University of Glasgow

The LIFT Project

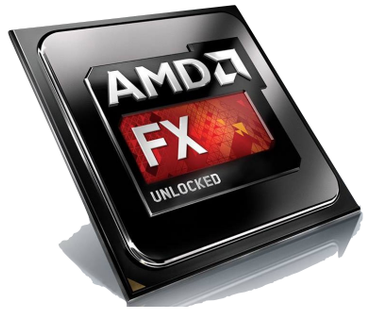
Performance Portable Parallel Code Generation via Rewrite Rules

Michel Steuwer, Adam Harries, Naums Mogers,
Federico Pizzuti, Toomas Remmelg, Larisa Stolzfus

INSPIRING PEOPLE

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!

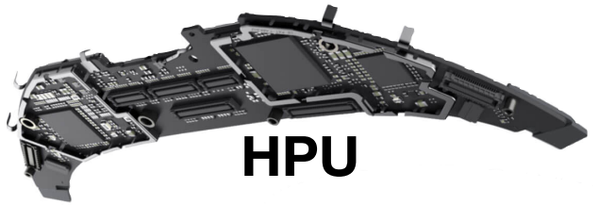


CPU

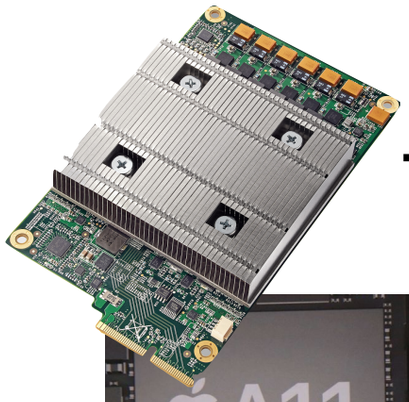
GPU

FPGA

Accelerator

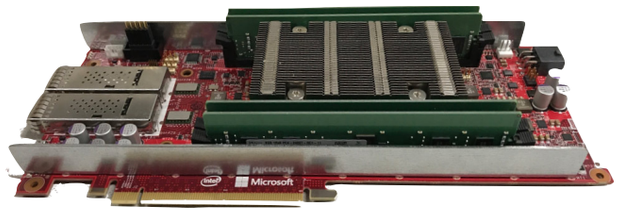


HPU



TPU

Brainwave



Neural Engine

Case Study: Parallel Reduction

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

```
kernel
void reduce0(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);

  for (unsigned int s=1;
       s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid == 0)
    g_odata[get_group_id(0)] = l_data[0];
}
```

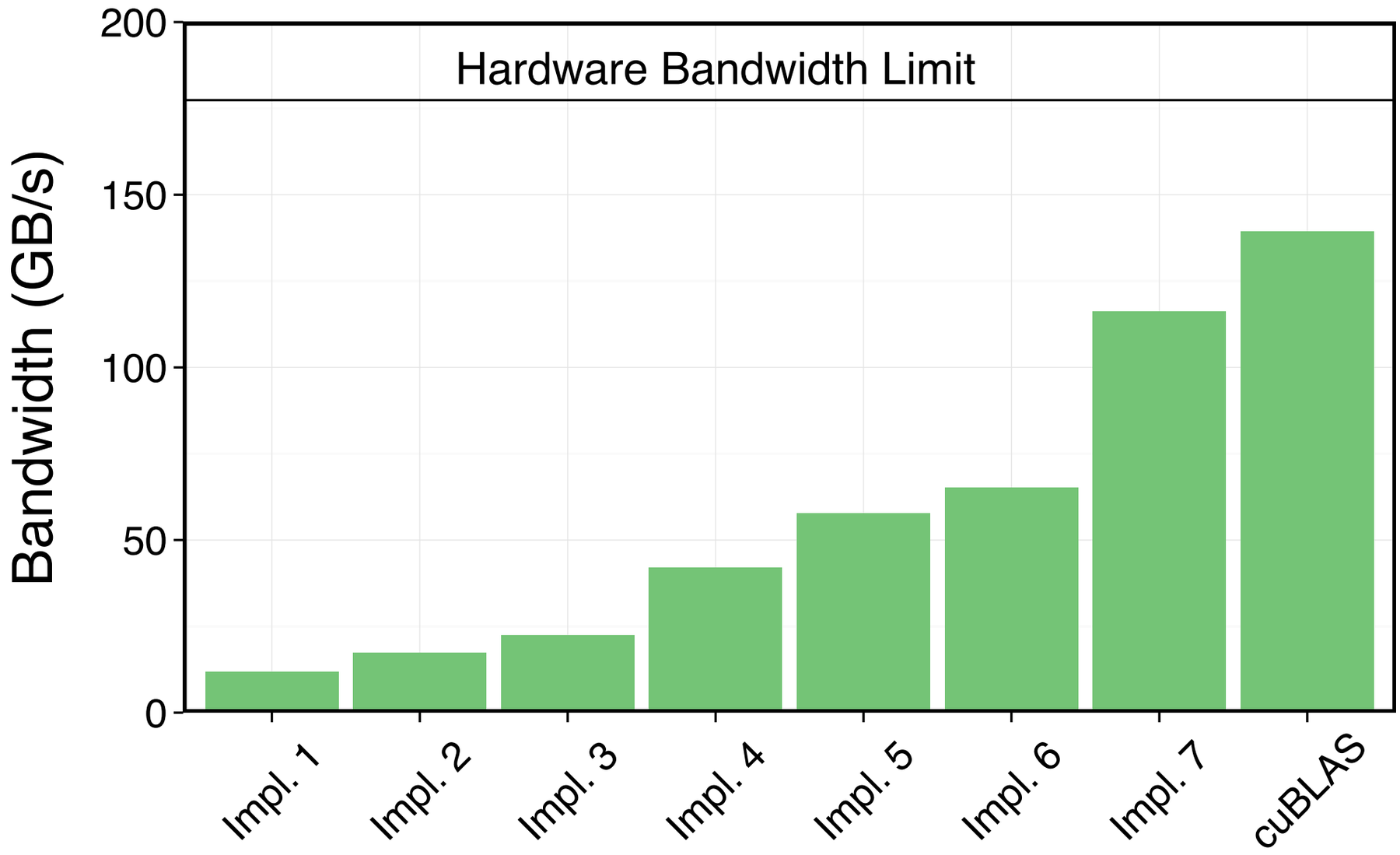
Unoptimized Implementation

```
kernel
void reduce6(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i =
    get_group_id(0) * (get_local_size(0)*2)
                    + get_local_id(0);
  unsigned int gridSize =
    WG_SIZE * get_num_groups(0);
  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_idata[i];
    if (i + WG_SIZE < n)
      l_data[tid] += g_idata[i+WG_SIZE];
    i += gridSize; }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (WG_SIZE >= 256) {
    if (tid < 128) {
      l_data[tid] += l_data[tid+128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (WG_SIZE >= 128) {
    if (tid <  64) {
      l_data[tid] += l_data[tid+ 64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (WG_SIZE >= 64) {
      l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) {
      l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) {
      l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) {
      l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) {
      l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) {
      l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0)
    g_odata[get_group_id(0)] = l_data[0];
}
```

Fully Optimized Implementation

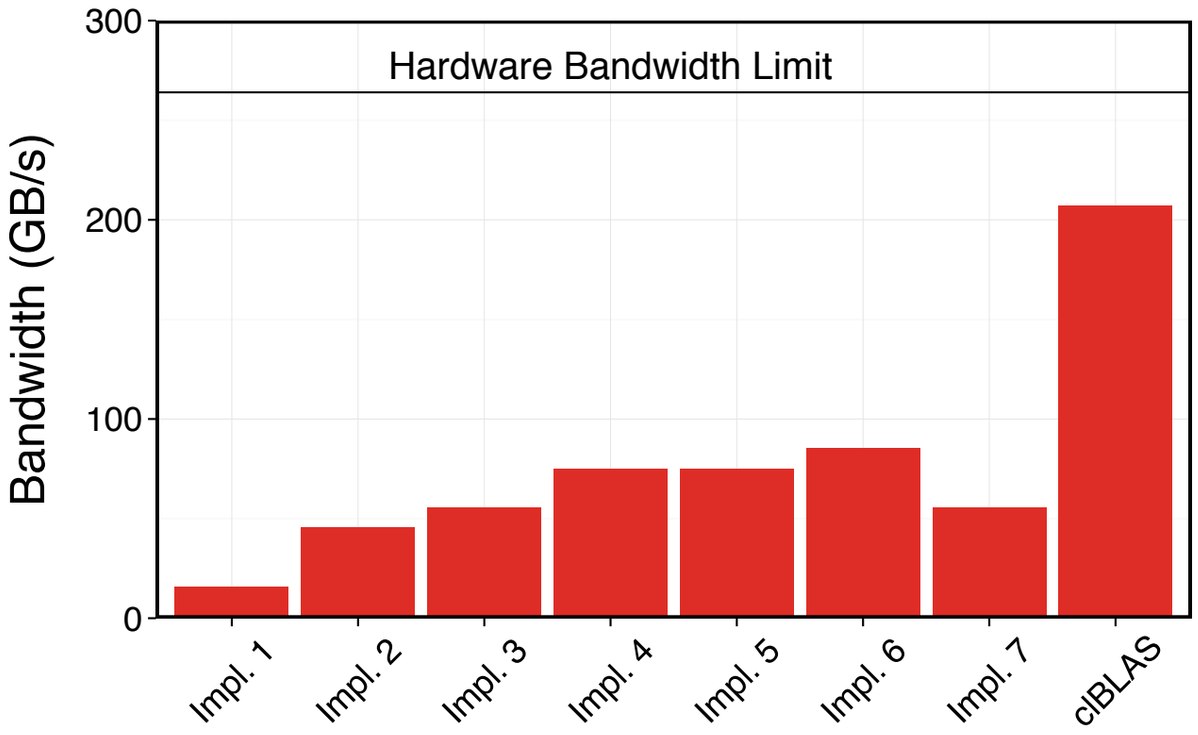
Performance Results Nvidia



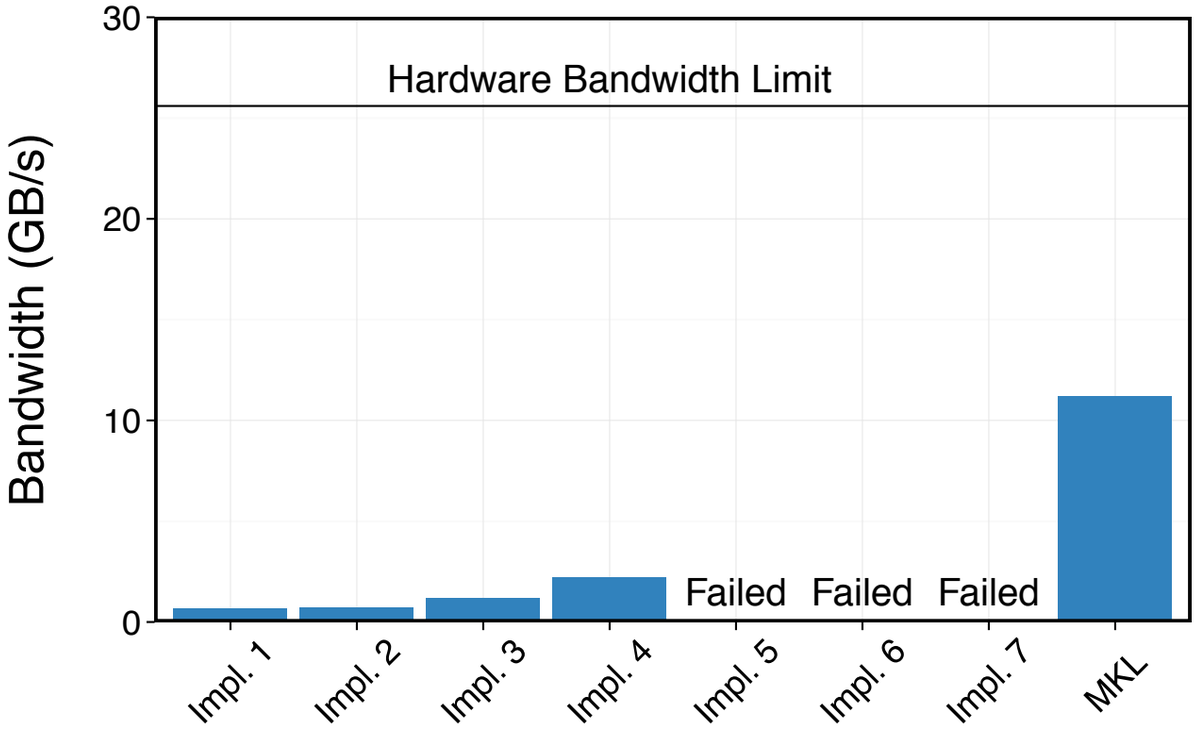
(a) Nvidia's GTX 480 GPU.

- … Yes! Optimising improves performance by a factor of **10**!
- Optimising is important, but …

Performance Results AMD and Intel



(b) AMD's HD 7970 GPU.



(c) Intel's E5530 dual-socket CPU.

- … unfortunately, optimisations in OpenCL are not portable!

