Introduction to Dataflow Computing
Dataflow Programming Basics
What dataflow graph is generated?

DFEVar x = io.input("x", type);
DFEVar y;

y = x + 1;

io.output("y", y, type);
What dataflow graph is generated?

DFEVar x = io.input("x", type);
DFEVar y;

y = x + x + x;

io.output("y", y, type);
Conditional Choice in Kernels

- Compute both values and use a multiplexer.
  - \( x = \text{control.mux}(\text{select, option0, option1, ..., optionN}) \)
  - \( x = \text{select} ? \text{option1} : \text{option0} \)

```
DFEVar x = io.input("x", type);
DFEVar y;

y = (x > 10) ? x + 1 : x - 1

io.output("y", y, type);
```
Scalar Inputs

• Stream inputs/outputs process arrays
  – Read and write a new value each cycle
  – Off-chip data transfer required: $O(N)$

• Counters can compute intermediate streams on-chip
  – New value every cycle
  – Off-chip data transfer required: None

• Compile time constants can be combined with streams
  – Static value through the whole computation
  – Off-chip data transfer required: None

• What about something that changes occasionally?
  – Don’t want to have to recompile → Scalar input
  – Off-chip data transfer required: $O(1)$
Scalar Inputs

• Consider:

```c
fn1(int N, int *q, int *p) {
    for (int i = 0; i < N; i++)
        q[i] = p[i] + 4;
}
```

```c
fn2(int N, int *q, int *p, int C) {
    for (int i = 0; i < N; i++)
        q[i] = p[i] + C;
}
```

• In fn2, we can change the value of C without recompiling, but it is constant for the whole loop

• MaxCompiler equivalent:

```cpp
dFEVar p = io.input("p", dfeInt(32));
dFEVar C = io.scalarInput("C", dfeInt(32));

dFEVar q = p + C;

io.output("q", q, dfeInt(32));
```

A scalar input can be changed once per stream, loaded into the chip before computation starts.
On-chip memories / tables

- An FPGA has a few MB of very fast block RAM
- Can be used to explicitly store data on chip:
  - Lookup tables
  - Temporary Buffers
- *Mapped* ROMs/RAMs can also be accessed by host

```plaintext
for (i = 0; i < N; i++) {
  q[i] = table[p[i]];
}

DFEVar p = io.input("p", dfeInt(10));

DFEVar q = mem.romMapped("table", p,
                         dfeInt(32), 1024);

io.output("q", q, dfeInt(32));
```
Stream Offsets

- *Stream offsets* allow us to compute on values in a stream other than the current value.
- Offsets are relative to the *current position* in a stream; *not* the start of the stream.
- Stream data will be buffered on-chip in order to be available when needed → uses BRAM
  - Maximum supported offset size depends on the amount of on-chip BRAM available. Typically 10s of thousands of points.
Moving Average in MaxCompiler

class MovingAverageSimpleKernel extends Kernel {
    MovingAverageSimpleKernel(KernelParameters parameters) {
        super(parameters);
        DFEVar x = io.input("x", dfeFloat(8, 24));
        DFEVar prev = stream.offset(x, -1);
        DFEVar next = stream.offset(x, 1);
        DFEVar sum = prev + x + next;
        DFEVar result = sum / 3;
        io.output("y", result, dfeFloat(8, 24));
    }
}
Kernel Execution
Kernel Execution
Kernel Execution
Kernel Execution
Kernel Execution
Kernel Execution

Diagram showing a flow of operations with numbers and symbols:

- Numbers 5, 4, 3, 2, 1, 0 at the top.
- Operations involving +, -, /, and symbols like x, ?.
- Flow of operations from top to bottom, with branching paths.

Diagram details:
- Number 5 at the top, followed by operations involving +, -.
- Branching paths leading to different operations and numbers.
Boundary Cases

