Towards Automated Design Space Exploration and Code Generation using Type Transformations

S Waqar Nabi & Wim Vanderbauwhede

www.tytra.org.uk
Using Safe Transformations and a Cost-Model for HPC on FPGAs

- The TyTra project context
  - Our approach, blue-sky target, down-to-earth target, where we are now, how we are different

- Key contributions
  - (1) Type transformations to create design-variants, (2) a new Intermediate Language, and (3) an FPGA Cost model

- The cost model
  - Performance and resource-usage estimates, some results

Using safe transformations and an associated light-weight cost-model opens the route to a fully automated design-space exploration flow
THE CONTEXT

Our approach, blue-sky target, down-to-earth target, where we are now, how we are different
Blue Sky Target
Blue Sky Target

Legacy Scientific Code

Heterogeneous HPC Target Description

Cost Model

Optimized HPC solution!

The goal that keeps us motivated!
(The pragmatic target is somewhat more modest….)
Our focus is on FPGA targets, and we currently require design entry in a Functional Language using High-Level Functions (maps, folds) [a kind of DSL]
The cunning plan…

1. Use the **functional programming paradigm** to (auto) generate **program-variants** which translate to **design-variants** on the FPGA.

2. Create an **Intermediate Language** that:
   - Is able to capture points entire design-space
   - Allows a light-weight cost-model to be built around it
   - Is a convenient target for front-end compiler

3. Create a **light-weight cost-model** that can **estimate** the performance and resource-utilization for each **variant**.

A performance portable code-base that builds on a purely software programming paradigm.
And you may very well ask…

Is your plan as cunning as a fox who’s just been appointed Professor of Cunning at Oxford University?

The jury is still out…
How our work is different

- Our observations on limitations of current tools and flows:

1. Design-entry in a *custom* high-level language which nevertheless has hardware-specific semantics

2. Architecture of the FPGA-solution specified by programmer; compilers cannot optimize it.

3. Solutions create soft-processors on the FPGA; not optimized for HPC (orientation towards embedded applications)

4. Design-space exploration requires prohibitively long time

5. Compiler is application specific (e.g. DSP applications)

We are not there yet, but in principle, our approach entirely eliminates the first four, and mitigates the fifth.
KEY CONTRIBUTIONS

(1) Type transformations for generating program variants, (2) a new Intermediate Language, and (3) a light-weight Cost Model
1. Type Transformations to Generate Program Variants

- Functional Programming

- Types
  - More general than types in C
  - Our focus is on types of functions that perform array operations
    - reshape, maps and folds

- Type transformations
  - Can be derived automatically
  - Provably correct
  - Essentially reshape the arrays

A functional paradigm with high-level functions allows creation of design-variants that are correct-by-construction.
Illustration of Variant Generation through Type-Transformation

- **typeA**: \( \text{Vect} (\text{im} \times \text{jm} \times \text{km}) \) \( \text{dataType} -- 1D \text{ data} 
  
  - Single execution thread

- **typeB**: \( \text{Vect} \ \text{km} \ (\text{Vect} \ \text{im} \times \text{jm} \ \text{dataType}) -- \text{transformed} \ 2D \text{ data} 
  
  - (\text{km} \ \text{concurrent execution threads})

- output = \( \text{map}_{\text{pipe}} \ \text{kernel}_\text{func} \ \text{input} -- \text{original program} 

- inputTr = \( \text{reshapeTo} \ \text{km} \ \text{input} -- \text{reshaping data} 

- output = \( \text{map}_{\text{par}} \ (\text{map}_{\text{pipe}} \ \text{kernel}_\text{func}) \ \text{inputTr} -- \text{new program} 

Simple and provably correct transformations in a high-level functional language translates to design-variants on the FPGA.
2. A New Intermediate Language

- Strongly and statically typed

- All computations expressed as SSA (Single-Static Assignments)

- Largely (and deliberately) based on the LLVM-IR

Manage-IR

- Deals with
  - memory objects (arrays)
  - streams (loops over arrays)
  - offset streams
  - loops over work-unit
  - block-memory transfers

