm{map}(\mathrm{vectorize}(n, f)) \gg \mathrm{asScalar}$$

```
λ (A, B) ↦
 A >> split(m) >> map(λ nRowsOfA ↦
  B >> split(n) >> map(λ mColsOfB ↦
    zip( transpose(nRowsOfA) >> split(k),
         transpose(mColsOfB) >> split(k) ) >>
    reduceSeq(init = make2DArray(n,m, 0.0f),
     λ (accTile, (tileOfA, tileOfB)) ↦
      zip(accTile, transpose(tileOfA)) >>
      map(λ (accRow, rowOfTileOfA) ↦
       zip(accRow, transpose(tileOfB)) >>
       map(λ (acc, colOfTileOfB) ↦
         zip(rowOfTileOfA, colOfTileOfB) >>
         map(mult) >> reduce(acc, add)
       ) >> join
      )
    ) >> transpose() >>
    map(transpose) >> transpose
  ) >> join >> transpose
) >> join
```

```
for (int i = 0;i<M/2; i++) {
  for (int j = 0;j<N/2; j++) {
    for (int k = 0;k<K/4; k++) {
      for (int l = 0;l<2; l++) {
        for (int m = 0;m<2; m++) {
          for (int n = 0;n<4; n++) {
            temp[n + 4*m + 8*N*i + 16*j + 8*l] =
              mult(
                A[n + 2*K*i + 4*k + K*l],
                B[n + 2*K*j + 4*k + K*m]
              );
          }
          for (int n = 0;n<4; n++) {
            C[m + 2*N*i + 2*j + N*l] +=
              temp[n + 4*m + 8*N*i + 16*j + 8*l];
          }
        }
      }
    }
  }
}
```

```
λ (A, B) ↦
 A >> split(m) >> map(λ nRowsOfA ↦
  B >> split(n) >> map(λ mColsOfB ↦
    zip( transpose(nRowsOfA) >> split(k),
         transpose(mColsOfB) >> split(k) ) >>
    reduceSeq(init = make2DArray(n,m, 0.0f),
     λ (accTile, (tileOfA, tileOfB)) ↦
      zip(accTile, transpose(tileOfA)) >>
      map(λ (accRow, rowOfTileOfA) ↦
       zip(accRow, transpose(tileOfB)) >>
       map(λ (acc, colOfTileOfB) ↦
         zip(rowOfTileOfA, colOfTileOfB) >>
         map(mult) >> reduce(acc, add)
       ) >> join
      )
    ) >> transpose() >>
    map(transpose) >> transpose
  ) >> join >> transpose
) >> join
```

```
for (int i = 0;i<M/2; i++) {
  for (int j = 0;j<N/2; j++) {
    for (int k = 0;k<K/4; k++) {
      for (int l = 0;l<2; l++) {
        for (int m = 0;m<2; m++) {
          for (int n = 0;n<4; n++) {
            temp[n + 4*m + 8*N*i + 16*j + 8*l] =
              mult(
                A[n + 2*K*i + 4*k + K*l],
                B[n + 2*K*j + 4*k + K*m]
              );
          }
          for (int n = 0;n<4; n++) {
            C[m + 2*N*i + 2*j + N*l] +=
              temp[n + 4*m + 8*N*i + 16*j + 8*l];
          }
        }
      }
    }
  }
}
```

```
zip(a, b) >> map(f)
⟹
zip(asVector(n, a), asVector(n, b))