What about the boundary cases?
More Complex Moving Average

• To handle the boundary cases, we must explicitly code special cases at each boundary

\[
y_i = \begin{cases} 
\frac{(x_i + x_{i+1})}{2} & \text{if } i = 0 \\
\frac{(x_{i-1} + x_i)}{2} & \text{if } i = N - 1 \\
\frac{(x_{i-1} + x_i + x_{i+1})}{3} & \text{otherwise}
\end{cases}
\]
Kernel Handling Boundary Cases

```java
class MovingAverageKernel extends Kernel {

    MovingAverageKernel(KernelParameters parameters) {
        super(parameters);

        // Input
        DFEVar x = io.input("x", dfeFloat(8, 24));
        DFEVar size = io.scalarInput("size", dfeUInt(32));

        // Data
        DFEVar prevOriginal = stream.offset(x, -1);
        DFEVar nextOriginal = stream.offset(x, 1);

        // Control
        DFEVar count = control.count.simpleCounter(32, size);
        DFEVar aboveLowerBound = count > 0;
        DFEVar belowUpperBound = count < size - 1;
        DFEVar withinBounds = aboveLowerBound & belowUpperBound;
        DFEVar prev = aboveLowerBound ? prevOriginal : 0;
        DFEVar next = belowUpperBound ? nextOriginal : 0;
        DFEVar divisor = withinBounds ? constant.var(dfeFloat(8, 24), 3) : 2;
        DFEVar sum = prev + x + next;
        DFEVar result = sum / divisor;

        io.output("y", result, dfeFloat(8, 24));
    }
}
```
The Stream Loop

```cpp
uint A[...];
uint B[...];
for (int count=0; ; count += 1)

DFEVar A = io.input("input", dfeUInt(32));
DFEVar B = A + 1;
io.output("output", B, dfeUInt(32));
```
Adding a Loop Counter

for (int count=0; ; count += 1)

DFEVar A = io.input(”input” , dfeUInt(32));
DFEVar count = control.count.simpleCounter(32);
DFEVar B = A + count;
io.output(”output” , B , dfeUInt(32));
int count = 0;
for (int i=0; i<N; ++i) {
    for (int j=0; j<M; ++j) {
        count += 1;
    }
}

DFEVar A = io.input("input", dfeUInt(32));
CounterChain chain = control.count.makeCounterChain();
DFEVar i = chain.addCounter(N, 1).cast(dfeUInt(32));
DFEVar j = chain.addCounter(M, 1).cast(dfeUInt(32));
DFEVar B = A + i*100 + j;
io.output("output", B, dfeUInt(32));
Loop Unrolling with Dependence

for (i = 0; ; i += 1) {
    float d = input[i];
    float v = 2.91 - 2.0*d;
    for (iter=0; iter < 4; iter += 1)
        v = v * (2.0 - d * v);
    output[i] = v;
}

DFEVar d = io.input(”d”, dfeFloat(8, 24));
DFEVar TWO = constant.var(dfeFloat(8,24), 2.0);
DFEVar v = constant.var(dfeFloat(8,24), 2.91) − TWO*d;
for ( int iteration = 0; iteration < 4; iteration += 1) {
    v = v*(TWO− d*v);
}
io.output(”output” , v, dfeFloat(8, 24));
for (i = 0; ; i += 1) {
    float d = input[count];
    float v = 2.91 - 2.0*d;
    for (iter=0; iter < 4; iter += 1)
        v = v * (2.0 - d * v);
    output[i] = v;
}

DFEVar d = io.input(”d”, dfeFloat(8, 24));
DFEVar TWO= constant.var(dfeFloat(8,24), 2.0);
DFEVar v = constant.var(dfeFloat(8,24), 2.91) − TWO*d;

for ( int iteration = 0; iteration < 4; iteration += 1) {
    v = v*TWO− d*v;
}
io.output(”output” , v, dfeFloat(8, 24));

• The software loop has a cyclic dependence (v)
• But the unrolled datapath is acyclic
Exercise: Chapter 4 Exercise 1

\[
\text{DFEVa}\text{r } x = \text{io.}\text{input}("x", \text{type}); \\
\text{DFEVa}\text{r } y; \\
y = x \times x + x; \\
\text{io.}\text{output}("y", y, \text{type});
\]

- Build Simulation
- Look at kernel Graph before & after
- Run

\[
\text{DFEVa}\text{r } x = \text{io.}\text{input}("x", \text{type}); \\
\text{DFEVa}\text{r } y; \\
\text{DFEVa}\text{r } \text{square} = x \times x; \\
\text{square.}\text{simWatch}("\text{square}"); \\
y = \text{square} + x; \\
y.\text{simWatch}("y"); \\
\text{io.}\text{output}("y", y, \text{type});
\]

- Modify source to add simWatch()
- Run
- Check watchpoint table
Dataflow Programming
Numeric Types
Number Representation

- DFEVars have a size in bits
  - CPU restricted to char, int, long, float, double (etc)
  - DFE is much more flexible
    - 7 bit integer
    - Float 16 bit mantissa, 8 bit exponent

- Choose type to represent number in DFEVar
  - With appropriate accuracy.
  - With appropriate dynamic range.

