The LIFT Project
Performance Portable Parallel Code Generation via Rewrite Rules

Michel Steuwer, Adam Harries, Naums Mogers, Federico Pizzuti, Toomas Remmelg, Larisa Stolzfus
What are the problem LIFT tries to tackle?

- Parallel processors everywhere
- Many different types: CPUs, GPUs, FPGAs, many more specialised, ...
- Parallel programming is hard
- Optimising is even harder
- **Problem:** No portability of performance!
Case Study: Parallel Reduction

- Optimising OpenCL is complex
- Understanding of target hardware required
- Program changes not obvious
- Is it worth it? ...

Unoptimized Implementation

case Study: Parallel Reduction

- Optimising OpenCL is complex
- Understanding of target hardware required
- Program changes not obvious
- Is it worth it? ...

Unoptimized Implementation
Performance Results Nvidia

(a) Nvidia’s GTX 480 GPU.

- Yes! Optimising improves performance by a factor of 10!
- Optimising is important, but …
Performance Results AMD and Intel

- … unfortunately, optimisations in OpenCL are not portable!

- **Challenge**: how to achieving portable performance?
LIFT: Performance Portable GPU Code Generation via Rewrite Rules

The overview of our pattern-based code generation approach is presented in Figure 5.3. The programmer writes a high-level expression composed of algorithmic patterns. Using a rewrite rule system, we transform this high-level expression into a low-level expression consisting of OpenCL patterns. At this rewrite stage, algorithmic and optimization choices in the high-level expression are explored. The generated low-level expression is then fed into our code generator that emits an OpenCL program which is, finally, compiled to machine code.

**Ambition:** automatic generation of *Performance Portable code*
4.5 Summary

This section describes how we compile a single high-level expression into OpenCL code that can be used to perform matrix multiplication on a GPU. The compilation process involves several phases: algorithmic exploration, OpenCL specific exploration, parameter exploration, and code generation.

5. Exploration and Compilation Strategy

This section will discuss how we explore different algorithmic choices and how we generate OpenCL code from high-level expressions. We use rewrite rules to automate the exploration process and ensure correctness.

5.1 Algorithmic Exploration Using Macro Rules

By design, each rewrite rule encodes a simple transformation. Some optimizations are achieved by composition.

5.2 OpenCL Specific Exploration

The exploration process is divided into four phases, which we discuss in the following: algorithmic exploration, and code generation.

5.3 Parameter Exploration

We perform many meaningless copies, requiring too much private, local, or global memory. We also ensure that the parameters picked will not generate an OpenCL kernel. Furthermore, we make sure that the values are in a reasonable range.

5.4 Code Generation

We ultimately reach a program, which allows us to specify an index function to perform the reordering is delayed until the next primitive.

6. Preliminary Evaluation

We determined that 760 OpenCL specific programs can be generated from around 8000 fully specialized programs. Additionally, we perform an automation of the exploration and code generation process.

7. Conclusion

We started from a matrix multiplication program and used rewrite rules to automate the process of generating OpenCL code. The results show that 46,000 fully specialized programs can be generated from around 8000 fully specialized programs.
Performance Results Matrix Multiplication

Nvidia GeForce GTX 480 (Fermi)

Throughput (Gflop/s)

Input Size

Nvidia GeForce GTX TITAN Black (Kepler)

Throughput (Gflop/s)

Input Size

AMD Radeon HD 7970 (Tahiti)

Throughput (Gflop/s)

Input Size

Performance close or better than hand-tuned MAGMA library
The LIFT Team
Automatic Performance Optimisation via Provably Correct Rewrite-Rules

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

1 for (int i = 0; i<M; i++) {
2   for (int j = 0; j<N; j++) {
3     for (int k = 0; k<K; k++) {
4       temp[k + K*N*i + K*j] =
5       mult(A[k + K*i], B[k + K*j]);
6     }
7     for (int k = 0; k<K; k++) {
8       C[j + N*i] +=
9       temp[k + K*N*i + K*j];
10    }
11  }
12}