  >> map(vectorize(n, f)) >> asScalar
```

```
λ (A, B) ↦
 A >> split(m) >> map(λ nRowsOfA ↦
  B >> split(n) >> map(λ mColsOfB ↦
    zip( transpose(nRowsOfA) >> split(k),
         transpose(mColsOfB) >> split(k) ) >>
    reduceSeq(init = make2DArray(n,m, 0.0f),
     λ (accTile, (tileOfA, tileOfB)) ↦
      zip(accTile, transpose(tileOfA)) >>
      map(λ (accRow, rowOfTileOfA) ↦
       zip(accRow, transpose(tileOfB)) >>
       map(λ (acc, colOfTileOfB) ↦
        zip(rowOfTileOfA >> asVector(k),
            colOfTileOfB >> asVector(k)) >>
         map(mult4) >> asScalar >>
          reduce(acc, add)
        ) >> join
      )
    ) >> transpose() >>
    map(transpose) >> transpose
  ) >> join >> transpose
) >> join
```

```
for (int i = 0;i<M/2; i++) {
  for (int j = 0;j<N/2; j++) {
    for (int k = 0;k<K/4; k++) {
      for (int l = 0;l<2; l++) {
        for (int m = 0;m<2; m++) {
          float4 t = mult4(
              vload4(A, K*i/2 + k + K*l/4),
              vload4(B, K*j/2 + k + K*m/4)
            );
          vstore4(t, temp, m + 2*N*i + 4*j + 2*l);
          for (int n = 0;n<4; n++) {
            C[m + 2*N*i + 2*j + N*l] +=
              temp[n + 4*m + 8*N*i + 16*j + 8*l];
          }
        }
      }
    }
  }
}
```

```
λ (A, B) ↦
 A >> split(m) >> map(λ nRowsOfA ↦
  B >> split(n) >> map(λ mColsOfB ↦
    zip( transpose(nRowsOfA) >> split(k),
         transpose(mColsOfB) >> split(k) ) >>
    reduceSeq(init = make2DArray(n,m, 0.0f),
     λ (accTile, (tileOfA, tileOfB)) ↦
      zip(accTile, transpose(tileOfA)) >>
      map(λ (accRow, rowOfTileOfA) ↦
       zip(accRow, transpose(tileOfB)) >>
       map(λ (acc, colOfTileOfB) ↦
       zip(rowOfTileOfA >> asVector(k),
           colOfTileOfB >> asVector(k)) >>
         map(mult4) >> asScalar >>
         reduce(acc, add)
       ) >> join
      )
    ) >> transpose() >>
    map(transpose) >> transpose
  ) >> join >> transpose
) >> join
```

```
for (int i = 0;i<M/2; i++) {
  for (int j = 0;j<N/2; j++) {
    for (int k = 0;k<K/4; k++) {
      for (int l = 0;l<2; l++) {
        for (int m = 0;m<2; m++) {
          float4 t = mult4(
              vload4(A, K*i/2 + k + K*l/4),
              vload4(B, K*j/2 + k + K*m/4)
            );
          vstore4(t, temp, m + 2*N*i + 4*j + 2*l);
          for (int n = 0;n<4; n++) {
            C[m + 2*N*i + 2*j + N*l] +=
              temp[n + 4*m + 8*N*i + 16*j + 8*l];
          }
        }
      }
    }
  }
}
```