Compute-IR

- Streaming model

- SSA instructions define the datapath
2. A New Intermediate Language

```plaintext
@main.a = addrSpace(12) ui18,
!"istream", !"CONT", !0, !"strobj_a"
@...[other ports]
define void @f1 ( ...args...) pipe {
  ui18 %1 = add ui18 %a, %b
  ui18 %2 = add ui18 %c, %c
  ui18 %3 = mul ui18 %1, %2
  ui18 %y = add ui18 %3, @k 
}
define void @main () {
call @f1(...args...) pipe 
}
```

```plaintext
@main.a_01 = ...
@main.a_02 = ...
@...[other ports]
define void @f1 ( ...args...) pipe {...}
define void @f2 ( ...args...) par {
call @f1(...args...) pipe
call @f1(...args...) pipe
call @f1(...args...) pipe
call @f1(...args...) pipe 
}
define void @main () {
call @f2(...args...) par 
}
```
3. Cost Model
THE FPGA COST-MODEL

Performance Estimate, Resource-utilization estimate, Experimental Results
A set of standardized experiments feed target-specific empirical data to the cost model, and the rest comes from the IR description.
Two Types of Estimates

- **Resource-Utilization Estimates**
  - ALUTs, REGs, DSPs

- **Performance Estimates**
  - Estimating memory-access bandwidth for specific data patterns
  - Estimating FPGA operating frequency

Both estimates needed to allow compiler to choose the best design variant.
1. Resource Estimates

◆ Observation
  o Regularity of FPGA fabric allows some very simple first or second order expressions to be built up for most instructions based on a few experiments.

◆ Key Determinants
  o Primitive (SSA) instructions used in IR of the kernel functions
  o Data-types
  o Structure of various functions (par, comb, par, seq)
  o Control logic over-head

A set of one-time simple synthesis experiments on the target device helps us create a very accurate resource-utilization cost model
Light-weight cost expressions associated with every legal SSA instruction in the TyTra-IR
2. Performance Estimate

- Effective Work-Unit Throughput (EWUT)
  - Work-Unit = Executing the kernel over the entire index-space

- Key Determinants
  - Memory execution model
  - Sustained memory bandwidth for the target architecture and design-variant
    - Data-access pattern
  - Design configuration of the FPGA
  - Operating frequency of the FPGA
  - Compute-bound or IO-bound?

Performance model is trickier, especially calculating estimates of sustained memory bandwidth and FPGA operating frequency.
2. Performance Estimate

- Effective Work-Unit Throughput (EWUT)
  - Work-Unit = Executing the kernel over the entire index-space

- Key Determinants
  - Memory execution model
  - Sustained memory bandwidth for the target architecture and design-variant
    - Data-access pattern
  - Design configuration of the FPGA
  - Operating frequency of the FPGA
  - Compute-bound or IO-bound?

Performance model is trickier, especially calculating estimates of sustained memory bandwidth and FPGA operating frequency.
Performance Estimate

Dependence on Memory Execution Model

Three **Types** of memory executions
A given design-variant can be categorized based on:
- Architectural description
- IR description
Performance Estimate
Dependence on Memory Execution Model

Three **Types** of memory executions
A given design-variant can be categorized based on:
- Architectural description
- IR description
Performance Estimate
Dependence on Memory Execution Model

**Work-Unit Iterations**

**Type A**

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td></td>
</tr>
<tr>
<td>Device-DRAM</td>
<td></td>
</tr>
<tr>
<td>Device-Buffers</td>
<td></td>
</tr>
<tr>
<td>Offset-Buffers</td>
<td></td>
</tr>
<tr>
<td>Device-Buffers</td>
<td></td>
</tr>
<tr>
<td>Device-DRAM</td>
<td></td>
</tr>
</tbody>
</table>

**All iterations**
Performance Estimate
Dependence on Memory Execution Model

Work-Unit Iterations
Type B

First Iteration only

Last Iteration only

All other iterations

Kernel Pipeline Execution
Device-Buffers ↔ Offset-Buffers
Device-DRAM ↔ Device-Buffers
Host ↔ Device-DRAM

Activity

Time
Performance Estimate
Dependence on Memory Execution Model

Once a design-variant is categorized, performance can be estimated accordingly.
2. Performance Estimate

- **Effective Work-Unit Throughput (EWUT)**
  - *Work-Unit* = *Executing the kernel over the entire index-space*

- **Key Determinants**
  - Memory execution model
  - **Sustained memory bandwidth for the target architecture and design-variant**
    - Data-access pattern
  - Design configuration of the FPGA
  - Operating frequency of the FPGA
  - Compute-bound or IO-bound?