2
A >> map(λ rowOfA ↦
   B >> map(λ colOfB ↦
       zip(rowOfA, colOfB) >>
       map(mult) >> reduce(0.0f, add)
   )
)

map(f) =⇒ split(k) >> map(map(f)) >> join

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];
        }
    }
}
\[
\text{A} \ggg \text{map}(\lambda \text{rowOfA} \mapsto \text{B} \ggg \text{map}(\lambda \text{colOfB} \mapsto \\
\text{zip(} \text{rowOfA}, \text{colOfB}) \ggg \\
\text{map(} \text{mult} \mapsto \text{reduce(} 0.0 \text{f}, \text{add}) \\
) \\
)
\]

\[
\text{map}(f) \mapsto \text{split}(k) \ggg \text{map(} \text{map}(f) \mapsto \text{join} \\
)
\]

\[
\text{A} \ggg \text{split}(m) \ggg \text{map}(\lambda \text{rowsOfA} \mapsto \\
\text{rowsOfA} \ggg \text{map}(\lambda \text{rowOfA} \mapsto \\
\text{B} \ggg \text{map}(\lambda \text{colOfB} \mapsto \\
\text{zip(} \text{rowOfA}, \text{colOfB}) \ggg \\
\text{map(} \text{mult} \mapsto \text{reduce(} 0.0 \text{f}, \text{add}) \\
) \\
)
\]

\[
\ggg \text{join}
\]
A >> map(λ rowOfA ↦
B >> map(λ colOfB ↦
    zip(rowOfA, colOfB) >>
    map(mult) >> reduce(0.0f, add)
)
)

map(f) ⟷ split(k) >> map(map(f)) >> join

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];
        }
    }
}

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];
            }
        }
    }
}

)}
\[
\begin{align*}
A \gg\gg & \text{split}(m) \gg\gg \text{map}(\lambda \text{rowsOfA} \mapsto \\
& \quad \text{rowsOfA} \gg\gg \text{map}(\lambda \text{rowOfA} \mapsto \\
& \quad \quad \text{B} \gg\gg \text{map}(\lambda \text{colOfB} \mapsto \\
& \quad \quad \quad \text{zip}(\text{rowOfA}, \text{colOfB}) \gg\gg \\
& \quad \quad \quad \text{map}(\text{mult}) \gg \text{reduce}(0.0f, \text{add}) \\
& \quad \})
\end{align*}
\]

\[
\begin{align*}
& \text{for (int } i = 0; i < M/2; i++) \{ \\
& \quad \text{for (int } l = 0; l < 2; l++) \{ \\
& \quad \quad \text{for (int } j = 0; j < N; j++) \{ \\
& \quad \quad \quad \text{for (int } k = 0; k < K; k++) \{ \\
& \quad \quad \quad \quad \text{temp}[k + 2*K*N*i + K*N*l + K*j] = \\
& \quad \quad \quad \quad \quad \text{mult}(A[k + K*l + 2*K*i], B[k + K*j]); \\
& \quad \quad \quad \} \\
& \quad \quad \} \\
& \quad \} \\
& \} \\
\end{align*}
\]
A >> split(m) >> map(λ rowsOfA ↦
rowsOfA >> map(λ rowOfA ↦
B >> map(λ colOfB ↦
zip(rowOfA, colOfB) >>
map(mult) >> reduce(0.0f, add)
))
)
) >> join

\[
\begin{align*}
X &\map(\lambda \ x \mapsto \ Y \map(\lambda \ y \mapsto f)) \\
Y &\map(\lambda \ y \mapsto X \map(\lambda \ x \mapsto f)) \mapsto \text{transpose}
\end{align*}
\]