$\text{map}(f) \implies \text{mapSeq}(f)$

$\text{map}(f) \implies \text{mapGlb}_{\{0,1,2\}}(f)$

$\text{reduce}(z, \oplus) \implies \text{reduceSeq}(z, \oplus)$

```
λ (A, B) ↦
 A >> split(m) >> mapGlb0(λ nRowsOfA ↦
  B >> split(n) >> mapGlb1(λ mColsOfB ↦
    zip( transpose(nRowsOfA) >> split(k),
         transpose(mColsOfB) >> split(k) ) >>
    reduceSeq(init = make2DArray(n,m, 0.0f),
     λ (accTile, (tileOfA, tileOfB)) ↦
      zip(accTile, transpose(tileOfA)) >>
      mapSeq(λ (accRow, rowOfTileOfA) ↦
       zip(accRow, transpose(tileOfB)) >>
       mapSeq(λ (acc, colOfTileOfB) ↦
       zip(rowOfTileOfA >> asVector(k),
           colOfTileOfB >> asVector(k)) >>
         mapSeq(mult4) >> asScalar >>
         reduceSeq(acc, add)
       ) >> join
      )
    ) >> transpose() >>
    map(transpose) >> transpose
  ) >> join >> transpose
) >> join
```

```
int i = get_global_id(0);
int j = get_global_id(1);
for (int k = 0;k<K/4; k++) {
  for (int l = 0;l<2; l++) {
    for (int m = 0;m<2; m++) {
      float4 t = mult4(
          vload4(A, K*i/2 + k + K*l/4),
          vload4(B, K*j/2 + k + K*m/4)
        );
      vstore4(t, temp, m + 2*N*i + 4*j + 2*l);
      for (int n = 0;n<4; n++) {
        C[m + 2*N*i + 2*j + N*l] +=
          temp[n + 4*m + 8*N*i + 16*j + 8*l];
      }
    }
  }
}
```

```
λ (A, B) ↦
 A >> split(m) >> mapGlb0(λ nRowsOfA ↦
  B >> split(n) >> mapGlb1(λ mColsOfB ↦
    zip( transpose(nRowsOfA) >> split(k),
         transpose(mColsOfB) >> split(k) ) >>
    reduceSeq(init = make2DArray(n,m, 0.0f),
     λ (accTile, (tileOfA, tileOfB)) ↦
      zip(accTile, transpose(tileOfA)) >>
      mapSeq(λ (accRow, rowOfTileOfA) ↦
       zip(accRow, transpose(tileOfB)) >>
       mapSeq(λ (acc, colOfTileOfB) ↦
       zip(rowOfTileOfA >> asVector(k),
           colOfTileOfB >> asVector(k)) >>
         mapSeq(mult4) >> asScalar >>
         reduceSeq(acc, add)
       ) >> join
      )
    ) >> transpose() >>
    map(transpose) >> transpose
  ) >> join >> transpose
) >> join
```

```
int i = get_global_id(0);
int j = get_global_id(1);
for (int k = 0;k<K/4; k++) {
  for (int l = 0;l<2; l++) {
    for (int m = 0;m<2; m++) {
      float4 t = mult4(
          vload4(A, K*i/2 + k + K*l/4),
          vload4(B, K*j/2 + k + K*m/4)
        );
      vstore4(t, temp, m + 2*N*i + 4*j + 2*l);
      for (int n = 0;n<4; n++) {
        C[m + 2*N*i + 2*j + N*l] +=
          temp[n + 4*m + 8*N*i + 16*j + 8*l];
      }
    }
  }
}
```

$\ldots \Longrightarrow \ldots \gg id$

$f \Longrightarrow$ toPrivate(f)

```
λ (A, B) ↦
  A >> split(m) >> mapGlb_0(λ nRowsOfA ↦
   B >> split(n) >> mapGlb_1(λ mColsOfB ↦
     zip( transpose(nRowsOfA) >> split(k),
          transpose(mColsOfB) >> split(k) ) >>
     reduceSeq(init = make2DArray(n,m, 0.0f) >>
                   toPrivate(mapSeq(mapSeq(id))),
      λ (accTile, (tileOfA, tileOfB)) ↦
       zip(accTile, transpose(tileOfA)) >>
       mapSeq(λ (accRow, rowOfTileOfA) ↦
        zip(accRow, transpose(tileOfB)) >>
        mapSeq(λ (acc, colOfTileOfB) ↦
        zip(rowOfTileOfA >> asVector(k),
            colOfTileOfB >> asVector(k)) >>
          mapSeq(mult4) >> asScalar >>
          reduceSeq(acc, add)
        ) >> join
       )
     ) >> toGlobal(mapSeq(mapSeq(mapSeq(id)))
    >> transpose() >>
    map(transpose) >> transpose
  ) >> join >> transpose
) >> join
```