Performance model is trickier, especially calculating estimates of sustained memory bandwidth and FPGA operating frequency.
Performance Estimate

Dependence on **Data Access Pattern**

- We have defined a \( \rho \) factor defined as a *scaling factor* of the *peak* memory bandwidth
  - *Varies from 0-1*
  - *Based on data-access pattern*
  - *Derived empirically through one-time standardized experiments on target node*
2. Performance Estimate

- **Effective Work-Unit Throughput (EWUT)**
  - *Work-Unit* = *Executing the kernel over the entire index-space*

- **Key Determinants**
  - *Memory execution model*
  - Sustained memory bandwidth for the target architecture and design-variant
    - Data-access pattern
  - *Design configuration of the FPGA*
  - *Operating frequency of the FPGA*
  - *Compute-bound or IO-bound?*

Determined from the IR description of design-variant

Performance model is trickier, especially calculating estimates of sustained memory bandwidth and FPGA operating frequency.
### Performance Estimates

#### The Parameters and their Evaluation

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Key Dependence</th>
<th>Short Description</th>
<th>Evaluation Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_{PB}$</td>
<td>Node architecture</td>
<td>The host-device peak bandwidth (typically PCI Express).</td>
<td>Architecture description fed to compiler</td>
</tr>
<tr>
<td>$\rho_H$</td>
<td>Node architecture &amp; design-variant</td>
<td>Scaling factor, host-device bandwidth</td>
<td><em>Experiments</em> with different data-patterns on the target node.</td>
</tr>
<tr>
<td>$G_{PB}$</td>
<td>Node architecture</td>
<td>The device DRAM peak bandwidth.</td>
<td>Architecture description fed to compiler</td>
</tr>
<tr>
<td>$\rho_H$</td>
<td>Device architecture &amp; design-variant</td>
<td>Scaling factor, host-device bandwidth</td>
<td><em>Experiments</em> with different data-patterns on the target node.</td>
</tr>
<tr>
<td>$N_{GS}$</td>
<td>Program</td>
<td>Global-size of work-items in NDRange</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_{WPT}$</td>
<td>Program</td>
<td>Words per tuple per work-item</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_{NDR}$</td>
<td>Program</td>
<td>Repetition of kernel over NDRange</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_{off}$</td>
<td>Program</td>
<td>Maximum offset in a stream</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$K_{PD}$</td>
<td>Program &amp; design-variant</td>
<td>Pipeline depth of kernel</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$F_D$</td>
<td>Program &amp; design-variant</td>
<td>Device’s operating frequency</td>
<td>Compiler costing of IR</td>
</tr>
<tr>
<td>$N_{TO}$</td>
<td>Program &amp; design-variant</td>
<td>Cycles per instruction</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_I$</td>
<td>Program &amp; design-variant</td>
<td>Instructions per PE</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$K_{NL}$</td>
<td>Program &amp; design-variant</td>
<td>Number of parallel kernel lanes</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$D_V$</td>
<td>Program &amp; design-variant</td>
<td>Degree of vectorization per lane</td>
<td>Compiler parse of IR</td>
</tr>
</tbody>
</table>
### Performance Estimates