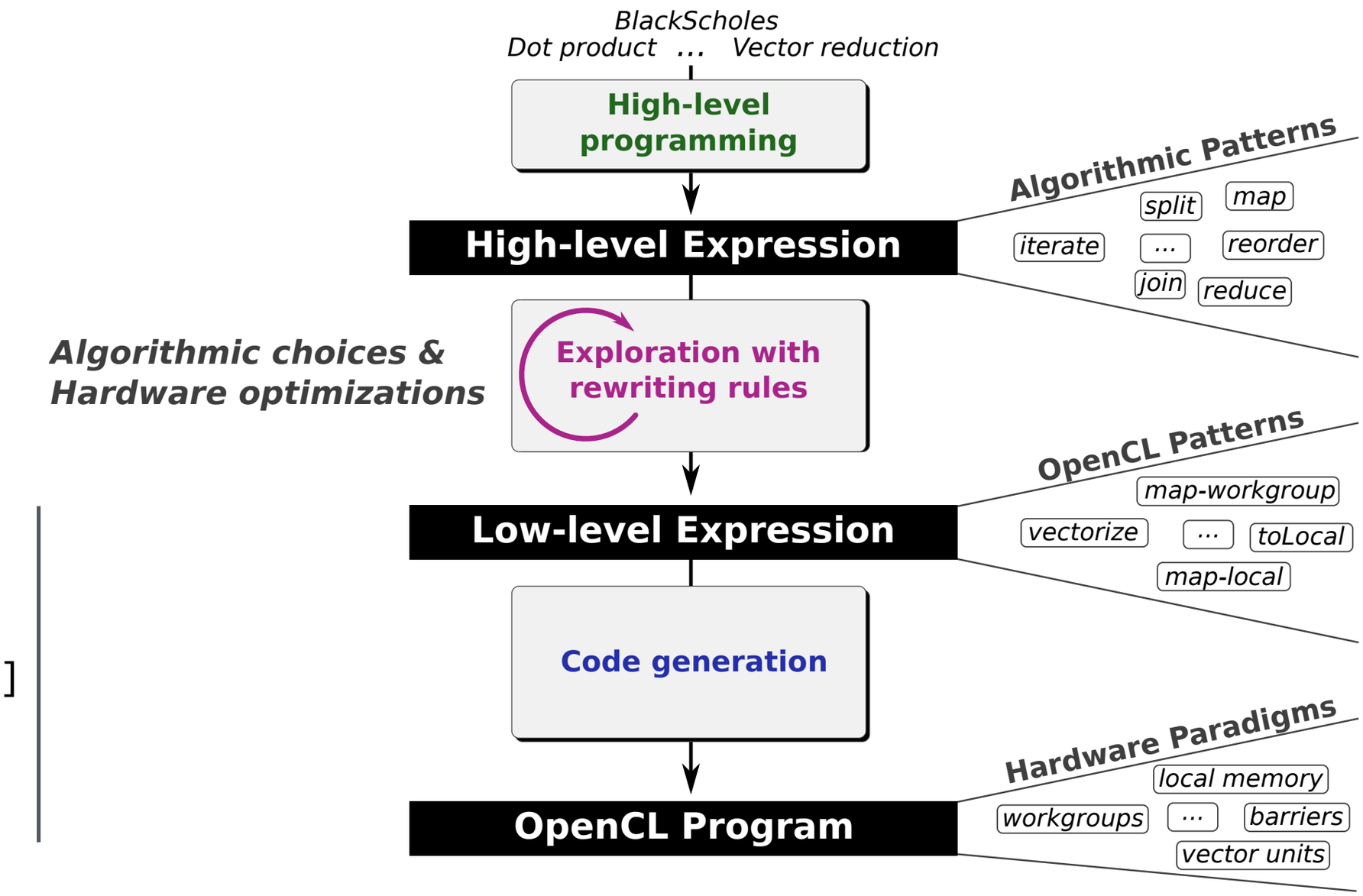
- **Challenge**: how to achieving portable performance?

LIFT: Performance Portable GPU Code Generation via Rewrite Rules

[ICFP 2015]

[GPGPU 2016]
[CASES 2016]

[CGO 2017]



Ambition: automatic generation of *Performance Portable* code

Exploration in Numbers for Matrix Multiplication

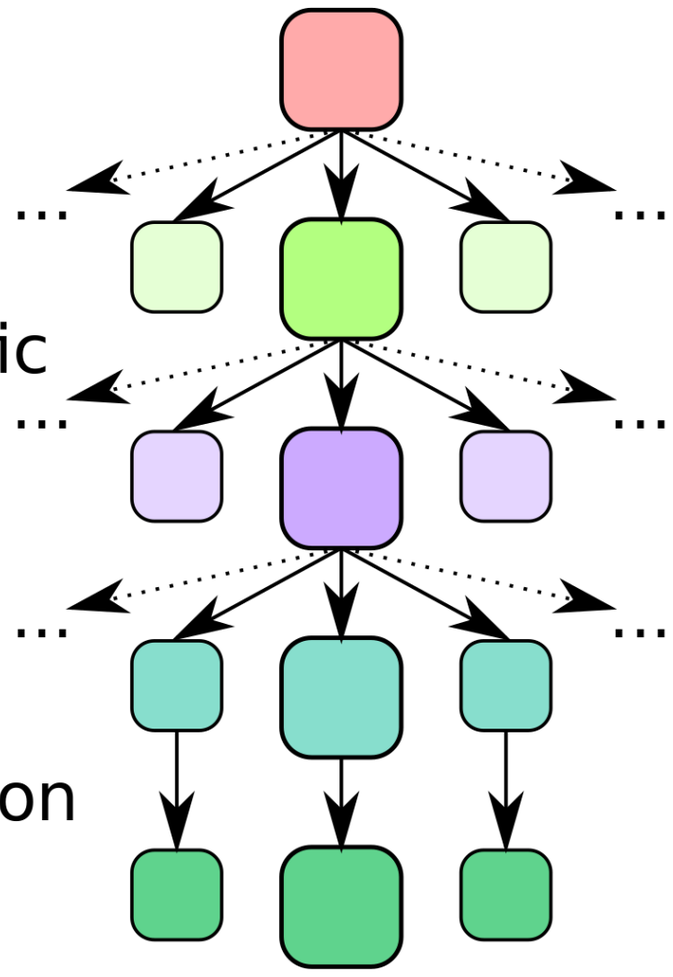
Phases:

Algorithmic Exploration

OpenCL specific Exploration

Parameter Exploration

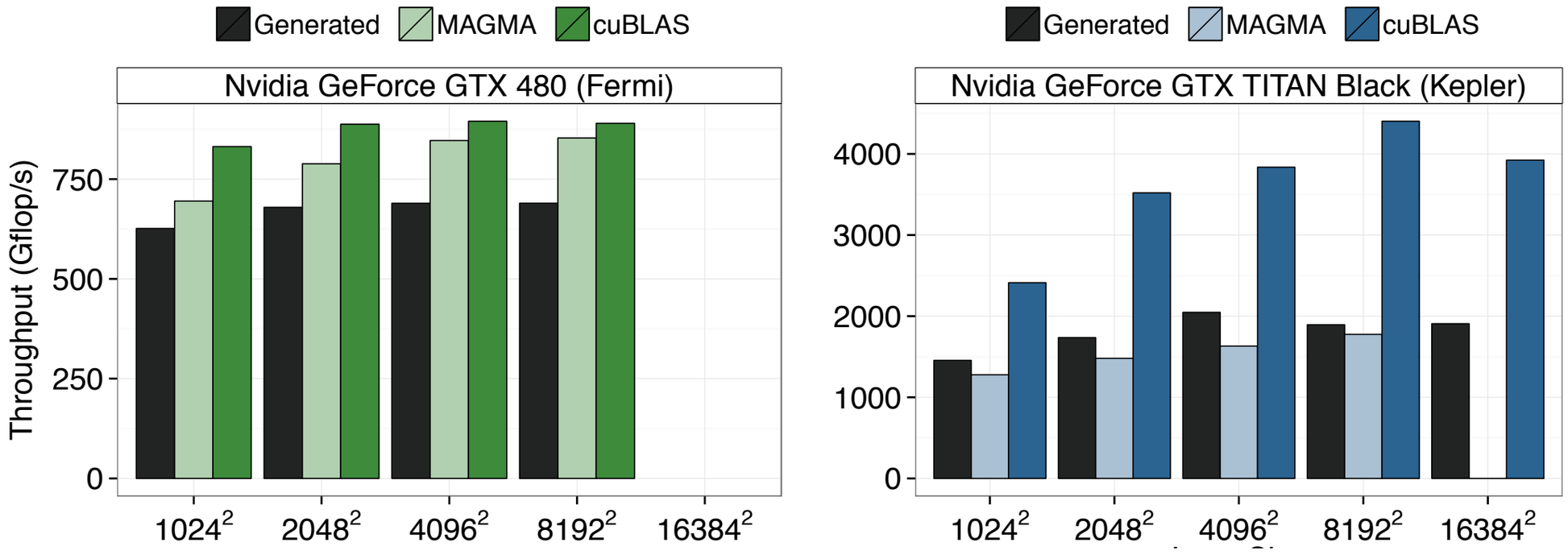
Code Generation

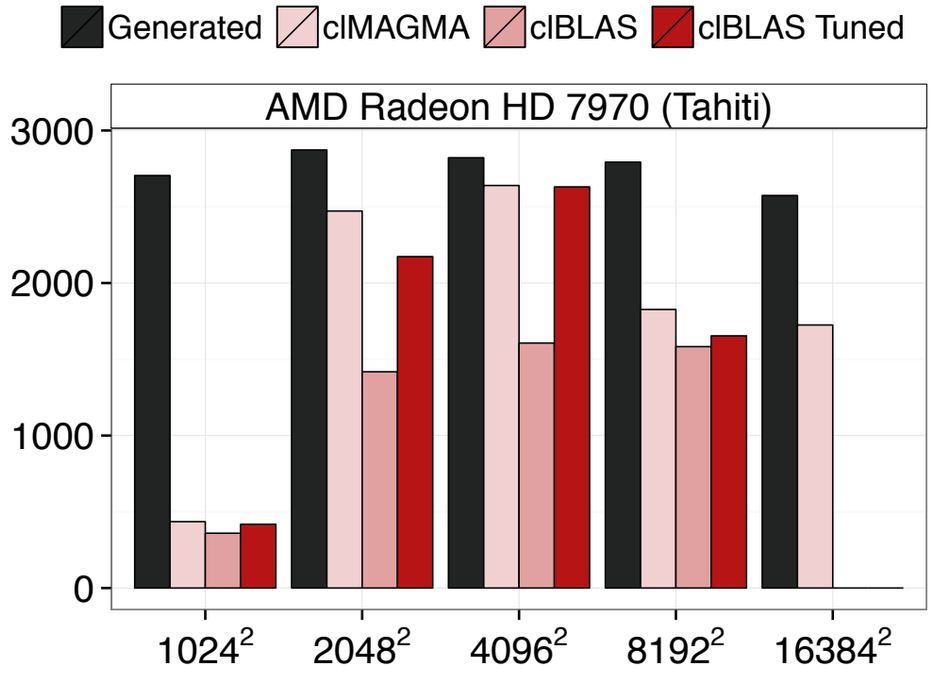


Program Variants:

High-Level Program	1
Algorithmic Rewritten Program	8
OpenCL Specific Program	760
Fully Specialized Program	46,000
OpenCL Code	46,000

Performance Results Matrix Multiplication





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 }
```

```
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 }
```

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

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

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

```
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 }
```

```
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 }
```

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

```
1  for (int i = 0; i<M/2; i++) {
2    for (int l = 0; l<2; l++) {
3      for (int j = 0; j<N; j++) {
4        for (int k = 0; k<K; k++) {
5          temp[k + 2*K*N*i + K*N*l + K*j] =
6            mult(A[k + K*l + 2*K*i], B[k + K*j]);
7        }
8        for (int k = 0;k<K;k++) {
9          C[j + N*l + 2*N*i] +=
10           temp[k + 2*K*N*i + K*N*l + K*j];
11       }
12     }
13   }
14 }
```

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

```
1  for (int i = 0; i<M/2; i++) {
2    for (int l = 0; l<2; l++) {
3      for (int j = 0; j<N; j++) {
4        for (int k = 0; k<K; k++) {
5          temp[k + 2*K*N*i + K*N*l + K*j] =
6            mult(A[k + K*l + 2*K*i], B[k + K*j]);
7        }
8        for (int k = 0;k<K;k++) {
9          C[j + N*l + 2*N*i] +=
10           temp[k + 2*K*N*i + K*N*l + K*j];
11       }
12     }
13   }
14 }
```

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

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

```
1  for (int i = 0; i<M/2; i++) {
2    for (int l = 0; l<2; l++) {
3      for (int j = 0; j<N; j++) {
4        for (int k = 0; k<K; k++) {
5          temp[k + 2*K*N*i + K*N*l + K*j] =
6            mult(A[k + K*l + 2*K*i], B[k + K*j]);
7        }
8        for (int k = 0;k<K;k++) {
9          C[j + N*l + 2*N*i] +=
10           temp[k + 2*K*N*i + K*N*l + K*j];
11       }
12     }
13   }
14 }
```

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

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

```
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
```

```
1  for (int i = 0; i<M/2; i++) {
2    for (int l = 0; l<2; l++) {
3      for (int j = 0; j<N; j++) {
4        for (int k = 0; k<K; k++) {
5          temp[k + 2*K*N*i + K*N*l + K*j] =
6            mult(A[k + K*l + 2*K*i], B[k + K*j]);
7        }
8        for (int k = 0;k<K;k++) {
9          C[j + N*l + 2*N*i] +=
10           temp[k + 2*K*N*i + K*N*l + K*j];
11       }
12     }
13   }
14 }
```

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

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

```
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
```

```
1  for (int i = 0; i<M/2; i++) {
2    for (int l = 0; l<2; l++) {
3      for (int j = 0; j<N; j++) {
4        for (int k = 0; k<K; k++) {
5          temp[k + 2*K*N*i + K*N*l + K*j] =
6            mult(A[k + K*l + 2*K*i], B[k + K*j]);
7        }
8        for (int k = 0;k<K;k++) {
9          C[j + N*l + 2*N*i] +=
10           temp[k + 2*K*N*i + K*N*l + K*j];
11       }
12     }
13   }
14 }
```