for (int i = 0; i < M/2; i++) {
    for (int l = 0; l < 2; l++) {
        for (int k = 0; k < K; k++) {
            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];
                }
            }
        }
    }
}
\[
\begin{align*}
\text{A} & \gg \text{split}(m) \gg \text{map}(\lambda \text{ rowsOfA} \mapsto \\
& \quad \text{rowsOfA} \gg \text{map}(\lambda \text{ rowOfA} \mapsto \\
\text{B} & \gg \text{map}(\lambda \text{ colOfB} \mapsto \\
& \quad \text{zip}(\text{rowOfA}, \text{colOfB}) \gg \\
& \quad \text{map}(\text{mult}) \gg \text{reduce}(0.0f, \text{add}) \\
\) \\
& ) \gg \text{join} \\
\) \\
X & \gg \text{map}(\lambda \ x \mapsto Y \gg \text{map}(\lambda \ y \mapsto f)) \\
\) \\
Y & \gg \text{map}(\lambda \ y \mapsto X \gg \text{map}(\lambda \ x \mapsto f)) \gg \text{transpose} \\
\) \\
\text{A} & \gg \text{split}(m) \gg \text{map}(\lambda \text{ rowsOfA} \mapsto \\
& \quad \text{rowsOfA} \gg \text{map}(\lambda \text{ rowOfA} \mapsto \\
\text{B} & \gg \text{map}(\lambda \text{ colOfB} \mapsto \\
& \quad \text{zip}(\text{rowOfA}, \text{colOfB}) \gg \\
& \quad \text{map}(\text{mult}) \gg \text{reduce}(0.0f, \text{add}) \\
\)) \\
& ) \gg \text{transpose} \\
& ) \gg \text{join} \\
\end{align*}
\]
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*l + K*N*l + K*N*i + K*N*l + K*N + i] =
        mult(A[k + K*l + 2*K*i], B[k + K*j]);
      }
      C[j + N*l + 2*N*i] +=
      temp[k + 2*K*N*i + K*N*l + K*N + i];
    }
  }
}
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

map(f) ⇔ split(k) >> map(map(f)) >> join
A >> split(m) >> map(\(\lambda\) rowsOfA \rightarrow 
B >> map(\(\lambda\) colOfB \rightarrow 
rowsOfA >> map(\(\lambda\) rowOfA \rightarrow 
zip(rowOfA, colOfB) >> 
map(mult) >> reduce(0.0f, add) 
)
) >> transpose
) >> join

map(f) \rightarrow split(k) >> map(map(f)) >> join

A >> split(m) >> map(\(\lambda\) rowsOfA \rightarrow 
B >> split(n) >> map(\(\lambda\) colsOfB \rightarrow 
colsOfB >> map(\(\lambda\) colOfB \rightarrow 
rowsOfA >> map(\(\lambda\) rowOfA \rightarrow 
zip(rowOfA, colOfB) >> 
map(mult) >> reduce(0.0f, add) 
)
)
) >> join >> transpose
) >> join

---

```java
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]);
            }
        }
    }
}
```
A >> split(m) >> map(\( \lambda \) rowsOfA \( \mapsto \))
B >> map(\( \lambda \) colOfB \( \mapsto \))
rowsOfA >> map(\( \lambda \) rowOfA \( \mapsto \))
zip(rowOfA, colOfB) >>
map(mult) >> reduce(0.0f, add)
)
) >> transpose
) >> join

map(f) \Rightarrow split(k) >> map(map(f)) >> join

A >> split(m) >> map(\( \lambda \) rowsOfA \( \mapsto \))
B >> split(n) >> map(\( \lambda \) colsOfB \( \mapsto \))
colsOfB >> map(\( \lambda \) colOfB \( \mapsto \))
rowsOfA >> map(\( \lambda \) rowOfA \( \mapsto \))
zip(rowOfA, colOfB) >>
map(mult) >> reduce(0.0f, add)
)
)
) (join) >> transpose
) >> join
A >> split(m) >> map(λ rowsOfA 💸 m/2; i++) {
  B >> split(n) >> map(λ colsOfB 💸 n/2; j++) {
    colsOfB >> map(λ colOfB 💸 K; k++) {
      rowsOfA >> map(λ rowOfA 💸 K; l++) {
        temp[k + 4*K*N*i + 2*K*N*i + 2*K*j + K*m] =
        mult(A[k + K*i + 2*K*i] B[k + K*m + 2*K*j]);
      }
    }
  }
}
for (int m = 0; m < 2; m++) {
  for (int l = 0; l < 2; l++) {
    for (int k = 0; k < K; k++) {
      for (int j = 0; j < N/2; j++) {
        for (int i = 0; i < M/2; i++) {
          C[m + 2*j + 2*N*i + 4*N*i] +=
          temp[k + 4*K*N*i + 2*K*N*i + 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

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

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(\( \lambda \) rowsOfA \( \mapsto \) B >> split(n) >> map(\( \lambda \) colsOfB \( \mapsto \) colsOfB >> map(\( \lambda \) colOfB \( \mapsto \) rowsOfA >> map(\( \lambda \) rowOfA \( \mapsto \) zip(rowOfA, colOfB) >> map(mult) >> reduce(0.0f, add) ) ) ) >> join >> transpose ) >> join