Parameters from **Architecture Description**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Key Dependence</th>
<th>Short Description</th>
<th>Evaluation Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_{PB}$</td>
<td>Node architecture</td>
<td>The host-device peak bandwidth (typically PCI Express).</td>
<td>Architecture description fed to compiler</td>
</tr>
<tr>
<td>$\rho_H$</td>
<td>Node architecture &amp; design-variant</td>
<td>Scaling factor, host-device bandwidth</td>
<td><em>Experiments</em> with different data-patterns on the target node.</td>
</tr>
<tr>
<td>$G_{PB}$</td>
<td>Node architecture</td>
<td>The device DRAM peak bandwidth.</td>
<td>Architecture description fed to compiler</td>
</tr>
<tr>
<td>$\rho_H$</td>
<td>Device architecture &amp; design-variant</td>
<td>Scaling factor, host-device bandwidth</td>
<td><em>Experiments</em> with different data-patterns on the target node.</td>
</tr>
<tr>
<td>$N_{GS}$</td>
<td>Program</td>
<td>Global-size of work-items in NDRange</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_{WPT}$</td>
<td>Program</td>
<td>Words per tuple per work-item</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_{NDR}$</td>
<td>Program</td>
<td>Repetition of kernel over NDRange</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_{off}$</td>
<td>Program</td>
<td>Maximum offset in a stream</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$K_{PD}$</td>
<td>Program &amp; design-variant</td>
<td>Pipeline depth of kernel</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$F_D$</td>
<td>Program &amp; design-variant</td>
<td>Device’s operating frequency</td>
<td>Compiler costing of IR</td>
</tr>
<tr>
<td>$N_{TO}$</td>
<td>Program &amp; design-variant</td>
<td>Cycles per instruction</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_I$</td>
<td>Program &amp; design-variant</td>
<td>Instructions per PE</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$K_{NL}$</td>
<td>Program &amp; design-variant</td>
<td>Number of parallel kernel lanes</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$D_V$</td>
<td>Program &amp; design-variant</td>
<td>Degree of vectorization per lane</td>
<td>Compiler parse of IR</td>
</tr>
</tbody>
</table>
### Performance Estimates

Parameters Calculated **Empirically**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Key Dependence</th>
<th>Short Description</th>
<th>Evaluation Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_{PB}$</td>
<td>Node architecture</td>
<td>The host-device peak bandwidth (typically PCI Express).</td>
<td>Architecture description fed to compiler</td>
</tr>
<tr>
<td>$\rho_H$</td>
<td>Node architecture &amp; design-variant</td>
<td>Scaling factor, host-device bandwidth</td>
<td><em>Experiments</em> with different data-patterns on the target node.</td>
</tr>
<tr>
<td>$G_{PB}$</td>
<td>Node architecture</td>
<td>The device DRAM peak bandwidth.</td>
<td>Architecture description fed to compiler</td>
</tr>
<tr>
<td>$\rho_H$</td>
<td>Device architecture &amp; design-variant</td>
<td>Scaling factor, host-device bandwidth</td>
<td><em>Experiments</em> with different data-patterns on the target node.</td>
</tr>
<tr>
<td>$N_{GS}$</td>
<td>Program</td>
<td>Global-size of work-items in NDRRange</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_{WPT}$</td>
<td>Program</td>
<td>Words per tuple per work-item</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_{NDR}$</td>
<td>Program</td>
<td>Repetition of kernel over NDRRange</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_{off}$</td>
<td>Program</td>
<td>Maximum offset in a stream</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$K_{PD}$</td>
<td>Program &amp; design-variant</td>
<td>Pipeline depth of kernel</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$F_D$</td>
<td>Program &amp; design-variant</td>
<td>Device’s operating frequency</td>
<td>Compiler costing of IR</td>
</tr>
<tr>
<td>$N_{TO}$</td>
<td>Program &amp; design-variant</td>
<td>Cycles per instruction</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_I$</td>
<td>Program &amp; design-variant</td>
<td>Instructions per PE</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$K_{NL}$</td>
<td>Program &amp; design-variant</td>
<td>Number of parallel kernel lanes</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$D_V$</td>
<td>Program &amp; design-variant</td>
<td>Degree of vectorization per lane</td>
<td>Compiler parse of IR</td>
</tr>
</tbody>
</table>
### Performance Estimates