```
1  for (int i = 0; i<M/2; i++) {
2    for (int j = 0; j<N; j++) {
3      for (int l = 0; l<2; l++) {
4        for (int k = 0; k<K; k++) {
5          temp[k + 2*K*N*i + K*N*l + K*j] =
6            mult(A[k + K*l + 2*K*i], B[k + K*j]);
7        }
8        for (int k = 0;k<K;k++) {
9          C[j + N*l + 2*N*i] +=
10           temp[k + 2*K*N*i + K*N*l + K*j];
11       }
12     }
13   }
14 }
```

```
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
```

```
1  for (int i = 0; i<M/2; i++) {
2    for (int j = 0; j<N; j++) {
3      for (int l = 0; l<2; l++) {
4        for (int k = 0; k<K; k++) {
5          temp[k + 2*K*N*i + K*N*l + K*j] =
6            mult(A[k + K*l + 2*K*i], B[k + K*j]);
7        }
8        for (int k = 0;k<K;k++) {
9          C[j + N*l + 2*N*i] +=
10           temp[k + 2*K*N*i + K*N*l + K*j];
11       }
12     }
13   }
14 }
```

```
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
```

```
1  for (int i = 0; i<M/2; i++) {
2    for (int j = 0; j<N; j++) {
3      for (int l = 0; l<2; l++) {
4        for (int k = 0; k<K; k++) {
5          temp[k + 2*K*N*i + K*N*l + K*j] =
6            mult(A[k + K*l + 2*K*i], B[k + K*j]);
7        }
8        for (int k = 0;k<K;k++) {
9          C[j + N*l + 2*N*i] +=
10           temp[k + 2*K*N*i + K*N*l + K*j];
11       }
12     }
13   }
14 }
```

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

```
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
```

```
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
```

```
1  for (int i = 0; i<M/2; i++) {
2    for (int j = 0; j<N; j++) {
3      for (int l = 0; l<2; l++) {
4        for (int k = 0; k<K; k++) {
5          temp[k + 2*K*N*i + K*N*l + K*j] =
6            mult(A[k + K*l + 2*K*i], B[k + K*j]);
7        }
8        for (int k = 0;k<K;k++) {
9          C[j + N*l + 2*N*i] +=
10           temp[k + 2*K*N*i + K*N*l + K*j];
11       }
12     }
13   }
14 }
```

```
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
```

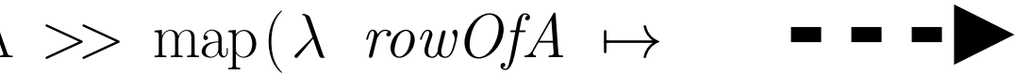
```
map(f) ⟹ split(k) >> map(map(f)) >> join
```

```
A >> split(m) >> map(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
```

```
1  for (int i = 0; i<M/2; i++) {
2    for (int j = 0; j<N; j++) {
3      for (int l = 0; l<2; l++) {
4        for (int k = 0; k<K; k++) {
5          temp[k + 2*K*N*i + K*N*l + K*j] =
6            mult(A[k + K*l + 2*K*i], B[k + K*j]);
7        }
8        for (int k = 0;k<K;k++) {
9          C[j + N*l + 2*N*i] +=
10           temp[k + 2*K*N*i + K*N*l + K*j];
11       }
12     }
13   }
14 }
```

```
1  for (int i = 0; i<M/2; i++) {
2    for (int j = 0; j<N/2; j++) {
3      for (int m = 0; m<2; m++) {
4        for (int l = 0; l<2; l++) {
5          for (int k = 0; k<K; k++) {
6            temp[k + 4*K*N*i + 2*K*N*l + 2*K*j
                 + K*m] =
7              mult(A[k + K*l + 2*K*i], B[k + K*
                 m + 2*K*j]);
8          }
9          for (int k = 0;k<K;k++) {
10           C[m + 2*j + 2*N*l + 4*N*i] +=
11             temp[k + 4*K*N*i + 2*K*N*l + 2*
                 K*j + K*m];
12         }
13       }
14     }
15   }
16 }
```

```
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
```



```
1  for (int i = 0; i<M/2; i++) {
2    for (int j = 0; j<N/2; j++) {
3      for (int m = 0; m<2; m++) {
4        for (int l = 0; l<2; l++) {
5          for (int k = 0; k<K; k++) {
6            temp[k + 4*K*N*i + 2*K*N*l + 2*K*j
                 + K*m] =
7              mult(A[k + K*l + 2*K*i], B[k + K*
                 m + 2*K*j]);
8          }
9          for (int k = 0;k<K;k++) {
10           C[m + 2*j + 2*N*l + 4*N*i] +=
11             temp[k + 4*K*N*i + 2*K*N*l + 2*
                 K*j + K*m];
12         }
13       }
14     }
15   }
16 }
```

```
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
```

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

```
1  for (int i = 0; i<M/2; i++) {
2    for (int j = 0; j<N/2; j++) {
3      for (int m = 0; m<2; m++) {
4        for (int l = 0; l<2; l++) {
5          for (int k = 0; k<K; k++) {
6            temp[k + 4*K*N*i + 2*K*N*l + 2*K*j
                 + K*m] =
7              mult(A[k + K*l + 2*K*i], B[k + K*
                 m + 2*K*j]);
8          }
9          for (int k = 0;k<K;k++) {
10           C[m + 2*j + 2*N*l + 4*N*i] +=
11             temp[k + 4*K*N*i + 2*K*N*l + 2*
                 K*j + K*m];
12         }
13       }
14     }
15   }
16 }
```

```
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
```

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

```
A >> split(m) >> map(λ rowsOfA ↦
 B >> 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
```

```
1  for (int i = 0; i<M/2; i++) {
2    for (int j = 0; j<N/2; j++) {
3      for (int m = 0; m<2; m++) {
4        for (int l = 0; l<2; l++) {
5          for (int k = 0; k<K; k++) {
6            temp[k + 4*K*N*i + 2*K*N*l + 2*K*j
                 + K*m] =
7              mult(A[k + K*l + 2*K*i], B[k + K*
                 m + 2*K*j]);
8          }
9          for (int k = 0;k<K;k++) {
10           C[m + 2*j + 2*N*l + 4*N*i] +=
11             temp[k + 4*K*N*i + 2*K*N*l + 2*
                 K*j + K*m];
12         }
13       }
14     }
15   }
16 }
```

```
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
```

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

```
A >> split(m) >> map(λ rowsOfA ↦
 B >> 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
```

```
1  for (int i = 0; i<M/2; i++) {
2    for (int j = 0; j<N/2; j++) {
3      for (int m = 0; m<2; m++) {
4        for (int l = 0; l<2; l++) {
5          for (int k = 0; k<K; k++) {
6            temp[k + 4*K*N*i + 2*K*N*l + 2*K*j
                 + K*m] =
7              mult(A[k + K*l + 2*K*i], B[k + K*
                 m + 2*K*j]);
8          }
9          for (int k = 0;k<K;k++) {
10           C[m + 2*j + 2*N*l + 4*N*i] +=
11             temp[k + 4*K*N*i + 2*K*N*l + 2*
                 K*j + K*m];
12         }
13       }
14     }
15   }
16 }
```

```
1  for (int i = 0; i<M/2; i++) {
2    for (int j = 0; j<N/2; j++) {
3      for (int l = 0; l<2; l++) {
4        for (int m = 0; m<2; m++) {
5          for (int k = 0; k<K; k++) {
6            temp[k + 4*K*N*i + 2*K*N*l + 2*K*j
                 + K*m] =
7              mult(A[k + K*l + 2*K*i], B[k + K*
                 m + 2*K*j]);
8          }
9          for (int k = 0;k<K;k++) {
10           C[m + 2*j + 2*N*l + 4*N*i] +=
11             temp[k + 4*K*N*i + 2*K*N*l + 2*
                 K*j + K*m];
12         }
13       }
14     }
15   }
16 }
```

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
```

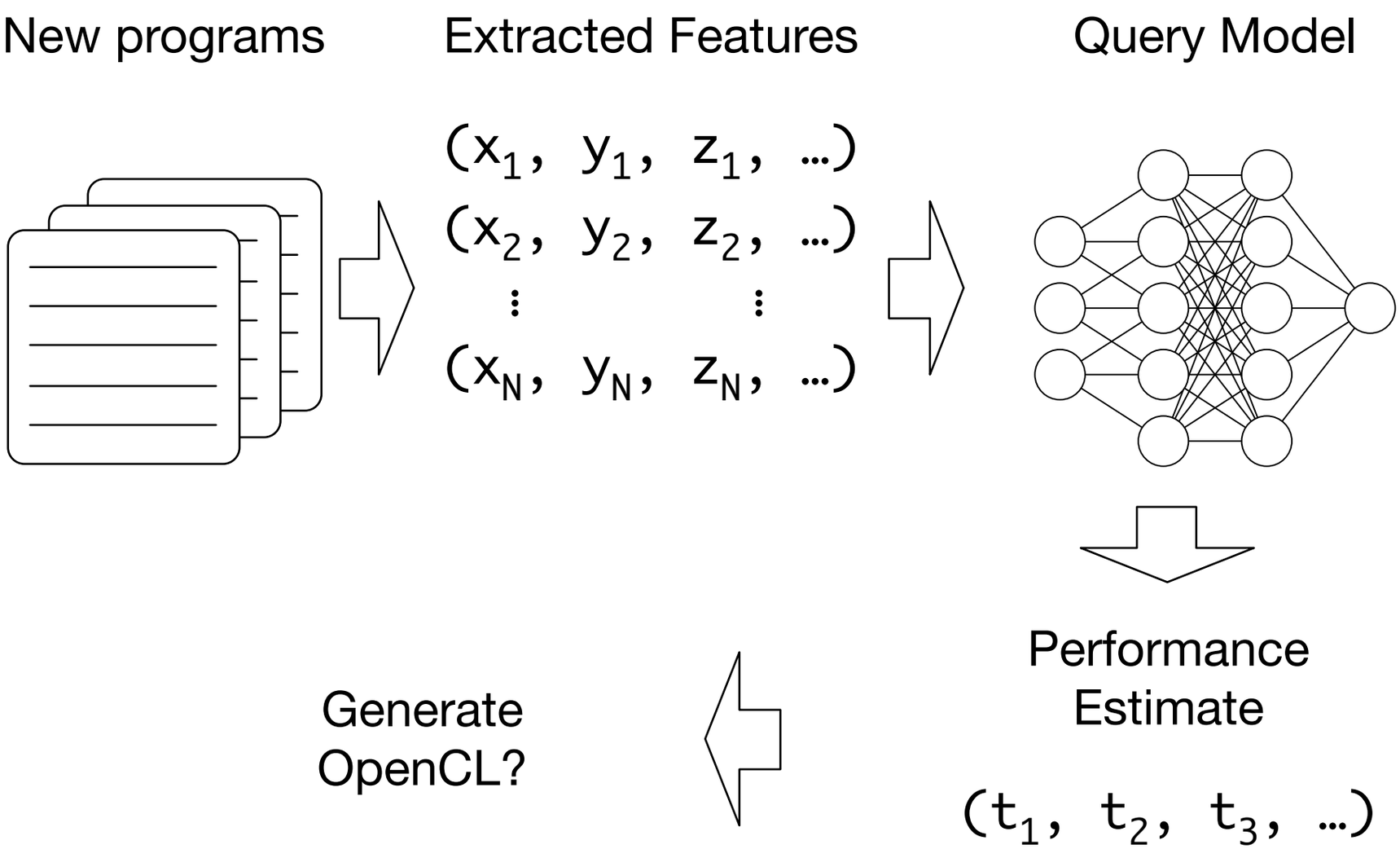
```
1  for (int i = 0;i<M/2; i++) {
2    for (int j = 0;j<N/2; j++) {
3      for (int k = 0;k<K/4; k++) {
4        for (int l = 0;l<2; l++) {
5          for (int m = 0;m<2; m++) {
6            for (int n = 0;n<4; n++) {
7              temp[n + 4*m + 8*N*i + 16*j + 8*l] =
8                mult(
9                  A[n + 2*K*i + 4*k + K*l],
10                 B[n + 2*K*j + 4*k + K*m]
11               );
12           }
13           for (int n = 0;n<4; n++) {
14             C[m + 2*N*i + 2*j + N*l] +=
15               temp[n + 4*m + 8*N*i + 16*j + 8*l];
16           }
17         }
18       }
19     }
20   }
21 }
```