- More bits == More FPGA area used
Number Representation for DFEs

- MaxCompiler has in-built support for floating point and fixed point/integer arithmetic
  - Depends on the type of the DFEVar
- Can type inputs, outputs and constants
- Or can cast DFEVars from one type to another
- Types are Java objects, just like DFEVars,

```java
// Create an input of type t
DFEVar io.input(String name, DFEType t);

// Create an DFEVar of type t with constant value
DFEVar constant.var(DFEType t, double value);

// Cast DFEVar y to type t
DFEVar x = y.cast(DFEType t);
```
DFE Floating Point - dfeFloat

• Floating point numbers with base 2, flexible exponent and mantissa

• Compatible with IEEE floating point **except** does not support denormal numbers
  – When Computing in Space you can use a larger exponent

```cpp
DFEType t = dfeFloat(int exponent_bits, int mantissa_bits);
```

• Examples:

<table>
<thead>
<tr>
<th></th>
<th>Exponent bits</th>
<th>Mantissa bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE single precision</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>IEEE double precision</td>
<td>11</td>
<td>53</td>
</tr>
<tr>
<td>DFE optimized low precision</td>
<td>7</td>
<td>17</td>
</tr>
</tbody>
</table>

**Why dfeFloat(7,17)…?**
DFE Fixed Point – dfeFixOffset

- Fixed point numbers
- Flexible integer and fraction bits
- Flexible sign mode
  - SignMode.UNSIGNED or SignMode.TWOSCOMPLEMENT

DFEType t = dfeFixOffset(int num_bits, int offset, SignMode sm);

- Common cases have useful aliases

<table>
<thead>
<tr>
<th>Function</th>
<th>Integer bits</th>
<th>Fraction bits</th>
<th>Sign mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfeInt(N)</td>
<td>N</td>
<td>0</td>
<td>TWOSCOMPLEMENT</td>
</tr>
<tr>
<td>dfeUInt(N)</td>
<td>N</td>
<td>0</td>
<td>UNSIGNED</td>
</tr>
<tr>
<td>dfeBool()</td>
<td>1</td>
<td>0</td>
<td>UNSIGNED</td>
</tr>
</tbody>
</table>
Mixed Types

• Can mix different types in a MaxCompiler kernel to use the most appropriate type for each operation
  – Type conversions costs area – must cast manually

• Types can be parameter to a kernel program
  – Can generate the same kernel with different types

```java
class MyKernel extends Kernel {
  public MyKernel(KernelParameters k, DFEType t_in, DFEType t_out) {
    super(k);

    DFEVar p = io.input("p", dfeFloat(8,24));
    DFEVar q = io.input("q", t_in);

    DFEVar r = p * p;

    DFEVar s = r + q.cast(r.getType());
    io.output("s", s.cast(t_out), t_out);
  }
}
```
Rounding

• When we remove bits from the RHS of a number we may want to perform *rounding*.
  – Casting / type conversion
  – Inside arithmetic operations

• Different possibilities
  – TRUNCATE: throw away unwanted bits
  – TONEAR: if >=0.5, round up (add 1)
  – TONEAREVEN: if >0.5 round up, if <0.5 round down, if =0.5 then round to the nearest even number

• Lots of less common alternatives:
  – Towards zero, towards positive infinity, towards negative infinity, random....

• Very important in iterative calculations – may affect convergence behaviour
Floating point arithmetic uses TONEAREVEN

Fixed point rounding is flexible, controlled by the `RoundingMode`

- TRUNCATE, TONEAR and TONEAREVEN are in-built

```java
DFEVar z;
...
optimization.pushRoundingMode(RoundingMode.TRUNCATE);
z = z.cast(smaller_type);
optimization.popRoundingMode();
```
Dataflow Computing Optimisation
Goals of optimisations:

- Fit more compute on DFE
- Increase frequency of kernels
- Reduce expensive data movements (e.g. back and forth between CPU and DRAM)
The four dimensions of Optimisation:

– **Bandwidth**
  How much data can you afford to move between DFE/CPU/DRAM?