Parameters derived from IR description of Kernel

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Key Dependence</th>
<th>Short Description</th>
<th>Evaluation Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_{PB}$</td>
<td>Node architecture</td>
<td>The host-device peak bandwidth (typically PCI Express).</td>
<td>Architecture description fed to compiler</td>
</tr>
<tr>
<td>$\rho_H$</td>
<td>Node architecture &amp; design-variant</td>
<td>Scaling factor, host-device bandwidth</td>
<td><em>Experiments</em> with different data-patterns on the target node.</td>
</tr>
<tr>
<td>$G_{PB}$</td>
<td>Node architecture</td>
<td>The device DRAM peak bandwidth.</td>
<td>Architecture description fed to compiler</td>
</tr>
<tr>
<td>$\rho_H$</td>
<td>Device architecture &amp; design-variant</td>
<td>Scaling factor, host-device bandwidth</td>
<td><em>Experiments</em> with different data-patterns on the target node.</td>
</tr>
<tr>
<td>$N_{GS}$</td>
<td>Program</td>
<td>Global-size of work-items in NDRRange</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_{WPT}$</td>
<td>Program</td>
<td>Words per tuple per work-item</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_{NDR}$</td>
<td>Program</td>
<td>Repetition of kernel over NDRRange</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_{off}$</td>
<td>Program</td>
<td>Maximum offset in a stream</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$K_{PD}$</td>
<td>Program &amp; design-variant</td>
<td>Pipeline depth of kernel</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$F_D$</td>
<td>Program &amp; design-variant</td>
<td>Device’s operating frequency</td>
<td>Compiler costing of IR</td>
</tr>
<tr>
<td>$N_{TO}$</td>
<td>Program &amp; design-variant</td>
<td>Cycles per instruction</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$N_I$</td>
<td>Program &amp; design-variant</td>
<td>Instructions per PE</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$K_{NL}$</td>
<td>Program &amp; design-variant</td>
<td>Number of parallel kernel lanes</td>
<td>Compiler parse of IR</td>
</tr>
<tr>
<td>$D_V$</td>
<td>Program &amp; design-variant</td>
<td>Degree of vectorization per lane</td>
<td>Compiler parse of IR</td>
</tr>
</tbody>
</table>
Performance Estimates

The Expressions

\[
EWUT_A = \frac{1}{\frac{N_{GS} \cdot N_{WPT}}{H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max \left( \frac{N_{GS} \cdot N_{WPT}}{G_{PB} \cdot \rho_G}, \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V} \right)}
\]

\[
EWUT_B = \frac{1}{\frac{N_{GS} \cdot N_{WPT}}{N_{WU} \cdot H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max \left( \frac{N_{GS} \cdot N_{WPT}}{G_{PB} \cdot \rho_G}, \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V} \right)}
\]

\[
EWUT_C = \frac{1}{\frac{N_{GS} \cdot N_{WPT}}{N_{WU} \cdot H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V}}
\]
Performance Estimates

The Expressions

\[
EWUT_A = \frac{1}{\frac{NGS \cdot NWPT}{HPB \cdot \rho_H}} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max\left(\frac{NGS \cdot NWPT}{G_{PB} \cdot \rho_G}, \frac{NGS \cdot NWPT \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V}\right)
\]

\[
EWUT_B = \frac{1}{\frac{NGS \cdot NWPT}{NWU \cdot HPB \cdot \rho_H}} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max\left(\frac{NGS \cdot NWPT}{G_{PB} \cdot \rho_G}, \frac{NGS \cdot NWPT \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V}\right)
\]

\[
EWUT_C = \frac{1}{\frac{NGS \cdot NWPT}{NWU \cdot HPB \cdot \rho_H}} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \frac{NGS \cdot NWPT \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V}
\]
Performance Estimates
The Expressions

\[
EWUT_A = \frac{N_{GS\cdot N_{WPT}}}{H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max\left(\frac{N_{GS\cdot N_{WPT}}}{G_{PB} \cdot \rho_G}, \frac{N_{GS\cdot N_{WPT} \cdot N_{TO} \cdot N_I}}{F_D \cdot K_{NL} \cdot D_V}\right)
\]

\[
EWUT_B = \frac{N_{GS\cdot N_{WPT}}}{N_{WU} \cdot H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max\left(\frac{N_{GS\cdot N_{WPT}}}{G_{PB} \cdot \rho_G}, \frac{N_{GS\cdot N_{WPT} \cdot N_{TO} \cdot N_I}}{F_D \cdot K_{NL} \cdot D_V}\right)
\]

\[
EWUT_C = \frac{N_{GS\cdot N_{WPT}}}{N_{WU} \cdot H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \frac{N_{GS\cdot N_{WPT} \cdot N_{TO} \cdot N_I}}{F_D \cdot K_{NL} \cdot D_V}
\]
Performance Estimates