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



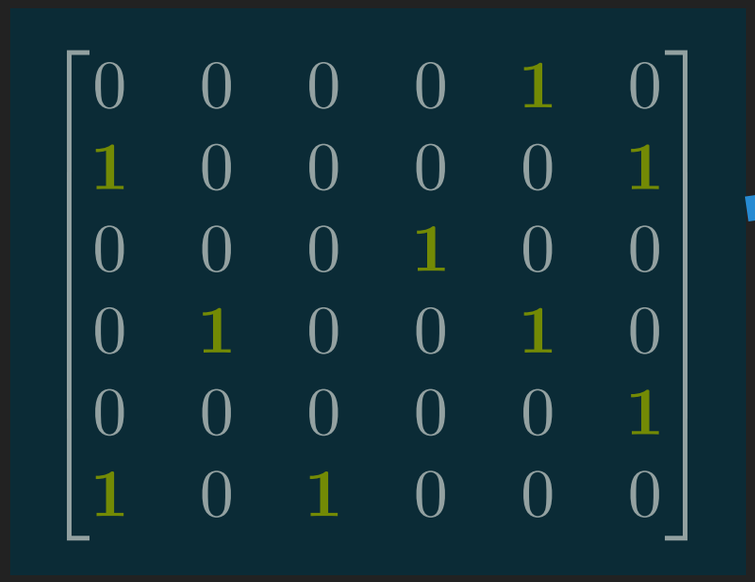
ADAM HARRIES

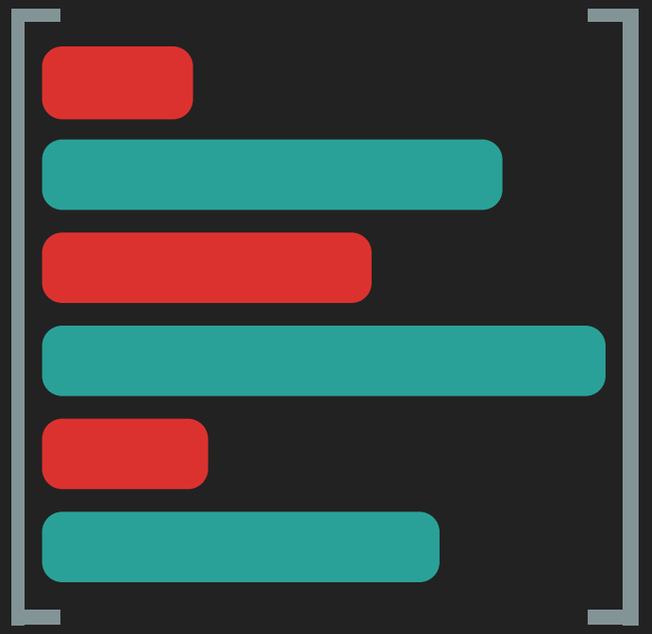
SPARSITY AND IRREGULARITY

+ FUTURE WORK

SPARSE DATA STRUCTURES IN LIFT

Research: Adding data structure primitives to support sparse data layouts



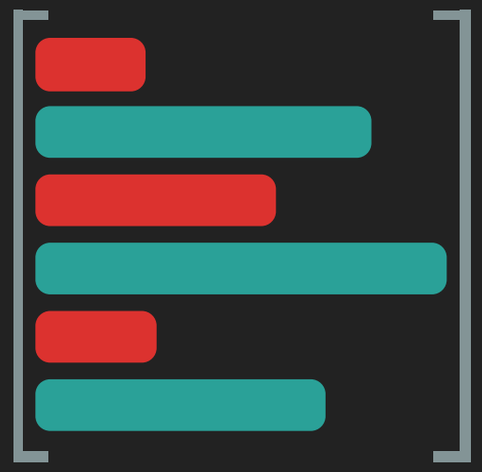


- Extend Lift language with notion of `Dynamic` arrays
- Composable array structure with low level space optimisations

```
StaticArray(
  DynamicArray((Int, T))
,M)
```

LOAD BALANCING PRIMITIVES

Research: introducing dynamic work assignment in a compositional manner

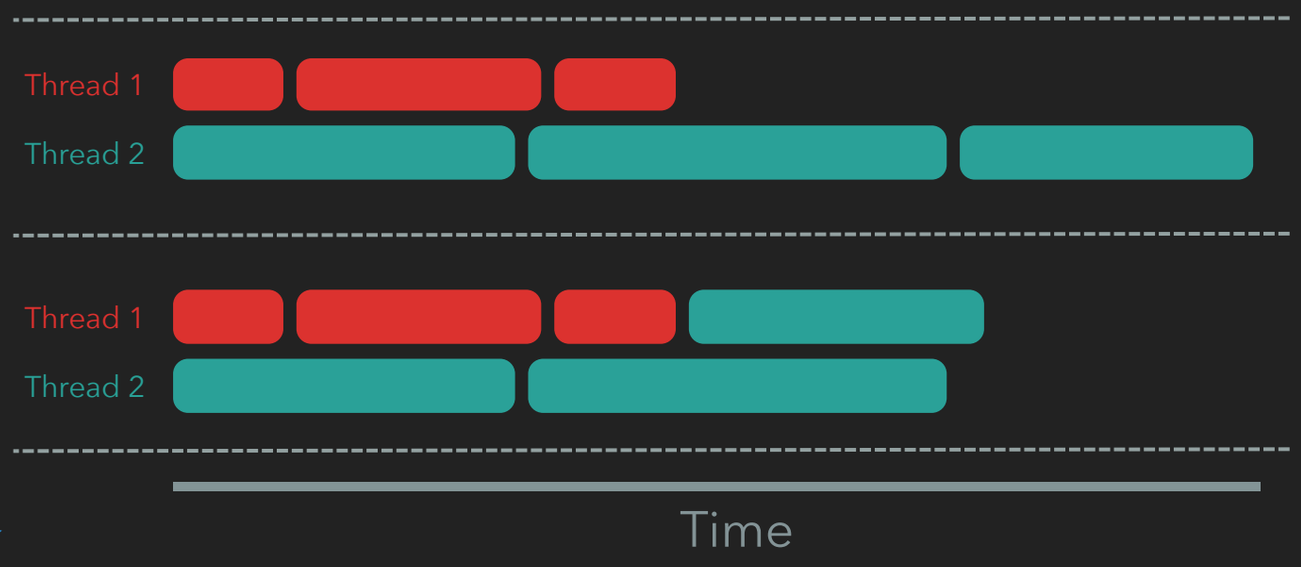


- Extending set of primitives in lift with dynamic load balancing variants
- Maintaining composition while mitigating irregularity

```
Map(
  λ(row → …),
  matrix
)
```

MapStatic

MapDynamic



FUTURE WORK

Research: Lifting Lift out of the GPU

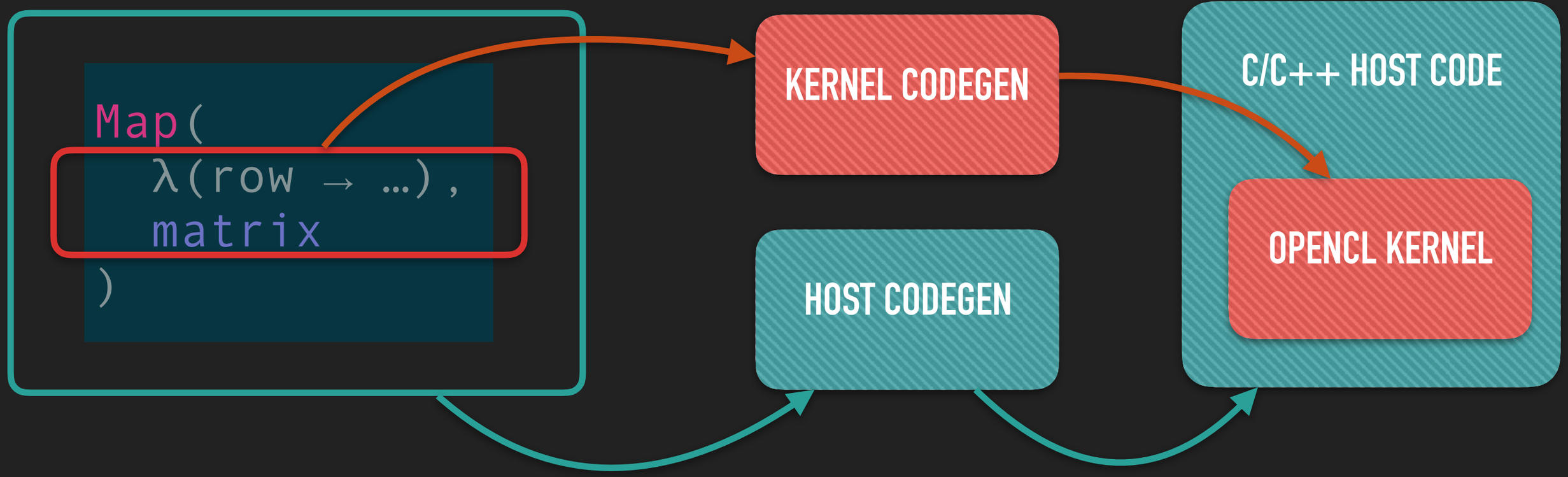
```
Map(
  λ(row → …),
  matrix
)
```

KERNEL CODEGEN

OPENCL KERNEL

FUTURE WORK

Research: Lifting Lift out of the GPU



Parallelizing non-associative sequential reductions

By Federico Pizzuti
Supervisor: Cristophe Dubach

Associative reductions are parallelizable

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

```
reduce(+,[1,2,3,4,5,6,7,8])
```

The operator + is **associative**, so this reduction is **parallelizable**

What about non-associative reductions?

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

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

```
reduce(⊙,[1,2,3,4,5,6,7,8])
```

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

How to parallelize non-associative reductions?

Key insight: Matrix multiplication ($\times$) is associative*

We rewrite the reduction using $\odot$ 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 ·

Rewrite reduction operator as matrix multiplication

Rearrange data as matrices

`[1,2,3,4,5,6,7,8]`

$$\begin{bmatrix} k & 1 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} k & 2 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} k & 3 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} k & 4 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} k & 5 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} k & 6 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} k & 7 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} k & 8 \\ 0 & 1 \end{bmatrix}$$

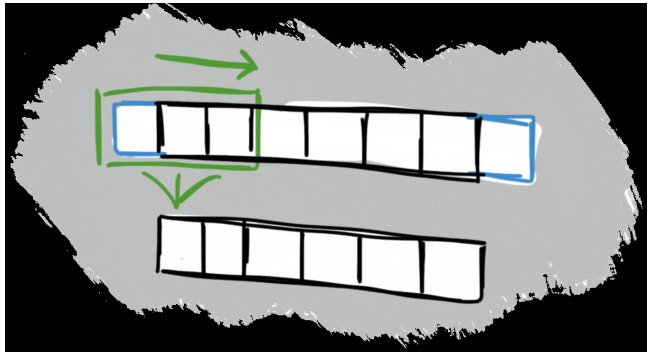
$$\texttt{reduce}(\odot, [1,2,3,4,5,6,7,8]) = x$$

$$\texttt{reduce}(\odot, \begin{bmatrix} k & 1 \\ 0 & 1 \end{bmatrix}, \ldots, \begin{bmatrix} k & 8 \\ 0 & 1 \end{bmatrix}) = \begin{bmatrix} a & x \\ 0 & 1 \end{bmatrix}$$

Decomposing Stencil Computations

stencil.c

```
for (int i = 0; i < N; i++) {
  int sum = 0;
  for (int j = -1; j <= 1; j++) {     // (a)
    int pos = i + j;
    pos = pos < 0 ? 0 : pos;          // (b)
    pos = pos > N - 1 ? N - 1 : pos;
    sum += A[pos]; }                  // (c)
  B[i] = sum;
}
```