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

```
for (int i = 0; i<M/2; i++) {
    for (int j = 0; j<N/2; j++) {
        for (int m = 0; m<K; 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 >> \text{split}(m) >> \text{map}(\lambda \text{rowsOfA} \mapsto 
\text{B} >> \text{split}(n) >> \text{map}(\lambda \text{colsOfB} \mapsto 
\text{colsOfB} >> \text{map}(\lambda \text{colOfB} \mapsto 
\text{rowsOfA} >> \text{map}(\lambda \text{rowOfA} \mapsto 
\begin{align*}
zip & (\text{rowOfA}, \text{colOfB}) \mapsto 
\text{map}(\text{mult}) \mapsto \text{reduce}(0.0f, \text{add})
\end{align*}
\mapsto 
\text{join} \mapsto \text{transpose}
\mapsto 
\end{align*}
\mapsto 
\text{join} \mapsto \text{transpose}
\mapsto 
\end{align*}
\mapsto \text{join} \mapsto \text{transpose} \mapsto 
\begin{align*}
X >> \text{map}(\lambda \text{x} \mapsto Y >> \text{map}(\lambda \text{y} \mapsto f))
Y >> \text{map}(\lambda \text{y} \mapsto X >> \text{map}(\lambda \text{x} \mapsto f)) \mapsto 
\begin{align*}
\text{for (int } i = 0; i < M/2; i++) \{ 
\text{for (int } j = 0; j < N/2; j++) \{ 
\text{for (int } m = 0; m < 2; m++) \{ 
\text{for (int } k = 0; k < K; k++) \{ 
\text{ temp}[k + 4*K*N*i + 2*K*N*l + 2*K*j] + K*m] = 
\text{ mult}[A[k + K*l + 2*K*i], B[k + K* m + 2*K*j]]; 
\} 
\} 
\} 
\} 
X \mapsto 
\begin{align*}
X >> \text{map}(\lambda \text{x} \mapsto Y >> \text{map}(\lambda \text{y} \mapsto f)) \mapsto 
\begin{align*}
\text{for (int } i = 0; i < M/2; i++) \{ 
\text{for (int } j = 0; j < N/2; j++) \{ 
\text{for (int } l = 0; l < 2; l++) \{ 
\text{for (int } m = 0; m < 2; m++) \{ 
\text{for (int } k = 0; k < K; k++) \{ 
\text{ temp}[k + 4*K*N*i + 2*K*N*l + 2*K*j] + K*m] = 
\text{ mult}[A[k + K*l + 2*K*i], B[k + K* m + 2*K*j]]; 
\} 
\} 
\} 
\} 
\} 
\end{align*}
\text{for (int } i = 0; i < M/2; i++) \{ 
\text{for (int } j = 0; j < N/2; j++) \{ 
\text{for (int } m = 0; m < 2; m++) \{ 
\text{for (int } k = 0; k < K; k++) \{ 
\text{ temp}[k + 4*K*N*i + 2*K*N*l + 2*K*j] + K*m] = 
\text{ mult}[A[k + K*l + 2*K*i], B[k + K* m + 2*K*j]]; 
\} 
\} 
\} 
\}
After algorithmic rewrites...
Tiled Matrix Multiplication

\[
\lambda (A, B) \mapsto \\
A >> \text{split}(m) >> \text{map}(\lambda nRowsOfA \mapsto \\
B >> \text{split}(n) >> \text{map}(\lambda mColsOfB \mapsto \\
\text{zip}(\text{transpose}(nRowsOfA) >> \text{split}(k), \\
\text{transpose}(mColsOfB) >> \text{split}(k)) >> \\
\text{reduceSeq}(\text{init} = \text{make2DArray}(n, m, 0.0f), \\
\lambda (accTile, (tileOfA, tileOfB)) \mapsto \\
\text{zip}(accTile, \text{transpose}(tileOfA)) >> \\
\text{map}(\lambda (accRow, rowOfTileOfA) \mapsto \\
\text{zip}(accRow, \text{transpose}(tileOfB)) >> \\
\text{map}(\lambda (acc, colOfTileOfB) \mapsto \\
\text{zip}(rowOfTileOfA, colOfTileOfB) >> \\
\text{map}(\text{mult}) >> \text{reduce}(acc, \text{add}) \\
) >> \text{join}
)
) >> \text{transpose}() >> \\
\text{map}(\text{transpose}) >> \text{transpose}
) >> \text{join} >> \text{transpose}
) >> \text{join}
\]

```java
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 Find Good Implementations?