The Expressions

\[
EWUT_A = \frac{N_{GS} \cdot N_{WPT}}{H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max\left(\frac{N_{GS} \cdot N_{WPT}}{G_{PB} \cdot \rho_G}, \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_v}\right)
\]

\[
EWUT_B = \frac{N_{GS} \cdot N_{WPT}}{N_{WU} \cdot H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max\left(\frac{N_{GS} \cdot N_{WPT}}{G_{PB} \cdot \rho_G}, \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_v}\right)
\]

\[
EWUT_C = \frac{N_{GS} \cdot N_{WPT}}{N_{WU} \cdot H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_v}
\]
Performance Estimates

The Expressions

\[ EWUT_A = \frac{N_{GS} \cdot N_{WPT}}{H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max \left( \frac{N_{GS} \cdot N_{WPT}}{G_{PB} \cdot \rho_G}, \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_{I}}{F_D \cdot K_{NL} \cdot D_V} \right) \] 

\[ EWUT_B = \frac{N_{GS} \cdot N_{WPT}}{N_{WU} \cdot H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max \left( \frac{N_{GS} \cdot N_{WPT}}{G_{PB} \cdot \rho_G}, \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_{I}}{F_D \cdot K_{NL} \cdot D_V} \right) \] 

\[ EWUT_C = \frac{N_{GS} \cdot N_{WPT}}{N_{WU} \cdot H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_{I}}{F_D \cdot K_{NL} \cdot D_V} \]
Performance Estimates
The Expressions

\[
EWUT_A = \frac{N_{GS} \cdot N_{WPT}}{H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max\left(\frac{N_{GS} \cdot N_{WPT}}{G_{PB} \cdot \rho_G}, \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V}\right)
\]

\[
EWUT_B = \frac{N_{GS} \cdot N_{WPT}}{N_{WU} \cdot H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \max\left(\frac{N_{GS} \cdot N_{WPT}}{G_{PB} \cdot \rho_G}, \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V}\right)
\]

\[
EWUT_C = \frac{N_{GS} \cdot N_{WPT}}{N_{WU} \cdot H_{PB} \cdot \rho_H} + \frac{N_{off}}{G_{PB} \cdot \rho_G} + \frac{K_{PD}}{F_D} + \frac{N_{GS} \cdot N_{WPT} \cdot N_{TO} \cdot N_I}{F_D \cdot K_{NL} \cdot D_V}
\]
Performance Estimates
Experimental Results (Type C)

<table>
<thead>
<tr>
<th>Resource</th>
<th>C2(E)</th>
<th>C2(A)</th>
<th>C1(E)</th>
<th>C1(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUTs</td>
<td>528</td>
<td>546</td>
<td>5764</td>
<td>5837</td>
</tr>
<tr>
<td>REGs</td>
<td>534</td>
<td>575</td>
<td>4504</td>
<td>4892</td>
</tr>
<tr>
<td>BRAM(bits)</td>
<td>5418</td>
<td>5400</td>
<td>11304</td>
<td>11250</td>
</tr>
<tr>
<td>DSPs</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cycles/Kernel</td>
<td>292</td>
<td>308</td>
<td>180</td>
<td>185</td>
</tr>
<tr>
<td>EWGT</td>
<td>57K</td>
<td>43K</td>
<td>92K</td>
<td>72K</td>
</tr>
</tbody>
</table>

Estimated (E) vs actual (A) cost and throughput for C2 and C1 configurations of a successive over-relaxation kernel.