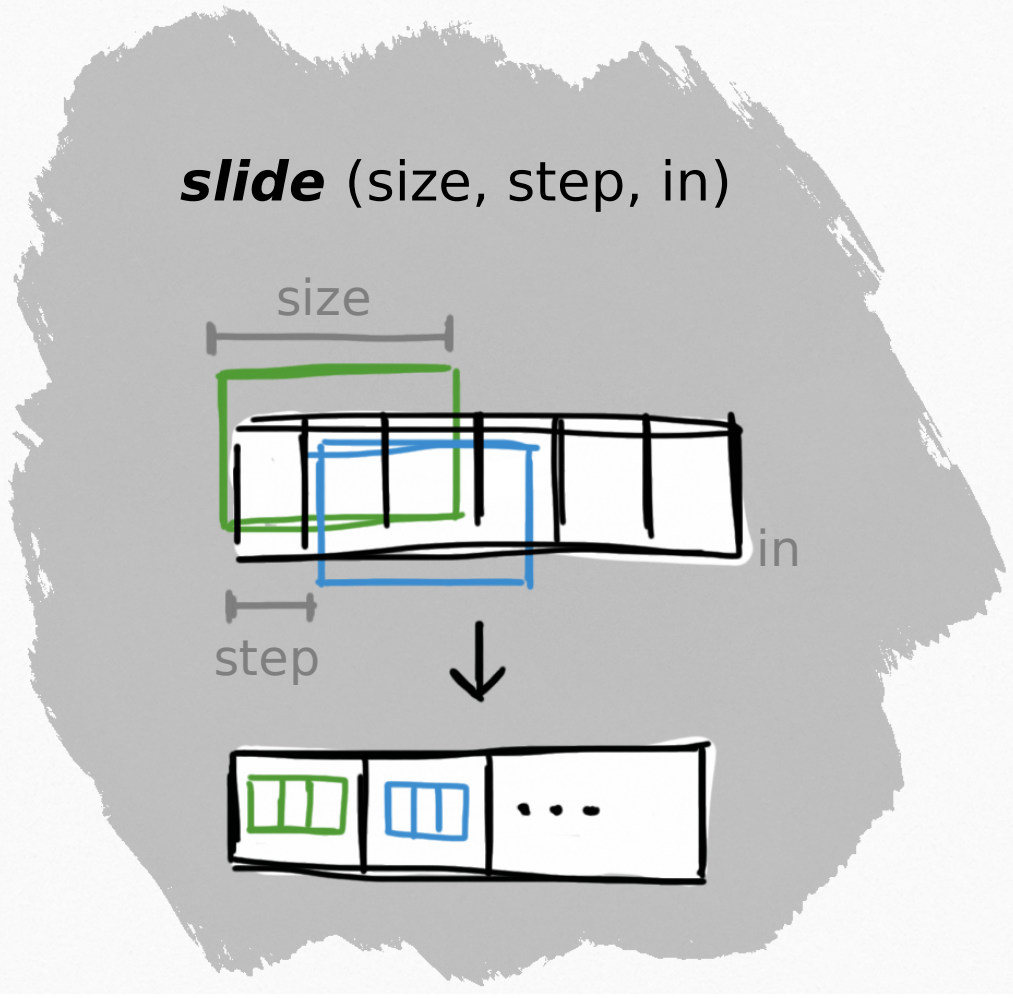


(a) Neighborhood: accessing neighboring elements according to stencil shape

(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

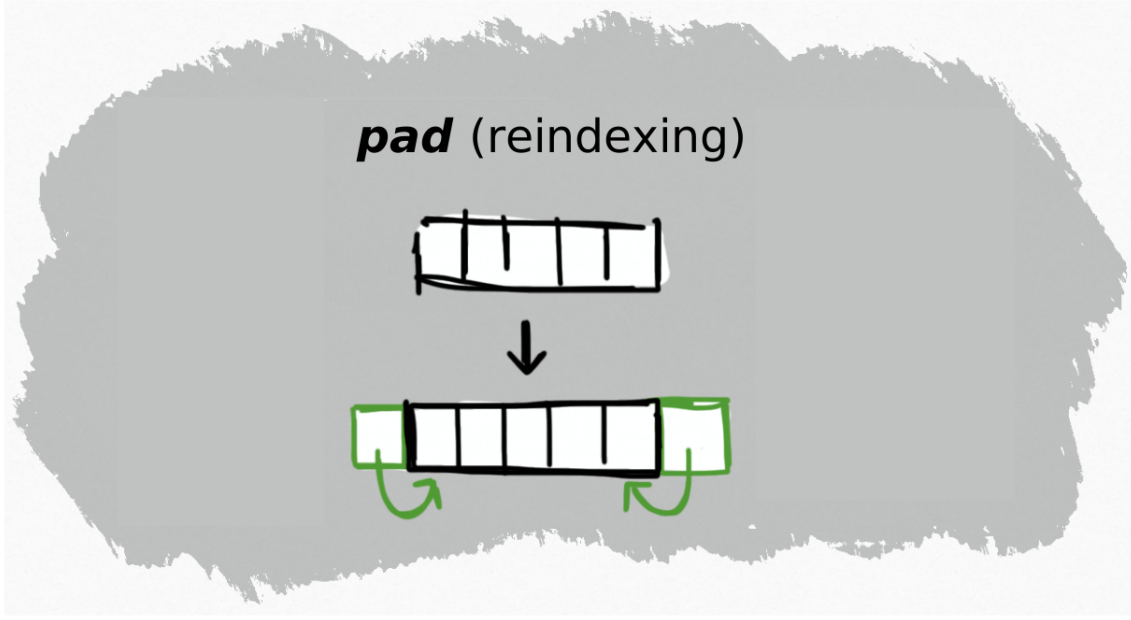


slide.example

$$\textbf{\textit{slide}}(3, 1, [a, b, c, d, e]) = \big[[a, b, c], [b, c, d], [c, d, e]\big]$$

$$\textbf{\textit{slide}} : (\mathit{size} : \texttt{Int},\ \mathit{step} : \texttt{Int},\ \mathit{in} : [T]_n) \to [[T]_{\mathit{size}}]_{\frac{n-\mathit{size}+\mathit{step}}{\mathit{step}}}$$

(b) Boundary Handling using Pad

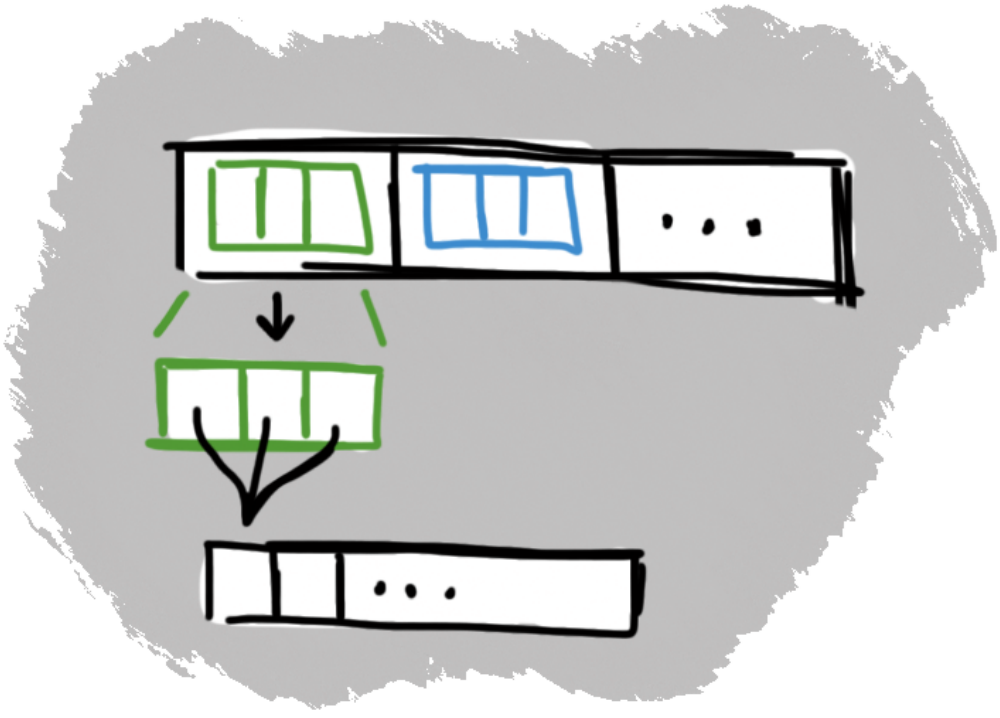


reindexing.example

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

$$\textbf{pad}\ (1, 1, clamp, [a, b, c, d]) = [a, a, b, c, d, d]$$

(c) Apply Stencil Function using Map



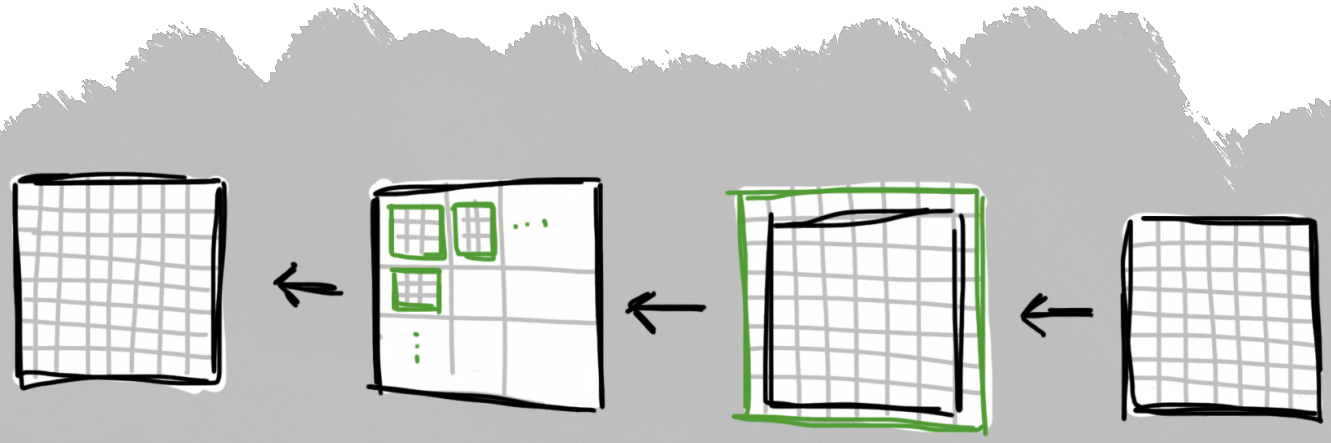
stencil-fun.example

$$\textbf{\textit{map}}(nbh \Rightarrow \textbf{\textit{reduce}}(\texttt{add, 0.0f}, nbh), [[0,1,2],[1,2,3]]) = [[3],[6]]$$

Multidimensional Stencil Computations

Idea: Express complex computations
as compositions of simple primitives

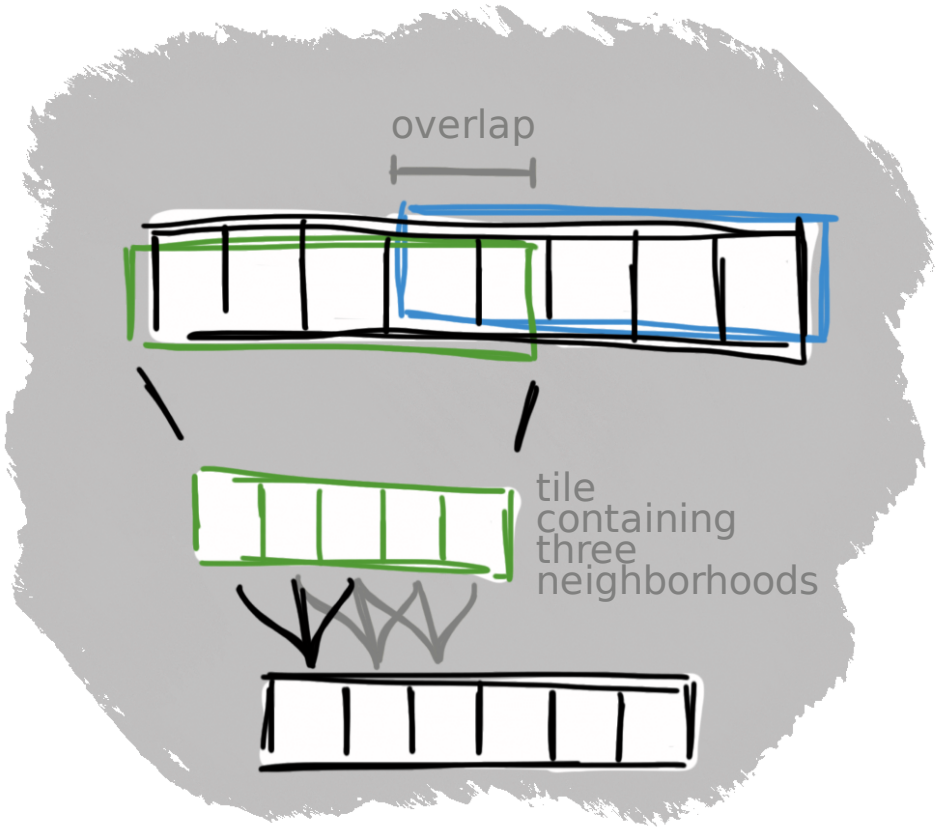
$$\textbf{map}_n(f,\ \textbf{slide}_n(size,\ step,\ \textbf{pad}_n(l,\ r,\ h,\ input)))$$



$$\textbf{map}_2\,(\text{f},\ \textbf{slide}_2\,(\text{size, step},\ \textbf{pad}_2\,(\text{l, r, h, in})))$$

Optimizing Stencil Computations

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.

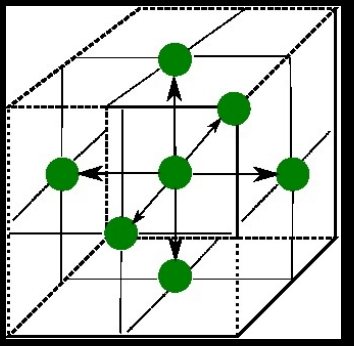
Overlap The shape of the stencil determines the overlap at the edges of tiles

3D Wave Simulations in Lift

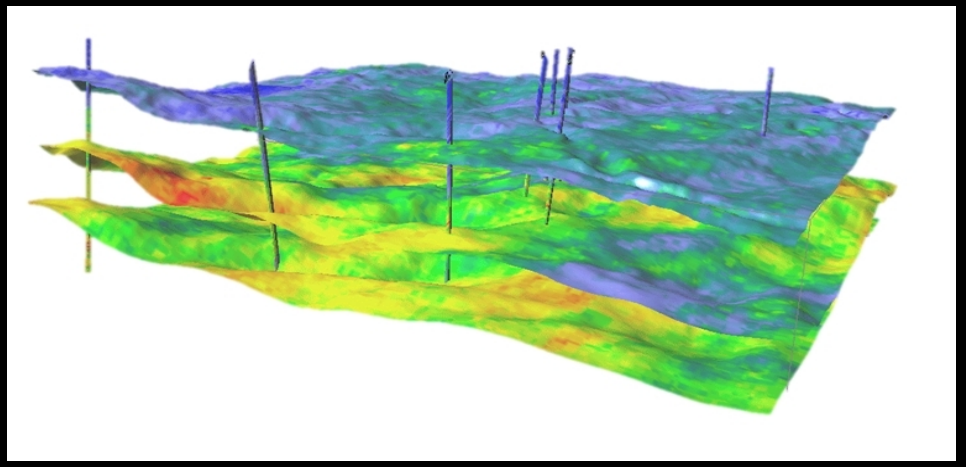
FDTD is a common approach to modelling the 3D wave equation

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

$$\psi^{n+1}_{x,y,z} = (2 - 6L^2)\psi^n_{x,y,z} + L^2(\psi^n_{x-1,y,z} + \psi^n_{x+1,y,z} + \psi^n_{x,y-1,z} + \psi^n_{x,y+1,z} + \psi^n_{x,y,z-1} + \psi^n_{x,y,z+1}) - \psi^{n-1}_{x,y,z}$$



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

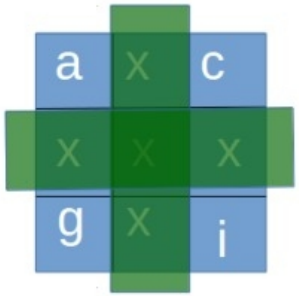


ground penetrating radar

room acoustics simulations

Implementing Basic Room Acoustics Simulations

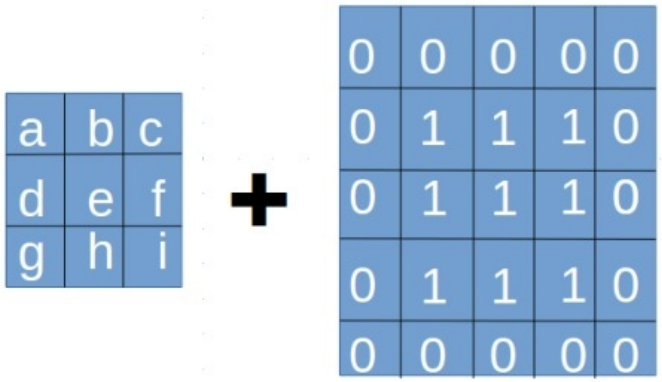
- The ***slide*** primitive makes more memory accesses than necessary for smaller stencils
- The ***pad*** primitive does not supply functionality for constant boundary conditions, only different index accesses
- Two new primitives were implemented to accommodate these shortcomings: ***select*** and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