• Identifying rule sequences beneficial across
  • optimisations
  • programs

• Smarter application of OpenCL specific rules
  • Local/Private memory when reuse
How to Find Good Implementations?

• Performance Modelling

New programs → Extracted Features

\[(x_1, y_1, z_1, \ldots)\]
\[(x_2, y_2, z_2, \ldots)\]
\[\vdots\]
\[(x_N, y_N, z_N, \ldots)\]

→ Query Model

\[(t_1, t_2, t_3, \ldots)\]

Generate OpenCL?
SPARSE DATA STRUCTURES IN LIFT

Research: Adding data structure primitives to support sparse data layouts

- Extend Lift language with notion of Dynamic arrays

- Composable array structure with low level space optimisations

```
StaticArray(
    DynamicArray((Int, T)),
    M)
```
LOAD BALANCING PRIMITIVES

Research: introducing dynamic work assignment in a compositional manner

- Extending set of primitives in lift with dynamic load balancing variants
- Maintaining composition while mitigating irregularity

\[
\text{Map}(\lambda(\text{row} \rightarrow \ldots), \text{matrix})
\]

MapStatic

MapDynamic

Thread 1
Thread 2

Time
FUTURE WORK

Research: Lifting Lift out of the GPU

Map(
  \lambda(row \rightarrow \ldots),
  \text{matrix}
)
FUTURE WORK

Research: Lifting Lift out of the GPU

Map(λ(row → ...), matrix)
Parallelizing non-associative sequential reductions

By Federico Pizzuti
Supervisor: Cristophe Dubach
Associative reductions are parallelizable

Summation of Numbers: \( \sum_{0}^{n} x_n = x_1 + \ldots + x_n \)

\(\text{reduce}(+, [1, 2, 3, 4, 5, 6, 7, 8])\)

The operator + is \textit{associative}, so this reduction is \textit{parallelizable}
What about non-associative reductions?

Polynomial evaluation: $\sum_{0}^{n} x_n \cdot k^n$

Define operator: $a \odot b = k \cdot a + b$

```
\text{reduce} (\odot, [1, 2, 3, 4, 5, 6, 7, 8])
```

The operator $\odot$ is \textbf{not associative}, so this reduction must be executed \textbf{sequentially}

How to parallelize non-associative reductions?
Key insight: Matrix multiplication ($\times$) is associative*

We rewrite the reduction using $\circledast$ in terms of a reduction that uses $\times$

The derived reduction is then parallelizable

*This works for all operators expressed in terms of semiring operations, not just $+$ and $\cdot$. 
Rewrite reduction operator as matrix multiplication

Rearrange data as matrices

\[
\begin{bmatrix}
1, 2, 3, 4, 5, 6, 7, 8
\end{bmatrix}
\times
\begin{bmatrix}
k_1 & 1 \\
0 & 1
\end{bmatrix}
\times
\begin{bmatrix}
k_2 & 0 \\
0 & 1
\end{bmatrix}
\times
\begin{bmatrix}
k_3 & 1 \\
0 & 1
\end{bmatrix}
\times
\begin{bmatrix}
k_4 & 1 \\
0 & 1
\end{bmatrix}
\times
\begin{bmatrix}
k_5 & 1 \\
0 & 1
\end{bmatrix}
\times
\begin{bmatrix}
k_6 & 1 \\
0 & 1
\end{bmatrix}
\times
\begin{bmatrix}
k_7 & 0 \\
0 & 1
\end{bmatrix}
\times
\begin{bmatrix}
k_8 & 0 \\
0 & 1
\end{bmatrix}
\]

\[
\text{reduce}(\odot, [1, 2, 3, 4, 5, 6, 7, 8]) = \mathbf{x}
\]

\[
\text{reduce}(\odot, \begin{bmatrix}
k_1 & 1 \\
0 & 1
\end{bmatrix}, \ldots, \begin{bmatrix}
k_8 & 0 \\
0 & 1
\end{bmatrix}) = \begin{bmatrix}
a & \mathbf{x} \\
0 & 1
\end{bmatrix}
\]
Decomposing Stencil Computations