[Note that the cycles/kernel are estimated very accurately, but the Effective Workgroup Throughput (EWGT) is off because of inaccuracy of frequency estimate for the FPGA]
Does the TyTra Approach Work?
CONCLUSION
The route to Automated Design Space Exploration on FPGAs for HPC Applications

- The larger aim is to create a turn-key compiler for:
  
  \[ \text{Legacy scientific code} \xrightarrow{} \text{Heterogeneous HPC Platform} \]
  
  - Current focus is on FPGAs, and on using a Functional Language design entry

- Our main contributions are:
  
  - Type transformations to create design-variants,
  - New Intermediate Language, and
  - FPGA Cost model

- Our FPGA Cost Model
  
  - Works on the TyTra-UIR, is light-weight, accurate (enough), and allows us to evaluate design-variants

Using safe transformations on a functional language paradigm and a light-weight cost-model to brings us closer to a turn-key HPC compiler for legacy code
The woods are lovely, dark and deep,
But I have promises to keep,
And lines to code before I sleep,
And lines to code before I sleep.
EXTRAS
Parallel Approaches

• What we do is very similar to:
  • Loop optimizations to accelerate a scientific application
  • Using skeletons to create a high-level abstraction for parallel programming
  • Tools that automatically explore design-space
Our Approach to a Light-Weight Cost Model

◆ An IR sufficiently low-level to expose the parameters needed for the
  o The TyTra-IR has sufficient structural information to associate it directly with resources on an FPGA

◆ Because TyTra-IR is a customized language, we can ensure that:
  o All legal instructions (and structures) have a cost associated with them
  o As long as the front-end compiler can target a HLL on the TyTra-IR, we can cost HL program variants
  o Costing resources on specific FPGA devices, and estimating memory bandwidth for various patterns on the target node, requires some empirical data.
  • We are working on creating a set of standardized experiments that

We are not there yet, but in principle, our approach entirely eliminates all these limitations.
Quite a few avenues...

- Experiment with more kernels, their program-variants, estimated vs actual costs, (correct) code-generation. Use (CHStone) benchmarks.
- Computation-aware caches, optimized for halo-based scientific computations
- Integrate with Altera-OpenCL platform for host-device communication
- Back-end optimizations, LLVM passes, LLVM $\rightarrow$ TyTra-IR translation
- Route to TyTra-IR from SAC
- Integrate Tytra-FPGA flow with SAC$\rightarrow$GPU(OpenCL flow) for heterogeneous targets
- Use of Multi-party Session Types to ensure correctness of transformations
  - Even code-generation for clusters?
- Abstract descriptions of target hardware
- SystemC-TLM model to profile application and high-level partitioning in a heterogeneous environment
Quite a few avenues...

- Experiment with more kernels, their program-variants, estimated vs actual costs, (correct) code-generation. Use (CHStone) benchmarks.

- Computation-aware caches, optimized for halo-based scientific computations

- Integrate with Altera-OpenCL platform for host-device communication

- Back-end optimizations, LLVM passes, LLVM \(\rightarrow\) TyTra-IR translation

- Route to TyTra-IR from SAC

- Integrate Tytra-FPGA flow with SAC\(\rightarrow\)GPU

- Use of Multi-party Session Types to ensure correctness of transformations
  - Even code-generation for clusters?

- Abstract descriptions of target hardware

- SystemC-TLM model to profile application and high-level partitioning in a heterogeneous environment

etcetera, etcetera, etcetera
The platform model for TyTra (FPGA)
# The Manage-IR; Memory Objects

```bash
@mem_a = addrSpace(3) <NLinear x ui18>,
!"hmap" , !"NULL" ,
!"init" , !"a.dat" ,
!"readPorts" , !1 ,
!"writePorts" , !1
```

<table>
<thead>
<tr>
<th>TyTra-IR</th>
<th>OpenCL view</th>
<th>LLVM-SPIR View</th>
<th>Hardware (FPGA)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cmem</strong> Constant Memory</td>
<td>Constant Memory</td>
<td>3: Constant</td>
<td></td>
</tr>
<tr>
<td><strong>Imem</strong> Instruction Memory</td>
<td>Constant Memory</td>
<td></td>
<td>DistRAM / BRAM</td>
</tr>
<tr>
<td><strong>Pipemem</strong> Pipeline registers</td>
<td></td>
<td></td>
<td>DistRAM</td>
</tr>
<tr>
<td><strong>Pmem</strong> Private Memory (Data Mem for Instruc’ Proc’)</td>
<td>Private Memory</td>
<td>0: Private</td>
<td>DistRAM</td>
</tr>
<tr>
<td><strong>Cachemem</strong> Data (and Constant) Cache</td>
<td></td>
<td></td>
<td>DistRAM / BRAM</td>
</tr>
<tr>
<td><strong>Lmem</strong> Local (shared) memory</td>
<td>Local Memory</td>
<td>4: Local</td>
<td>M20K (BRAM) or Dist RAM</td>
</tr>
<tr>
<td><strong>Gmem</strong> Global memory</td>
<td>Global Memory</td>
<td>1: Global</td>
<td>On-board DRAM</td>
</tr>
<tr>
<td><strong>Hmem</strong> Host memory</td>
<td>Host Memory</td>
<td></td>
<td>Host communication</td>
</tr>
</tbody>
</table>
The Manage-IR; Stream Objects