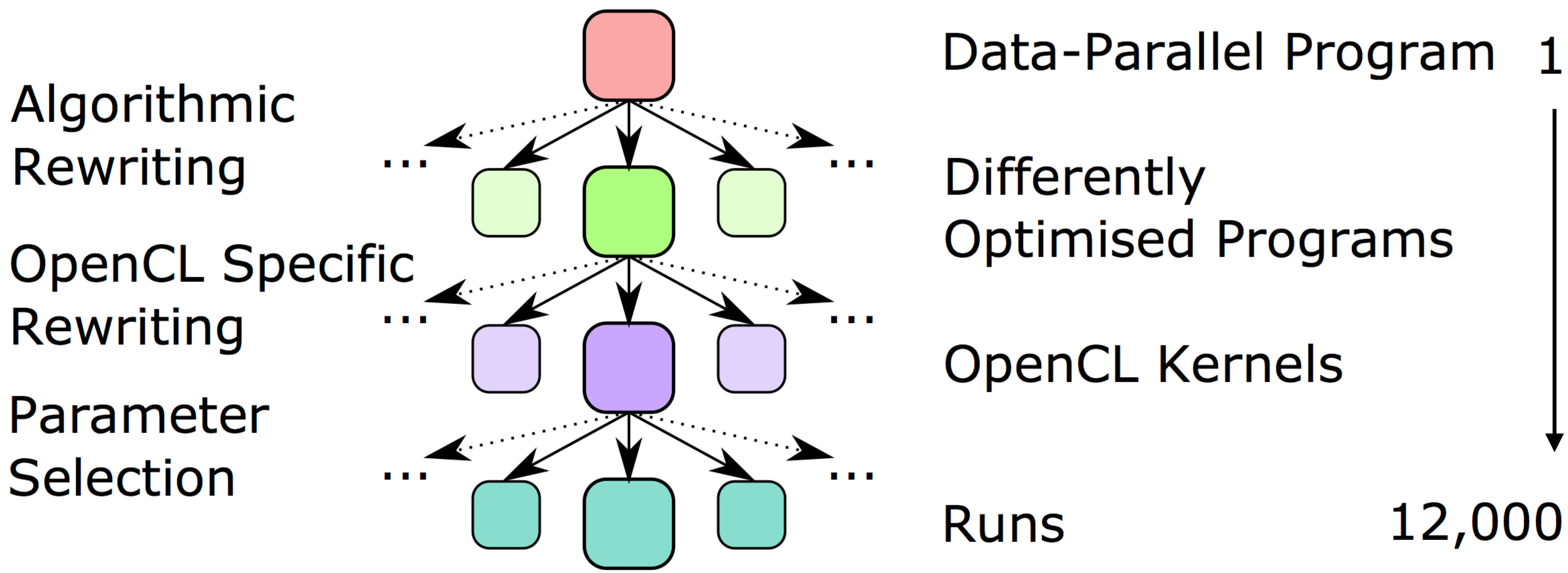
```
int i = get_global_id(0);
int j = get_global_id(1);

float4 temp_0; float4 temp_1;
float4 temp_2; float4 temp_3;
float acc_0; float acc_1;
float acc_2; float acc_3;

for (int k = 0;k<K/4; k++) {

  temp_0 = mult4(vload4(k + K*i/2,A),
    vload4(k + K*j/2,B));
  acc_0 += temp_0.s0 + temp_0.s1 +
    temp_0.s2 + temp_0.s3;

  temp_1 = mult4(vload4(k + K*i/2,A,
    vload4(k + K + 2*K*j/4,B));
  acc_1 += temp_1.s0 + temp_1.s1 +
    temp_1.s2 + temp_1.s3;

  temp_2 = mult4(vload4(k + K + 2*K*i/4,A),
    vload4(k + K*j/2,B));
  acc_2 += temp_2.s0 + temp_2.s1 +
     temp_2.s2 + temp_2.s3;

  temp_3 = mult4(vload4(k + K + 2*K*i/4, A),
    vload4(k + K + 2*K*j/4, B));
  acc_3 += temp_3.s0 + temp_3.s1 +
    temp_3.s2 + temp_3.s3;
}
C[2*N*i + 2*j] = id(acc_0);
C[1 + 2*N*i + 2*j] = id(acc_1);
C[N + 2*N*i + 2*j] = id(acc_2);
C[1 + N + 2*N*i + 2*j] = id(acc_3);
```

```
λ (A, B) ↦
 A >> split(m) >> mapGlb0(λ nRowsOfA ↦
  B >> split(n) >> mapGlb1(λ mColsOfB ↦
    zip( transpose(nRowsOfA) >> split(k),
         transpose(mColsOfB) >> split(k) ) >>
    reduceSeq(init = make2DArray(n,m, 0.0f) >>
              toPrivate(mapSeq(mapSeq(id))),
     λ (accTile, (tileOfA, tileOfB)) ↦
      zip(accTile, transpose(tileOfA)) >>