(a) Neighborhood: accessing neighboring elements according to stencil shape
(b) Boundary Handling: what happens at the border of the input array?
(c) Stencil Function: compute single output element for a given neighborhood
(a) Create Neighborhoods using Slide

**slide** (size, step, in)

\[
\text{slide}(3, 1, [a, b, c, d, e]) =
[[a, b, c], [b, c, d], [c, d, e]]
\]

**slide** : \((size : \text{Int}, \ step : \text{Int}, \ in : [\mathcal{T}]_n) \rightarrow [[\mathcal{T}]_{\text{size}}]_{n - \text{size + step}}^\text{step}\)
(b) Boundary Handling using Pad

pad (reindexing)

clamp(i, n) = (i < 0) ? 0 : ((i >= n) ? n-1 : i)

pad (1, 1, clamp, [a, b, c, d]) = [a, a, b, c, d, d]
(c) Apply Stencil Function using Map

\[ \text{map}(nbh \Rightarrow \text{reduce}(\text{add}, 0.0f, nbh), \quad \left[ \begin{array}{c} [0, 1, 2], [1, 2, 3] \end{array} \right]) = \left[ \begin{array}{c} [3], [6] \end{array} \right] \]
Multidimensional Stencil Computations

**Idea:** Express complex computations as compositions of simple primitives

\[ \text{map}_n(f, \text{slide}_n(\text{size}, \text{step}, \text{pad}_n(l, r, h, \text{input}))) \]

\[ \text{map}_2(f, \text{slide}_2(\text{size}, \text{step}, \text{pad}_2(l, r, h, \text{in}))) \]
Exploiting Locality through Overlapped Tiling:

**Locality** Close neighborhoods share elements that can be grouped in tiles

**Local Memory** On GPUs, local memory can be used to cache tiles.

**Overlap** The shape of the stencil determines the overlap at the edges of tiles
3D Wave Simulations in Lift

**FDTD** is a common approach to modelling the 3D wave equation

\[
\frac{\partial^2 \Psi}{\partial t^2} = c^2 \nabla^2 \Psi
\]

However it can be difficult to abstract out boundary conditions, in particular absorbing boundary conditions.

ground penetrating radar

room acoustics simulations
Implementing Basic Room Acoustics Simulations

- The *slide* primitive makes more memory accesses than necessary for smaller stencils.
- The *pad* primitive does not supply functionality for constant boundary conditions, only different index accesses.
- Two new primitives were implemented to accommodate these shortcomings: *select* and *boundary*.

**Selection of stencil shape (select)**

Targeting different groups of neighbouring values.

LIFT: $\text{map}(\text{reduce}(\text{ngh}[-1]+\text{ngh}[0]+\text{ngh}[1] \circ \text{slide}(\text{size}, \text{step})) \leftarrow \text{input}$

C: $\text{for}(i = 0 \text{ to } n) \quad \text{updated}[i] = \text{data}[i-1]+\text{data}[i]+\text{data}[i+1]$

**On-the-fly boundary handling (boundary)**

Creating masking values when needed instead of storing in memory.

LIFT: $\text{boundaryValue} = \text{ArrayGen}(\text{idx}) \times \text{border1} + !\text{ArrayGen}(\text{idx}) \times \text{border2}$

C: $\text{boundaryValue} = \text{if}(\text{idx} \leq M \text{ or } \text{idx} > 0) \text{ return } 1.0 \text{ else return } 0.0$
Results from Select and Boundary Optimisations

comparable results to original room acoustics benchmark, but still slower than same optimised benchmark
To close the performance gap, implementing the commonly used 3D stencil optimisation “2.5D Tiling”

- Performs calculations in two dimensions in parallel and the third sequentially
- Performance improvements of up to 5-15% seen in original room acoustics simulations when used in conjunction with local memory
Develop a workflow for creating performant, portable, productive 3D wave models.
Develop a workflow for creating performant, portable, productive 3D wave models.

So far, this top layer has been ignored!

There are a wide range of stencil-focused DSLs that could be extended to accommodate 3D wave models and compile into Lift.
Computational Efficiency Optimisation Of Convolutional Neural Networks Using A Functional Data-Parallel Language

By Naums Mogers
Supervisor: Christophe Dubach
Neural networks (NNs) depend on hardware-specific low-level optimizations.