– **Area**
  Resource usage as reported in your _build.log. This tells you which percentage of the chip will be doing compute every tick.

– **Utilisation**
  Actual compute. This is not reported by the tools and can vary during run time. This tells you which proportion of compute is useful compute (e.g. muxes will ‘throw away’ data).

– **Frequency**
  Higher frequency means higher throughput.
Optimisation – Introduction

• These four dimensions affect each other, e.g:
  – Increasing utilisation makes it harder to build at high frequency
  – Increasing frequency brings your bandwidth utilisation closer to their limit since you consume data at a faster rate
  – Higher Utilisation means more data is required to feed the compute unit
Optimisation – Introduction

• Most important to always remember is:

In a DFE, computation is done in space, not in time!
Optimisation – Example

• Consider the CPU code

```c
if ( cond )
    y = exp(x*a);
else
    y = exp(x*b);
```
Optimisation – Example

• Consider the CPU code
  
  If( cond )
  
  y = exp(x*a)

  else

  y = exp(x*b)

• Hw Implementation 1

• But...
Optimisation – Example

• Consider the CPU code
  
  If( cond )
  
  y = exp(x*a)

  else
  
  y = exp(x*b)

• Hw Implementation 1

• But...

Exponentials are expensive!
Optimisation – Example

• Consider the CPU code
  
  If( cond )
  y = exp(x*a)
  else
  y = exp(x*b)

• Hw Implementation 2

• But...
Optimisation – Example

• Consider the CPU code
  
  \[
  \text{If( cond )} \\
  y = \exp(x \times a) \\
  \text{else} \\
  y = \exp(x \times b)
  \]

• Hw Implementation 2

• But...

There still are two multiplications when only one result will be used
Consider the CPU code

\[
\text{If( cond )}
\]
\[
y = \exp(x*a)
\]
\[
\text{else}
\]
\[
y = \exp(x*b)
\]

Hw Implementation 3

Now the implementation is optimal
Porting N-Body to DFEs

- Very large N (~90,000 particles)
- Brute force approach
- Look at options, find optimal architecture
Problem to port to DFE

- Small code base

```c
for (int t = 0; t < T; t++) {
    memset(a, 0, N * sizeof(coord3d_t));
    for (int q = 0; q < N; q++) {
        for (int j = 0; j < N; j++) {
            float rx = p[j].p.x - p[q].p.x;
            float ry = p[j].p.y - p[q].p.y;
            float rz = p[j].p.z - p[q].p.z;
            float dd = rx*rx + ry*ry + rz*rz + EPS;
            float d = 1/ sqrtf(dd * dd * dd);
            float s = m[j] * d;
            a[q].x += rx * s;
            a[q].y += ry * s;
            a[q].z += rz * s;
        }
    }
    for (int i = 0; i < N; i++) {
        p[i].p.x += p[i].v.x;
        p[i].p.y += p[i].v.y;
        p[i].p.z += p[i].v.z;
        p[i].v.x += a[i].x;
        p[i].v.y += a[i].y;
        p[i].v.z += a[i].z;
    }
}
```

- Very long running time: ~85 seconds per timestep,
DFE Porting Process

Analysis

Architecture

Implementation
DFE Porting Process Overview

- **Step 1: Analyse Code**
  - Profile code, measure time taken
  - Measure memory requirements and working set size
  - Understand numerical requirements
- **Step 2: Architect Solution**
  - Evaluate and model partitioning options
  - Estimate speedup
- **Step 3: Implementation**
  - Transform code into partitioned architecture
  - Implement C models
  - Compile DFE (.max file)
  - Optimise and Achieve Speedup
Aim: Have a complete map of all computation and dataflow, and timings for each block of computation.