Selection of stencil shape (*select*)

Targeting different groups of neighbouring values

LIFT: map(reduce(nbh[-1]+nbh[0]+nbh[1] ∘ slide(size,step)) « input
C: for(i = 0 → n) updated[i] = data[i-1]+data[i]+data[i+1]

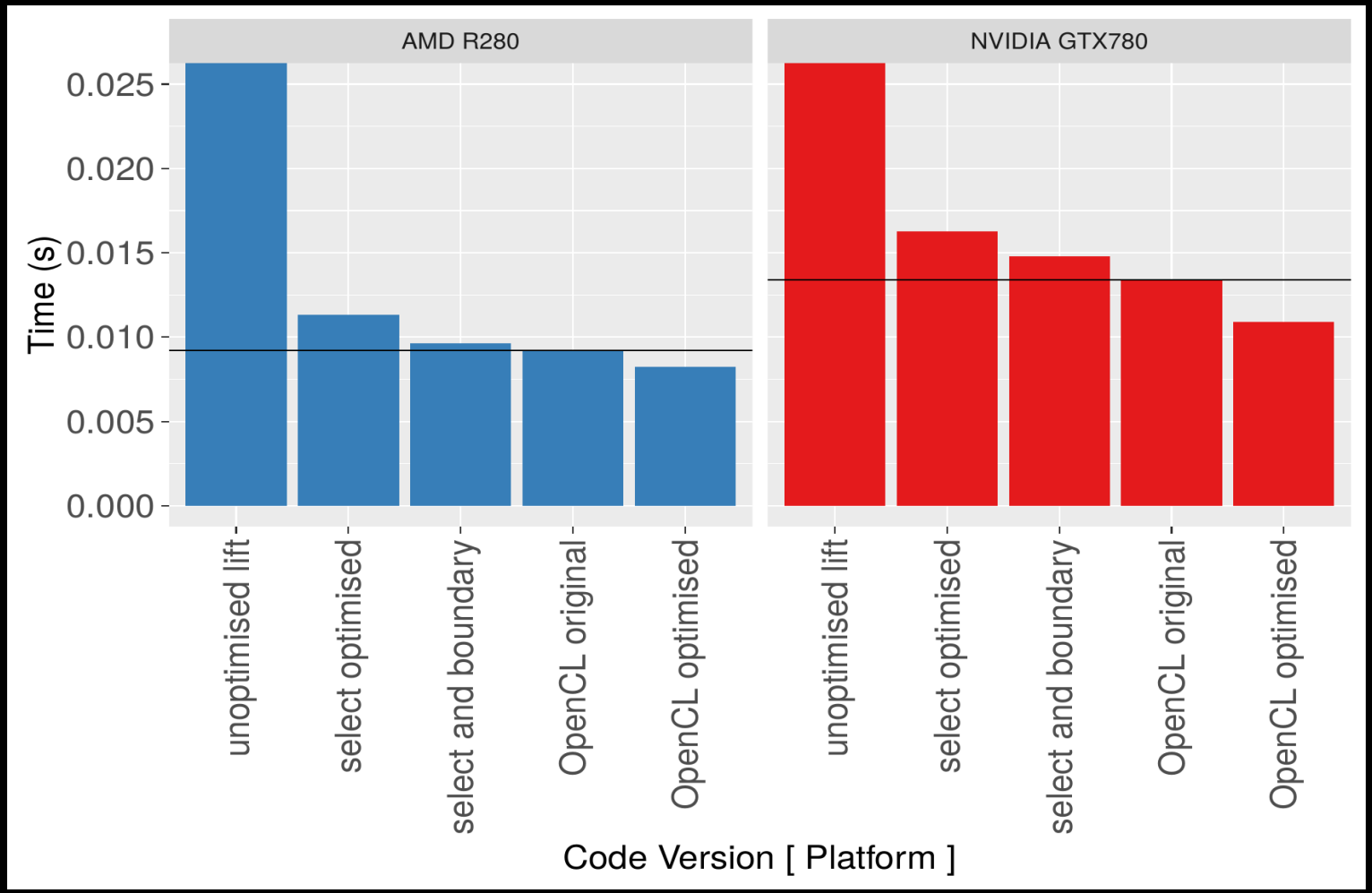


On-the-fly boundary handling (*boundary*)

Creating masking values when needed instead of storing in memory

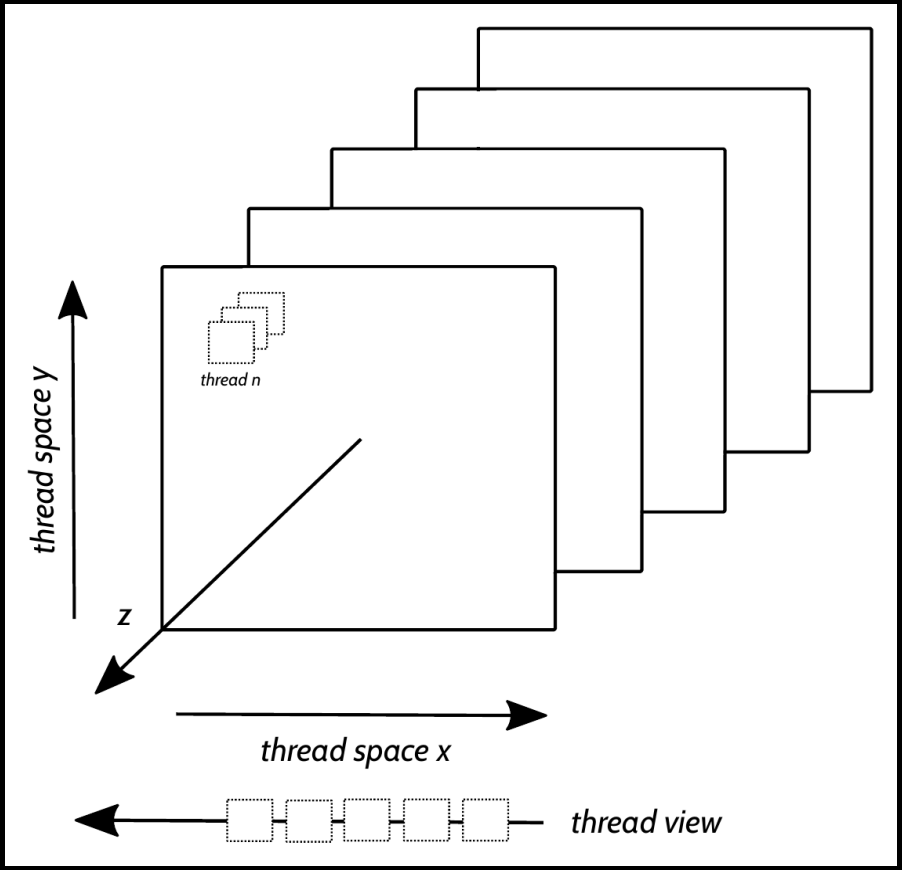
LIFT: boundaryValue = ArrayGen(idx)*border1 + !ArrayGen(idx)*border2
C: boundaryValue = if(idx <= M || idx > 0) return 1.0 else return 0.0

Results from Select and Boundary Optimisations



comparable results to original room acoustics benchmark, but still slower than same optimised benchmark

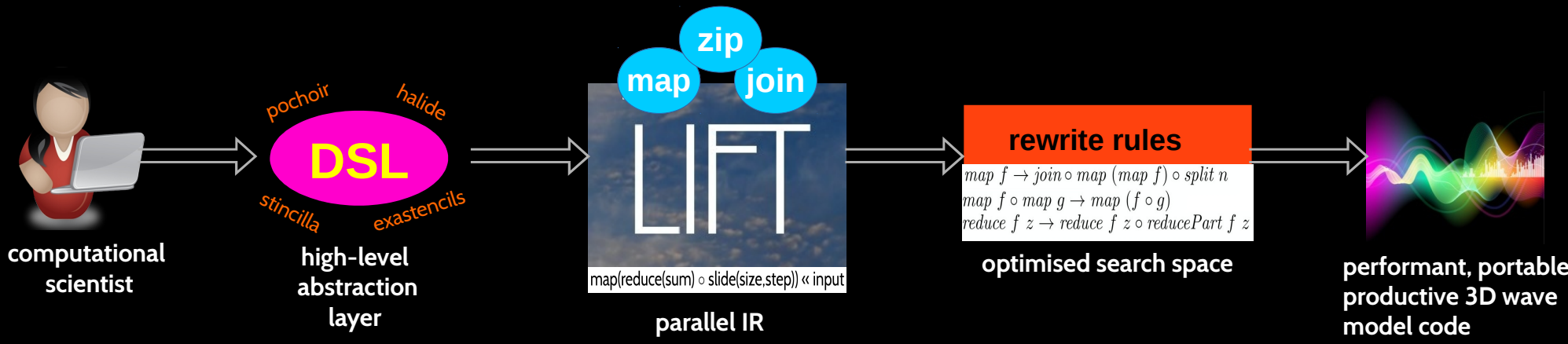
Optimising 3D Stencils



2.5D Tiling Algorithm

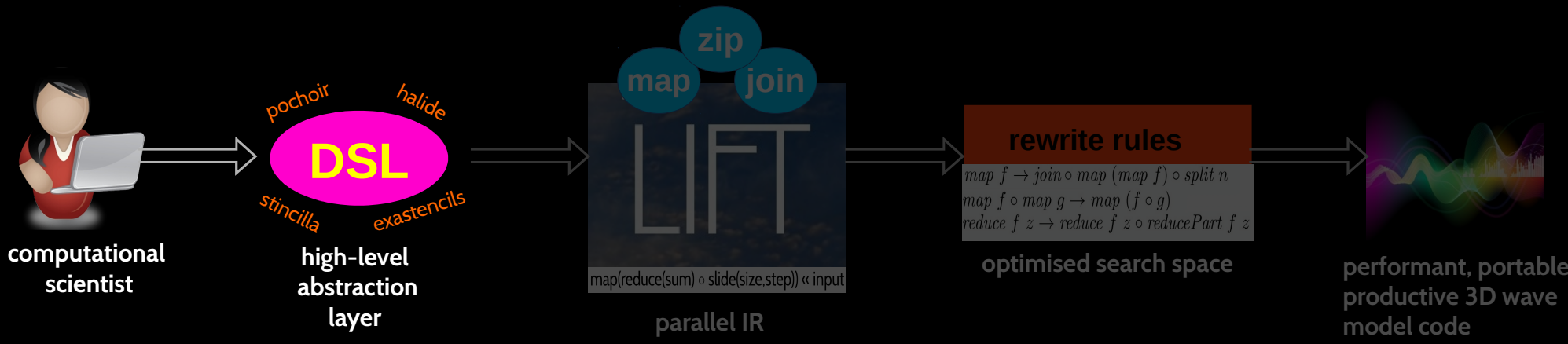
- To close the performance gap, implementing the commonly used 3D stencil optimisation “2.5D Tiling”
- Performs calculations in two dimensions in parallel and the third sequentially
- Performance improvements of up to 5-15% seen in original room acoustics simulations when used in conjunction with local memory

Long Term Goal



Develop a workflow for creating performant, portable, productive 3D wave models

Long Term Goal



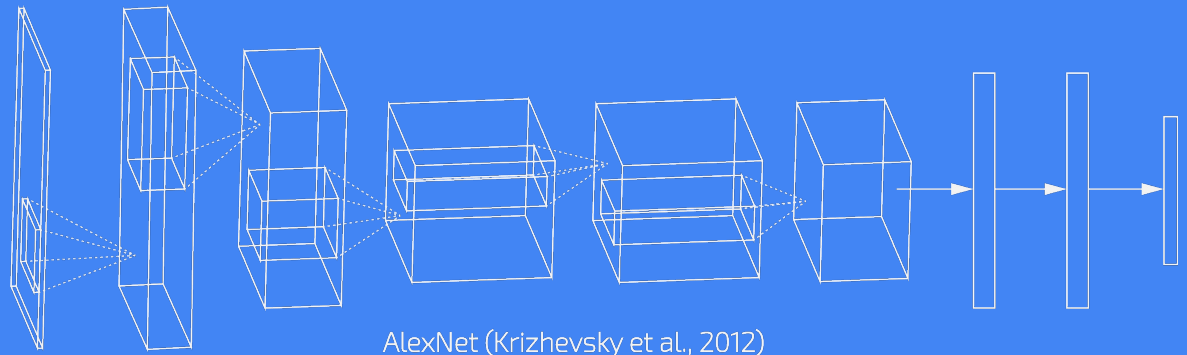
Develop a workflow for creating performant, portable, productive 3D wave models

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**

EPSRC Centre for Doctoral Training in
Pervasive Parallelism

Computational Efficiency Optimisation Of Convolutional Neural Networks Using A Functional Data-Parallel Language



AlexNet (Krizhevsky et al., 2012)

By **Naums Mogers**
Supervisor: Christophe Dubach

Problem

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

Approach

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

Approach

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

Current progress

1. Implementation of Convolutional and Fully Connected layers in Lift

Current progress

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

Current progress

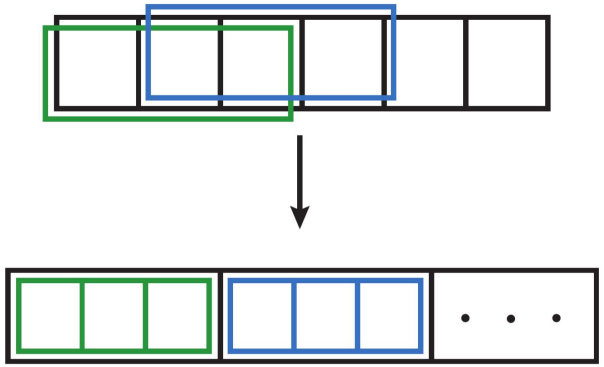
1. Implementation of Convolutional and Fully Connected layers in Lift