      mapSeq(λ (accRow, rowOfTileOfA) ↦
       zip(accRow, transpose(tileOfB)) >>
       mapSeq(λ (acc, colOfTileOfB) ↦
       zip(rowOfTileOfA >> asVector(k),
           colOfTileOfB >> asVector(k)) >>
         mapSeq(mult4) >> asScalar >>
         reduceSeq(acc, add)
       ) >> join
      )
    ) >> toGlobal(mapSeq(mapSeq(mapSeq(id)))
    >> transpose() >>
    map(transpose) >> transpose
  ) >> join >> transpose
) >> join
```

```
int i = get_global_id(0);
int j = get_global_id(1);

float4 temp_0; float4 temp_1;
float4 temp_2; float4 temp_3;
float acc_0; float acc_1;
float acc_2; float acc_3;

for (int k = 0;k<K/4; k++) {

  temp_0 = mult4(vload4(k + K*i/2,A),
    vload4(k + K*j/2,B));
  acc_0 += temp_0.s0 + temp_0.s1 +
    temp_0.s2 + temp_0.s3;

  temp_1 = mult4(vload4(k + K*i/2,A,
    vload4(k + K + 2*K*j/4,B));
  acc_1 += temp_1.s0 + temp_1.s1 +
    temp_1.s2 + temp_1.s3;

  temp_2 = mult4(vload4(k + K + 2*K*i/4,A),
    vload4(k + K*j/2,B));
  acc_2 += temp_2.s0 + temp_2.s1 +
     temp_2.s2 + temp_2.s3;

  temp_3 = mult4(vload4(k + K + 2*K*i/4, A),
    vload4(k + K + 2*K*j/4, B));
  acc_3 += temp_3.s0 + temp_3.s1 +
    temp_3.s2 + temp_3.s3;
}
C[2*N*i + 2*j] = id(acc_0);
C[1 + 2*N*i + 2*j] = id(acc_1);
C[N + 2*N*i + 2*j] = id(acc_2);
C[1 + N + 2*N*i + 2*j] = id(acc_3);
```