- Find out where the computation is happening (Oprofile can help) and where the data is going
- Identify major loops / draw loop graph
- Measure time spent inside major loops
Analysis: Step 1 – Dynamic Analysis

- Loop over time step
  - Loop over N particles
    - Compute forces and accelerations
  - Loop over N particles
    - Update velocities and positions
Analysis: Step 1 – Dynamic Analysis

Loop over time step

Loop over N particles

Loop over N particles
Compute forces and accelerations

Loop over N particles
Update velocities and positions

$O(N^2)$

$O(N)$
Analysis: Step 2 – Static Analysis

Aim: Understand amount of data being moved around and amount of compute to perform on it

- Analyse the data flow between the critical loops.
  - Examine what data structures are being created.
  - Identify which loops are going to work with very large arrays.
- Analyse computation inside the critical loops.
  - Count the number of floating point operations per data point
  - Analyse loop dependencies
- Understand the mathematical algorithms being used.
  - Relationship between input and runtime, memory use.
  - Understand precision requirements of each part of the algorithm
Analysis: Step 2 – Static Analysis

Loop over time step

<table>
<thead>
<tr>
<th>Loop over N particles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop over N particles</td>
</tr>
<tr>
<td>Compute forces and accelerations</td>
</tr>
<tr>
<td>Uses array of size $O(N)$</td>
</tr>
</tbody>
</table>

$O(N^2)$

<table>
<thead>
<tr>
<th>Loop over N particles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update velocities and positions</td>
</tr>
<tr>
<td>Uses array of size $O(N)$</td>
</tr>
</tbody>
</table>

$O(N)$
Analysis: Step 2 – Static Analysis

Loop over time step

Loop over N particles

Loop over N particles
  Compute forces and accelerations
  Use array of size $O(N)$

Update Forces

Update Position & velocities

Loop over N particles
  Update velocities and positions
  Use array of size $O(N)$

$O(N^2)$

$O(N)$
Analysis: Step 2 – Static Analysis

Loop over time step

Loop over N particles

Loop over N particles
  Compute forces and accelerations
  Use array of size $O(N)$
  $\sim 20$ FP Operations

Loop over N particles
  Update velocities and positions
  Use array of size $O(N)$
  $\sim 6$ FP Operations
Aim: Consider various architecture choices and understand the pros and cons of each choice

• Examine volume of data flowing through algorithm.
  – How large is the working set, i.e. does it need to be stored in LMEM or FMEM?
  – Is data access pattern known statically or calculated dynamically?
  – How much computation would be done with each loaded data value?
  – Consider the ratio of Computation to Communication!
• Examine volume of data flowing through algorithm.

  – How large is the working set, i.e. does it need to be stored in LMEM or FMEM?

On a MAX3 card, you have around 4.5MB of available ultra fast access (>10TB/s) of storage in FMEM*. If you need more than that, then you will have to use LMEM which offers 12GB, 24GB, 48GB or 96GB of storage per DFE.

* Some of this FMEM will be used by MaxCompiler for automatic buffering (for example for scheduling, or in FIFOs between Kernels). How much, varies widely from one design to another.
Analysis: Step 3 – DFE Architecture Options

Loop over time step

Loop over N particles

Loop over N particles
  Compute forces and accelerations
  Use array of size $O(N)$
  \textbf{Read 4 floats/particle $\rightarrow$ up to 1.4MB}
  $\sim$20 FP Operations

\textbf{Update 3 floats/particle $\rightarrow$ up to 1MB}

Loop over N particles
  Update velocities and positions
  Use array of size $O(N)$
  \textbf{Read 3 floats/particle $\rightarrow$ up to 1MB}
  \textbf{Update 6 floats/particle $\rightarrow$ up to 2.1MB}
  $\sim$6 FP Operations

$O(N^2)$

$O(N)$
• Examine volume of data flowing through algorithm.

  – Is data access pattern known statically?
    If the pattern is static then you can either use one of the command generators provided (LINEAR1D, ...) or generate commands on the CPU and stream them in.

  – Is data access pattern computed dynamically?
    If the address of the data you need to read or write needs to be computed on the Dataflow Engine, then your access pattern is dynamic and you will have to generate the LMEM command inside a Kernel.
• For N-Body problem, access pattern is static and linear 1D

```java
for (int q = 0; q < N; q++) {
    for (int j = 0; j < N; j++) {
        ...
    }
}
```

<table>
<thead>
<tr>
<th>q=0</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>j=0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analysis: Step 3 – DFE Architecture Options

- For N-Body problem, access pattern is static and linear 1D

```java
for (int q = 0; q < N; q++) {
    for (int j = 0; j < N; j++) {
        ...
    }
}
```

<table>
<thead>
<tr>
<th>q=0</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>j=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analysis: Step 3 – Architecture Options

• For N-Body problem, access pattern is static and linear 1D

```java
for (int q = 0; q < N; q++) {
    for (int j = 0; j < N; j++) {
        ...
    }
}
```