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

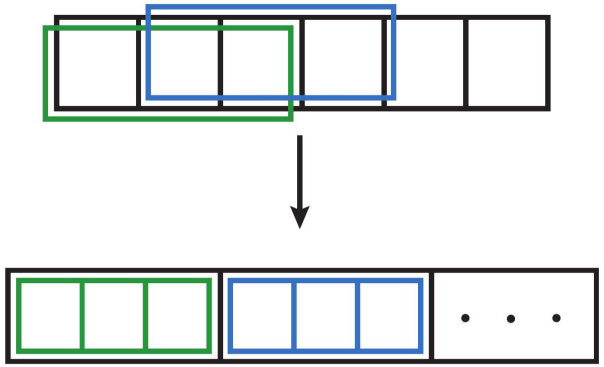
Sliding



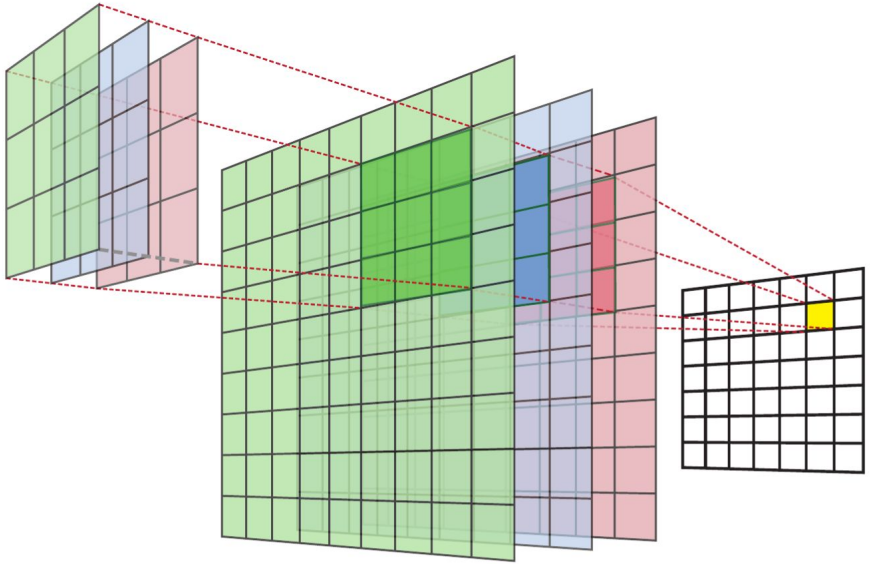
1D Slide example

Convolutional layer: the algorithm

Sliding → Weighted summation

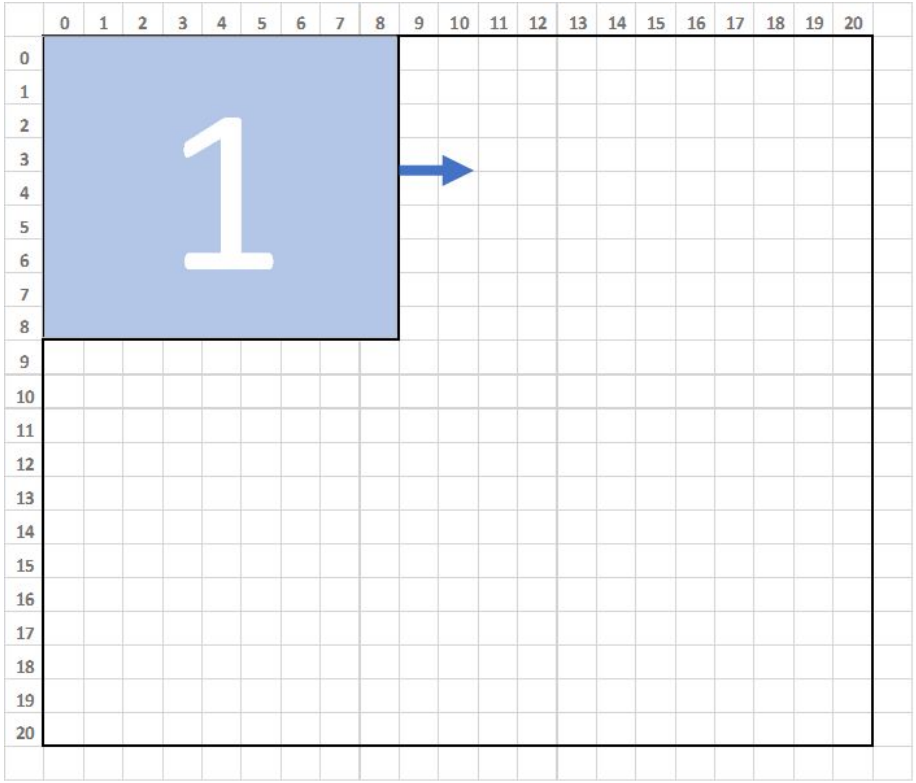


1D Slide example



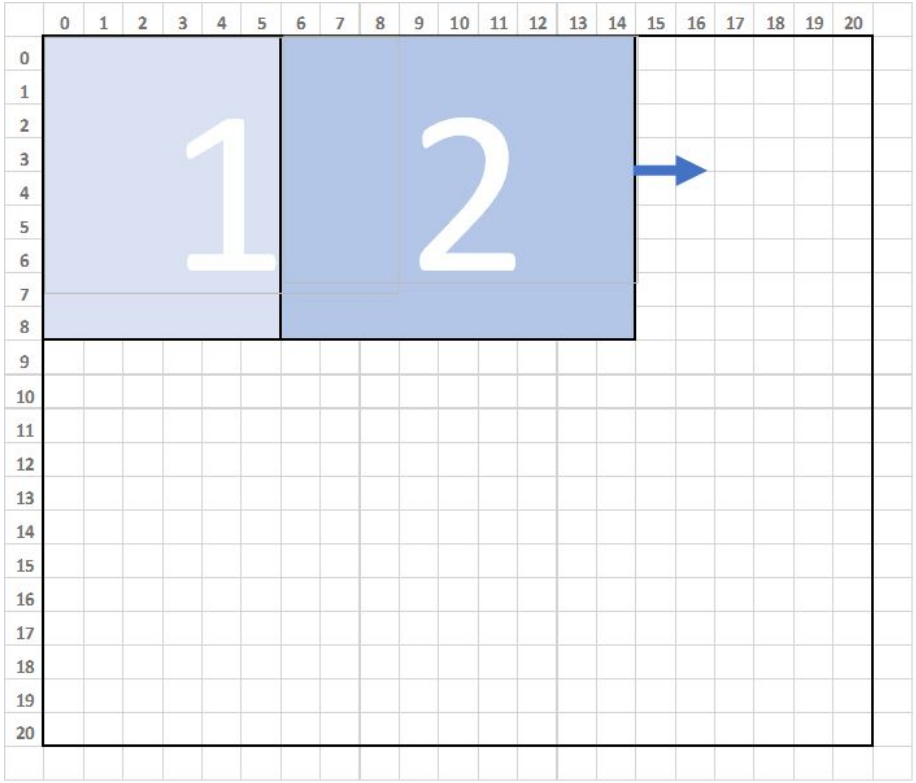
Convolutional layer: optimisation

- **Input tiling**



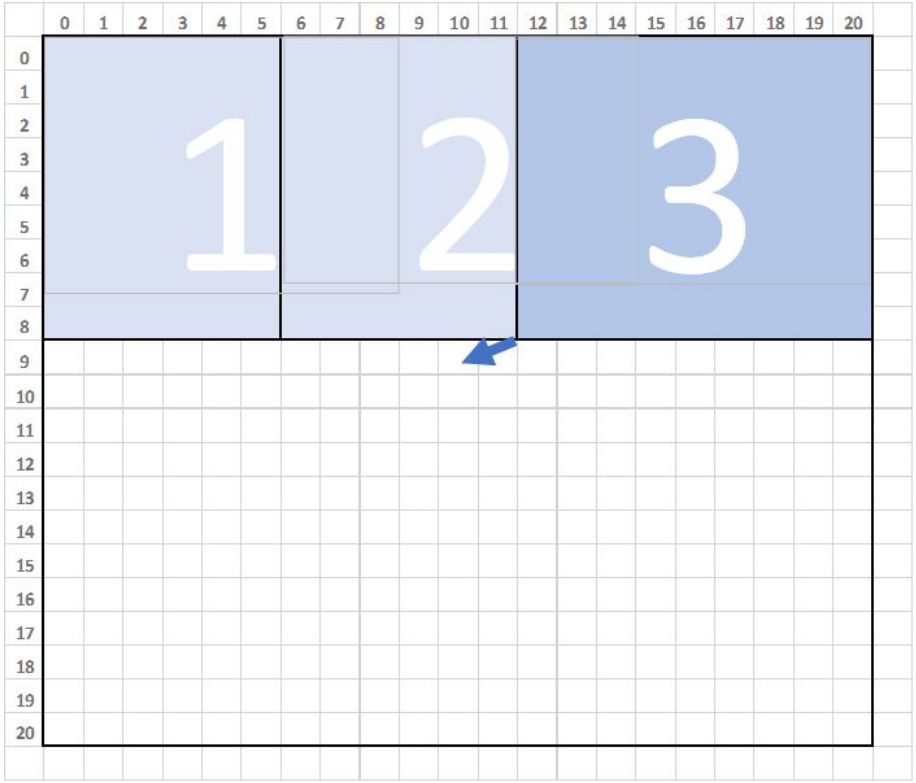
Convolutional layer: optimisation

- **Input tiling**



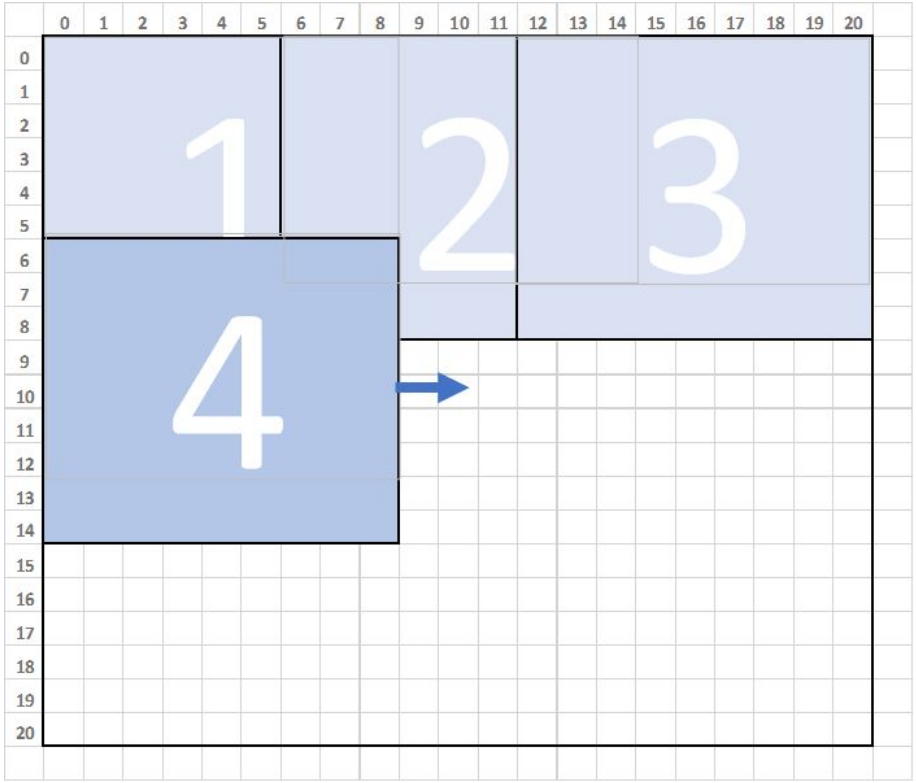
Convolutional layer: optimisation

- **Input tiling**



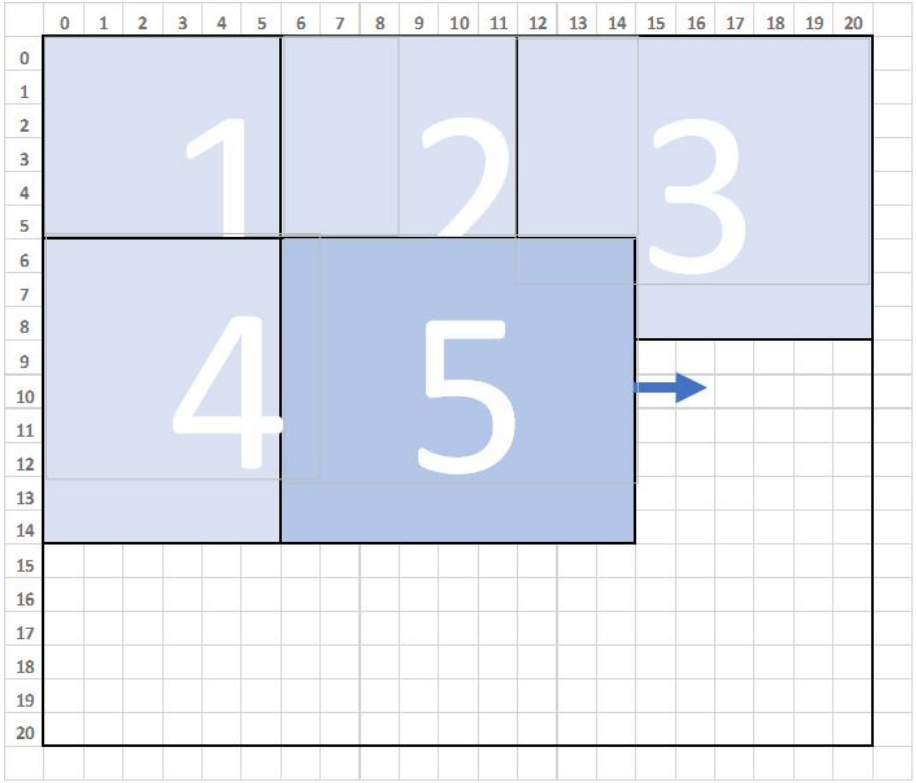
Convolutional layer: optimisation

- **Input tiling**



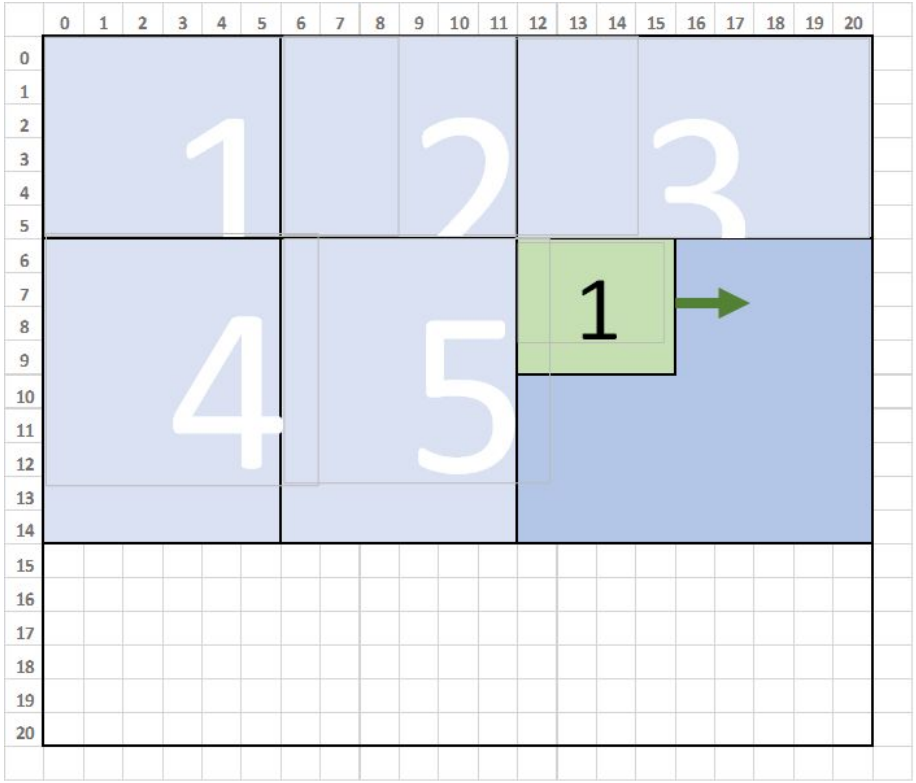
Convolutional layer: optimisation

- **Input tiling**



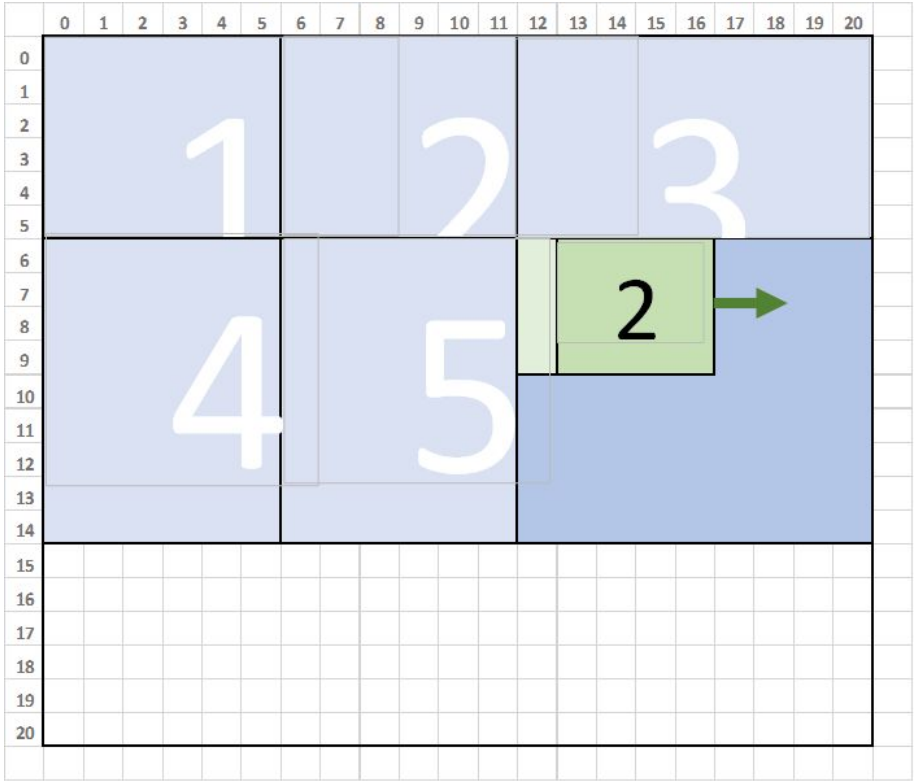
Convolutional layer: optimisation

- Input tiling
- Tiled sliding



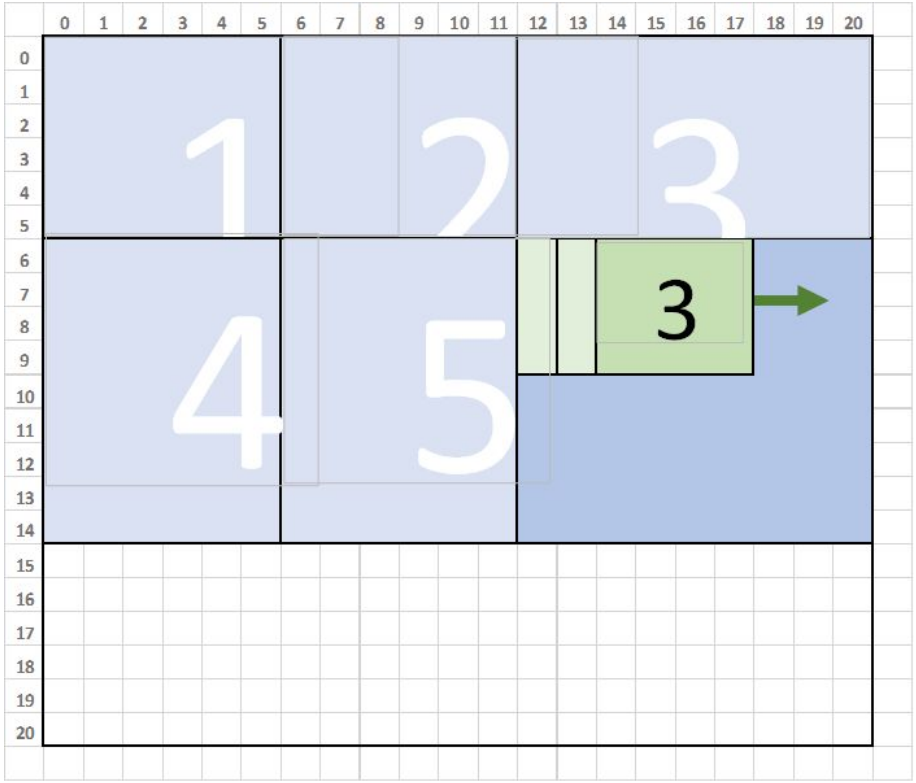
Convolutional layer: optimisation

- Input tiling
- Tiled sliding



Convolutional layer: optimisation

- Input tiling
- Tiled sliding



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

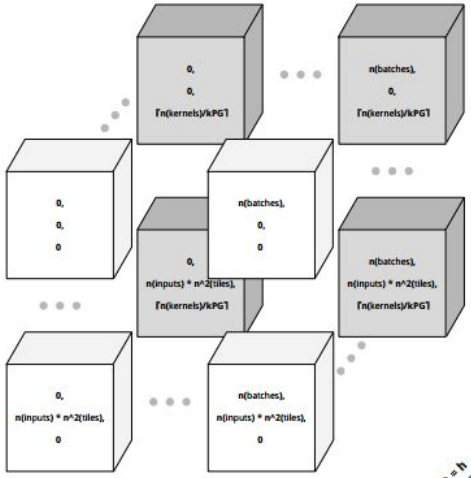
- Kernels are much smaller than images, therefore multiple kernels can be processed in the same OpenCL workgroup

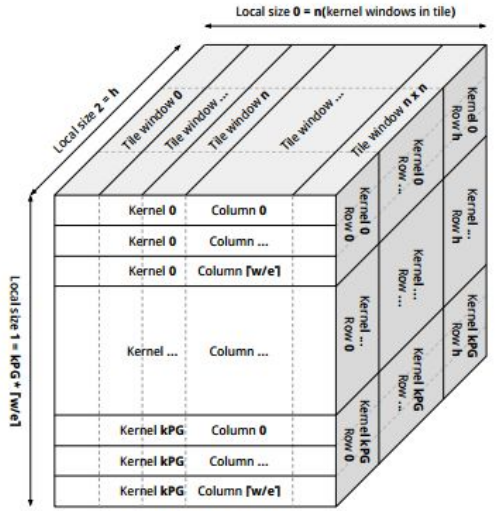
↓ ↓ ↓