The generated code is highly optimised for the Mali GPU

Automated Exploration Using Rewrite Rules

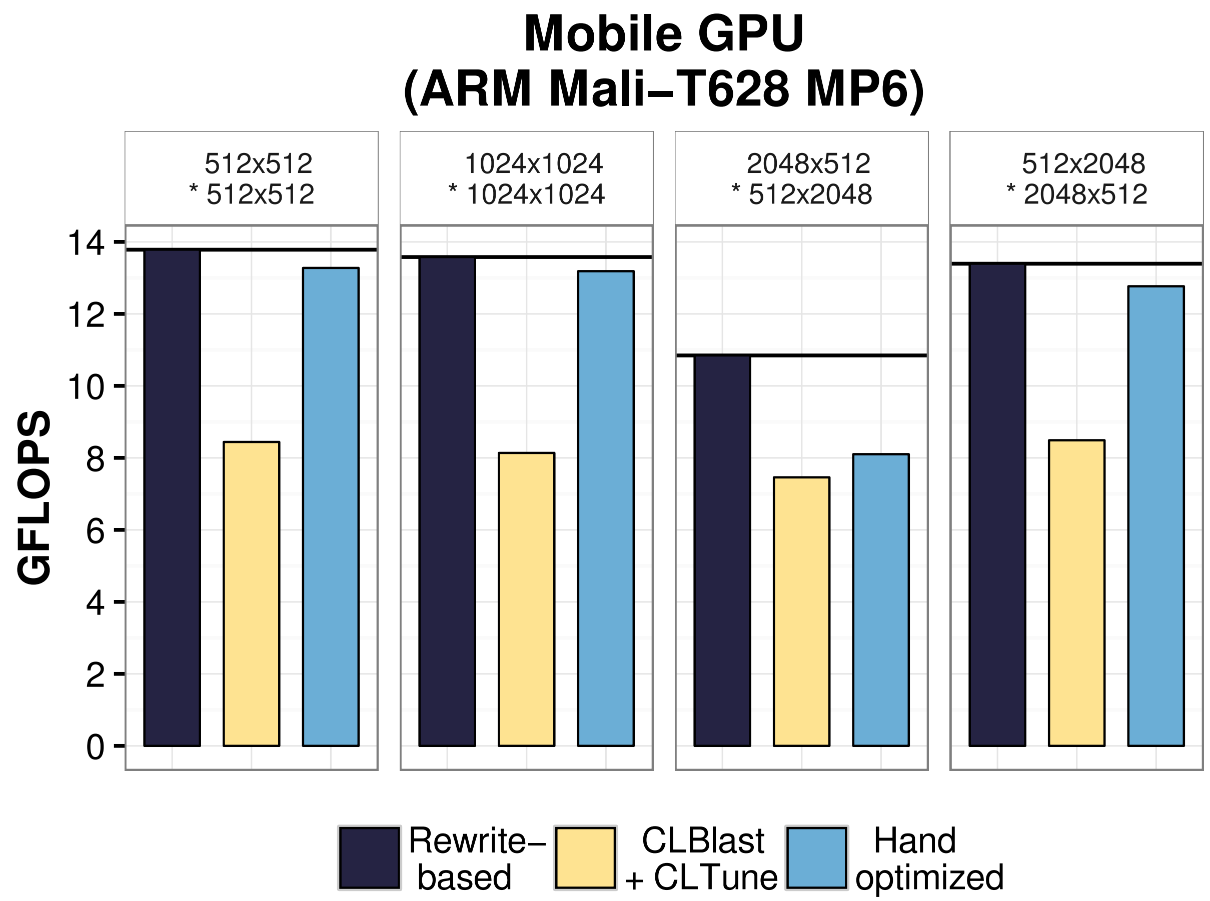


- Exploration using rewrite rules is fully automated

Automated Exploration Using Rewrite Rules

- Apply rules everywhere possible
- Stop after a certain number of applications
- Classical auto-tuning for selecting parameters
- Pick the best kernel for all sizes and devices