Manual approach:

- Requires expertise in both machine learning and performance programming
- Costly to develop and maintain
- Hard to port to new platforms

Automated approaches:

- Caffe, Tensorflow, Theano, Torch have limited functional and performance portability
- Autotuners are not performance-portable because of no structural optimizations
1. Use 

**Lift**, a functional data-parallel language

- Abstracted from hardware, pure and safe
- Compiles to OpenCL, which supports low-level optimisations
- Implements device-specific and tunable optimisational methods as rewrite rules
1. Use **Lift**, a functional data-parallel language
   ○ Abstracted from hardware, pure and safe
   ○ Compiles to OpenCL, which supports low-level optimisations
   ○ Implements device-specific and tunable optimisational methods as rewrite rules

2. Extend Lift to support NN-specific primitives such as:
   ○ `conv`, `norm`, `pool`, `fully_connected`

3. Implement a set of fine-grained generic optimizations
1. Implementation of Convolutional and Fully Connected layers in Lift
1. Implementation of **Convolutional** and **Fully Connected** layers in Lift

2. Implementation of optimisations such as:
   - Input tiling
   - Weighted summation sequentialisation
   - Kernel grouping
   - Neuron grouping
   - Memory locality exploitation
   - Coalesced memory access
1. Implementation of Convolutional and Fully Connected layers in Lift

2. Implementation of optimisations such as:
   - Input tiling
   - Weighted summation sequentialisation
   - Kernel grouping
   - Neuron grouping
   - Memory locality exploitation
   - Coalesced memory access

3. Explorational framework
Convolutional layer: the algorithm

Sliding

1D Slide example
Convolutional layer: the algorithm

1D Slide example

Sliding → Weighted summation

Kernel → Image → Feature map
Convolutional layer: optimisation

- Input tiling
Convolutional layer: optimisation

- Input tiling
Convolutional layer: optimisation

- Input tiling
Convolutional layer: optimisation

- Input tiling
Convolutional layer: optimisation

- Input tiling
Convolutional layer: optimisation

- Input tiling
- Tiled sliding
Convolutional layer: optimisation

- Input tiling
- Tiled sliding
Convolutional layer: optimisation

- Input tiling
- Tiled sliding
Convolutional layer: optimisation

Weighted summation sequentialisation

- Process multiple pixels in each thread **sequentially**
- Store intermediary results in **private** memory
  ↓  ↓  ↓
- **Reduce** the number of local memory accesses
Convolutional layer: optimisation

**Weighted summation sequentialisation**

- Process multiple pixels in each thread *sequentially*
- Store intermediary results in *private* memory
  ↓  ↓  ↓
- Reduce the number of local memory accesses

**Kernel grouping**

- Kernels are much smaller than images, therefore *multiple* kernels can be processed in the same OpenCL workgroup
  ↓  ↓  ↓
- Reduce the number of global memory accesses by reusing input data stored in local memory (*exploit memory locality*)
Convolutional layer: the Lift expression

Conv = λ((K, B, X) => { }
  MapWr0(0)。(λ((inputs_batch) => { }
    MapWr1(1).λ((input_tile) => { }
      MapWr2(2).λ((kernels_group) => { }
        MapLc1(0).λ((pass_window) => { }
          ReduceWindowAndAddBias() o 
        MapLc1(2).λ((window_row, kernels_row) => { }
          MapLc1(1).SecondPartialReduction()
          /* Reduce and load a row into local memory */) o 
        JoinSequences() o 
        MapLc1(1).FirstPartialReduction()
        /* Weigh and reduce a single tuple of elements sequentially */) o 
        Split(els_per_workitem) o ZipWithInput(window_row) $ kernels_row 
      }) $ Zip(pass_window, kernels_group) 
    }) $ LoadWindowIntoLocal() $ input_tile 
  }) $ GroupKernels(kernelPerGroup)(K, B) 
}) $ inputs_batch 
}) o SlideX() $ X)
1. Add **new optimisational methods**: generic and domain-specific.

2. Investigate ways of **optimising back-propagation**.

3. Investigate the possibility of using **ML to choose** optimisational parameters.
LIFT is Open-Source Software

http://www.lift-project.org/
https://github.com/lift-project/lift
The LIFT Project
Performance Portable Parallel Code Generation via Rewrite Rules

www.lift-project.org  @LIFTlang