- Reduce the number of global memory accesses by reusing input data stored in local memory (exploit memory locality)

Convolutional layer: the Lift expression


```
1  Conv = λ((K, B, X) => {
2    MapWrg(0)(λ((inputs_batch) => {
3      MapWrg(1)(λ((input_tile) => {
4        MapWrg(2)(λ((kernels_group) => {
5          MapLcl(0)(λ((pass_window) => {
6            ReduceWindowAndAddBias() o
7            MapLcl(2)(λ((window_row, kernels_row) => {
8              MapLcl(1)(SecondPartialReduction()
9              /* Reduce and load a row into local memory */) o
10             JoinSequences() o
11             MapLcl(1)(FirstPartialReduction()
12             /* Weigh and reduce a single tuple
13             of elements sequentially */) o
14             Split(els_per_workitem) o ZipWithInput(window_row) $ kernels_row
15           })) $ Zip(pass_window, kernels_group)
16         })) o LoadWindowIntoLocal() $ input_tile
17       })) $ GroupKernels(kernelPerGroup)(K, B)
18     })) $ inputs_batch
19   })) o SlideX() $ X})
```





Future work

1. Add new optimisational methods: generic and domain-specific.

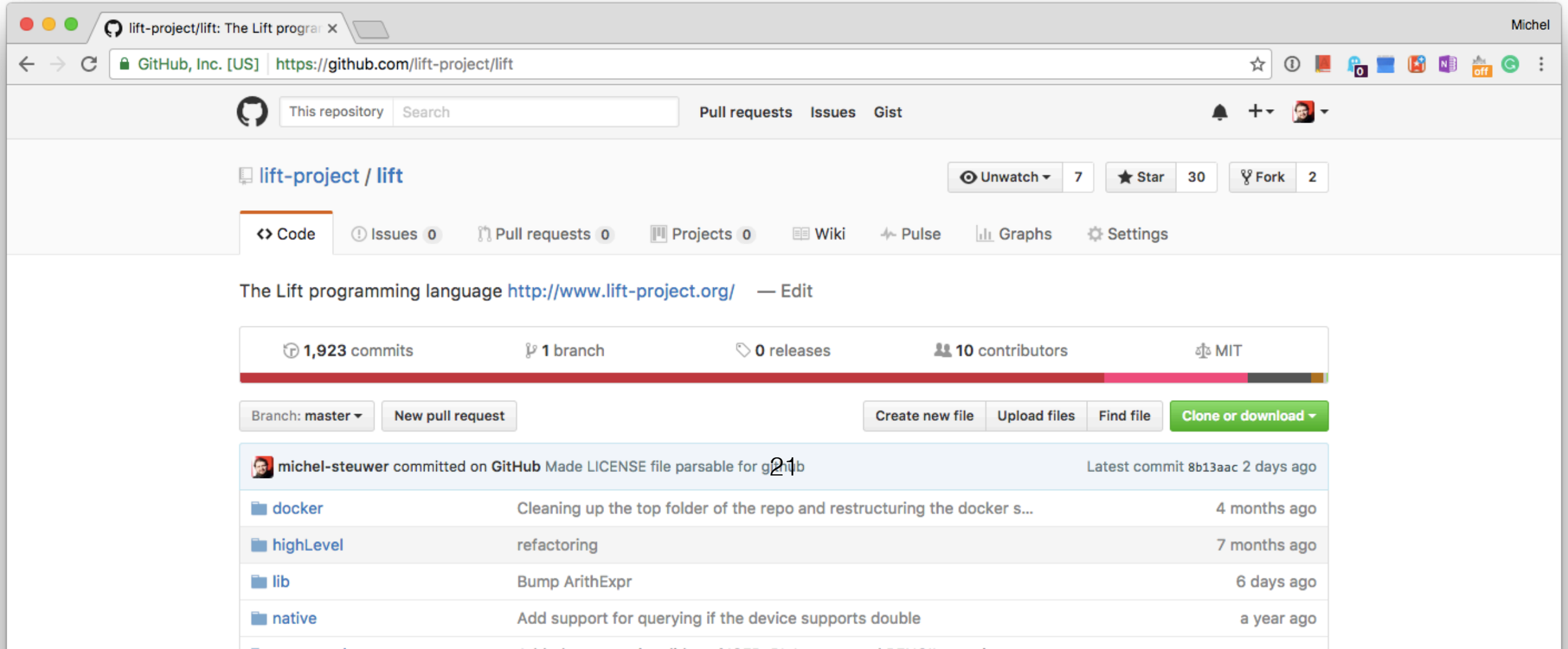
2. Investigate ways of optimising back-propagation.

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



University of Glasgow

The LIFT Project

Performance Portable Parallel
Code Generation via Rewrite Rules

www.lift-project.org

@LIFTlang

INSPIRING PEOPLE

#UofGWorldChangers

@UofGlasgow