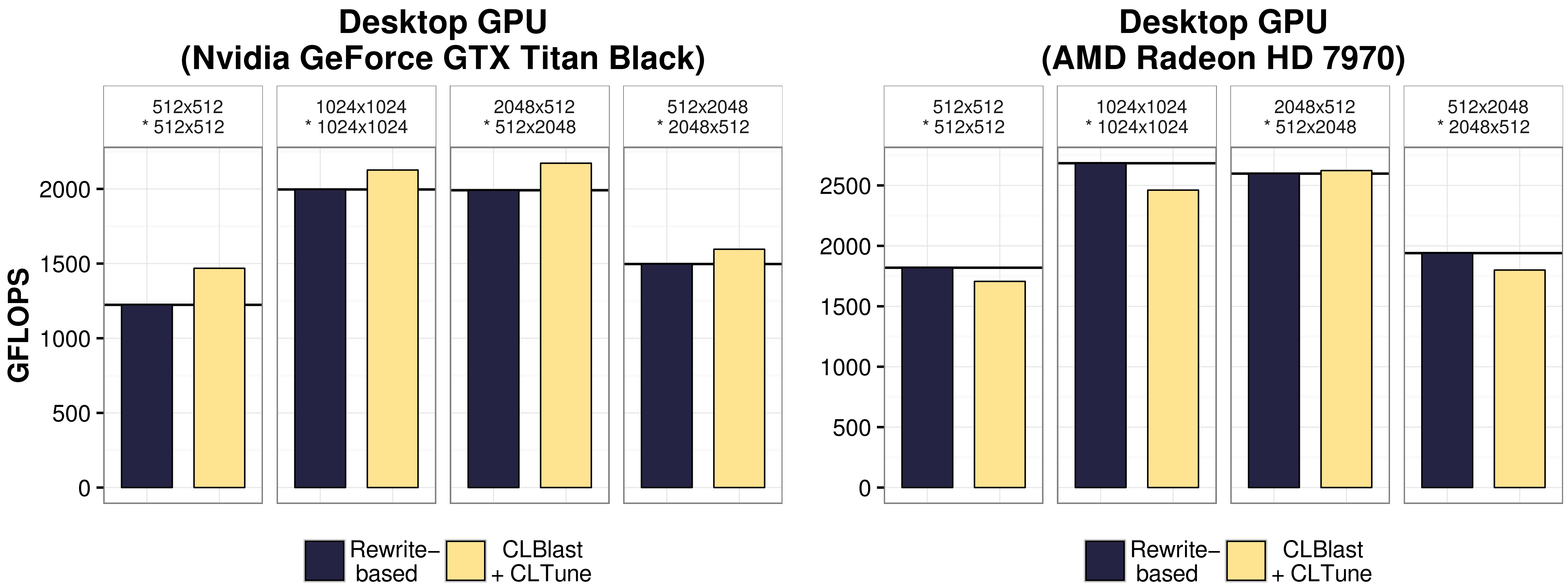
Mali Performance Results



- Our rewrite based approach outperforms hand optimised code on the Mali GPU

THE UNIVERSITY of EDINBURGH
informatics

Performance Portability



- The same methodology achieves good performance across different classes of GPUs

THE UNIVERSITY of EDINBURGH informatics

Extensibility

- New optimisations are expressible as rewrite rules
- Automatically used in the exploration for any program

Extensibility

- New optimisations are expressible as rewrite rules
- Automatically used in the exploration for any program

Example:

OpenCL *dot* built-in rule:

$$\mathrm{zip}(x,\ y) >> \mathrm{mapSeq}(\mathrm{mult4}) >> \mathrm{asScalar} >> \mathrm{reduceSeq}(z,\ \mathrm{add}) \implies \mathrm{dot}(x,\ y) >> \mathrm{reduceSeq}(z,\ \mathrm{add})$$

Before:

```
...
temp = mult4(vload4(k + K*i/2, A),
  vload4(k + K*j/2, B));
acc += temp.s0 + temp.s1 +
  temp.s2 + temp.s3;
...
```

After:

```
...
temp = dot(vload4(k + K*i/2, A),
  vload4(k + K*j/2, B));
acc += temp;
...
```

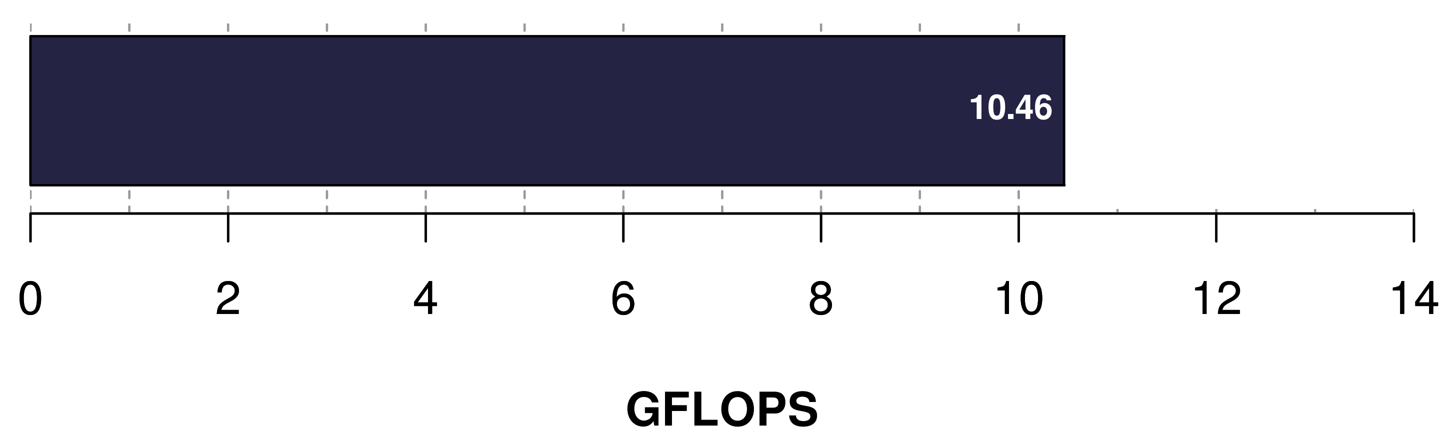
Extensibility

- New optimisations are expressible as rewrite rules
- Automatically used in the exploration for any program