<table>
<thead>
<tr>
<th>q=0</th>
<th>j=2</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
Analysis: Step 3 – Architecture Options

• For N-Body problem, access pattern is static and linear 1D

```cpp
for (int q = 0; q < N; q++) {
    for (int j = 0; j < N; j++) {
        ...
    }
}
```

![Diagram showing the access pattern for N-Body problem](image)
Analysis: Step 3 – Architecture Options

• For N-Body problem, access pattern is static and linear 1D

```java
for (int q = 0; q < N; q++) {
    for (int j = 0; j < N; j++) {
        ...
    }
}
```
Analysis: Step 3 – Architecture Options

- For N-Body problem, access pattern is static and linear 1D

```java
for (int q = 0; q < N; q++) {
    for (int j = 0; j < N; j++) {
        ...
    }
}
```

<table>
<thead>
<tr>
<th>q=1</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>j=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analysis: Step 3 – Architecture Options

• Examine volume of data flowing through algorithm.
  
  – How much computation would be done with each loaded data value?

By carefully choosing your memory access pattern, you can increase data reuse and decrease memory bandwidth requirement. LMEM has a limit which depends on the platform and its frequency. For some DFE at 350MHz this is about 33.5 GB/s.

With complex access patterns and many streams, actual bandwidth could be different.
Analysis: Step 3 – Architecture Options

- What needs to be on the DFE, and what can stay on the CPU?
  - How many functions require access to the largest arrays?
  - Do the functions that use the large arrays also have long runtime?

Moving the bulk of the compute to the DFE might not be the right answer.

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU Time</th>
<th>DFE Time</th>
<th>Data Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function1</td>
<td>1000s</td>
<td>5s</td>
<td>5s</td>
</tr>
<tr>
<td>Function2</td>
<td>1s</td>
<td>1s</td>
<td>10G transferred</td>
</tr>
</tbody>
</table>

Final result only

CPU time 1001s

Option 1 time 11s

Option 2 time 6s
Examine what data can be pre-computed.

- Which functions actually need to be run inside the loops?

Consider the following loops:

```python
for i = 0..99 do
    double a = cos(i*2*PI/100)
    for j = 0..9999
        // do some compute
```

Assume that we wish to put these loops onto a DFE and that each iteration of j takes one cycle. Putting the computation of a onto the DFE as well means that we will be using hardware resources to compute a cosine that is needed only once every 10,000 cycles. This is very wasteful. Instead, it would be better to compute the 100 different values of a and store them into an EMEM on the DFE.
NOTE: There is a high overhead(*) to create a new kernel (each running in their own clock domain), so keep the number of kernels low.

- Your design can have one or more kernels. How do you decide how many kernels to build:

1. Your design may have multiple passes. Each pass could have a separate kernel.

2. You may be able to partition your design into pieces with dynamic and/or different input and output bandwidth requirements

(*) A Maxeler architecture is a Globally Asynchronous Locally Synchronous (GALS)
Analysis: Step 3 – Architecture Options

- **NBody Option 1**
Analysis: Step 3 – Architecture Options

- **NBody Option 2**

  ![Diagram](image)

  - **CPU**
  - **Kernel 1**: Compute forces and accelerations
  - **LMEM**: Position, mass
  - **DFE**: Write masses and positions. Do this each time step
  - Send accelerations
  - Read positions and masses
Analysis: Step 3 – Architecture Options

- **NBody Option 3**

  - **Kernel 1**
    - Compute forces and accelerations
    - FMEM Position, mass
  - **Kernel 2**
    - Update positions and velocities
    - FMEM Velocity

  - **DFE**
    - Send positions and accelerations
    - Send updated positions
    - Write masses & positions. <Only once>
    - Write velocities. Only once.
    - Send updated positions
Analysis: Step 3 – Architecture Options

- NBody Option 4

![Diagram showing CPU, DFE, FMEM, and Kernel 1 with steps: Write masses and positions each time step. Send accelerations. Compute forces and accelerations.]

- Kernel 1
  - Compute forces and accelerations

- FMEM
  - Position, mass
Conclusions – Porting CPU Software to DFEs

• Look at Options
• Process: Analysis, Architecture, Implementation
• Carefully minimise the number of kernels needed.
• First move data from CPU to DFE and then consider which computations need to move with the data