Example:

OpenCL *dot* built-in rule:

$$\mathrm{zip}(x,\ y) \gg \mathrm{mapSeq}(\mathrm{mult4}) \gg \mathrm{asScalar} \gg \mathrm{reduceSeq}(z,\ \mathrm{add}) \Longrightarrow \mathrm{dot}(x,\ y) \gg \mathrm{reduceSeq}(z,\ \mathrm{add})$$



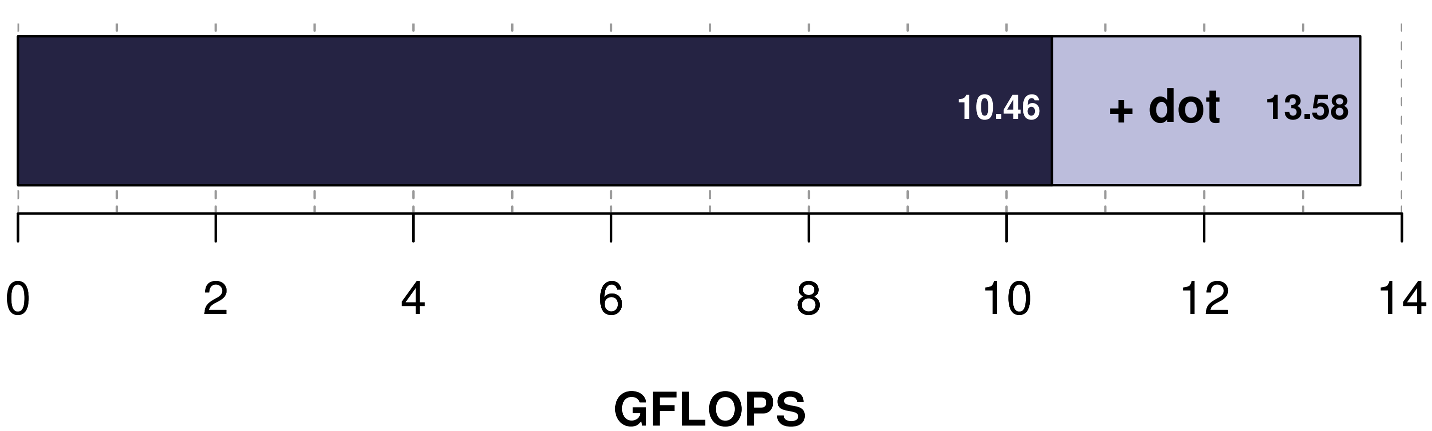
Extensibility

- New optimisations are expressible as rewrite rules
- Automatically used in the exploration for any program

Example:

OpenCL *dot* built-in rule:

$$\text{zip}(x,\ y) >> \text{mapSeq}(\text{mult4}) >> \text{asScalar} >> \text{reduceSeq}(z,\ \text{add})$$
$$\implies \text{dot}(x,\ y) >> \text{reduceSeq}(z,\ \text{add})$$



THE UNIVERSITY *of* EDINBURGH
informatics

Conclusion

- Classical auto-tuning is not performance portable
- Our approach outperforms hand-tuned OpenCL code on Mali, where the auto-tuner fails to deliver
- Easily extensible by adding new rules
- Using a functional approach along with **rewrite rules** we can achieve **performance portability**

Future Work

- Building performance models to avoid executing code in the exploration to evaluate expressions
- Using reinforcement learning to guide the exploration
- Investigating a larger number of optimisations

Toomas Remmelg - toomas.remmelg@ed.ac.uk
http://www.lift-project.org/

Supported by:

THE UNIVERSITY *of* EDINBURGH
informatics