- Can have a 1-1 or Many-1 relation with memory objects
- Have a 1-1 relation with arguments to pipe functions (i.e. port connections to compute-cores)
The Manage-IR; \texttt{repeat} blocks

- Repeatedly call a kernel without referring back to the host (\textit{outer}-loop)
- May involve block memory transfers between iterations
The Manage-IR; stream windows

- Access offsets in streams
- Use on-chip buffers for storing data read from memory

```perl
%a_e = ui18 @main.a, !tir.stream.offset, !+1
%a_w = ui18 @main.a, !tir.stream.offset, !-1
%a_n = ui18 @main.a, !tir.stream.offset, !-NDim1
%a_s = ui18 @main.a, !tir.stream.offset, !+NDim1
```
The Compute-IR

- Structural semantics
  - `@function_name (...args...) par`
  - `@function_name (...args...) seq`
  - `@function_name (...args...) pipe`
  - `@function_name (...args...) comb`
  - Nesting these functions gives us the expressiveness to explore various parallelism configurations

- Streaming ports

- Counters and nested counters

- SSA data-path instructions
Example: Simple Vector Operation

The Kernel

```fortran
! ------------ the kernel --------------
! A dummy, stupid kernel with minimal operations allowing us to
! explore vectors/streaming, pipelining, multi-threading,
! Y = K (constant) + {(A + B) * (C + C)}
! where A, B, C and Y are vectors
!
! using verbose way of vector ops to expose loop
!
do n = 1,ntot
   y(n) =  K + ( (a(n)+b(n)) * (c(n)+c(n)) )
end do
```
Version 1 – Single Pipeline (C2)
Version 1 – Single Pipeline (C2)
The parser can also automatically find ILP and schedule in an ASAP fashion.
Version 2 – 4 Parallel Pipelines (C1)
Version 2 – 4 Parallel Pipelines

Stream Control

Core_Compute

add
mul
ad
d
Wr

Rd
add
add

Core_Compute

Imem a

Imem b

Imem c

Stream Control

Core_Compute

Rd
add
add

Core_Compute

Rd
add
add

Core_Compute

Rd
add
add

Core_Compute

Rd
add
add

Core_Compute

Imem y
```c
; a par block template for exposing ILP in the pipeline

define void @f1 ( ui18 %1, ui18 %2, ui18 %a, ui18 %b, ui18 %c ) par {
  ui18 %1 = add ui18 %a, %b
  ui18 %2 = add ui18 %c, %c
  ret void
}

; ** the single pipeline template function

define void @f2 ( ui18 %y, ui18 %a, ui18 %b, ui18 %c ) pipe {
  call @f1 ( ui18 %1, ui18 %2, ui18 %a, ui18 %b, ui18 %c ) par
  ui18 %3 = mul ui18 %1, %2
  ui18 %y = add ui18 %3, @k
  ret void
}

; ** top level par block that instantiates multiple pipelines

define void @f3 ( ui18 %y_01 , ui18 %y_02 , ui18 %y_03 , ui18 %y_04 ,
  ui18 %a_01 , ui18 %a_02 , ui18 %a_03 , ui18 %a_04 ,
  ui18 %b_01 , ui18 %b_02 , ui18 %b_03 , ui18 %b_04 ,
  ui18 %c_01 , ui18 %c_02 , ui18 %c_03 , ui18 %c_04 ) par {
  call @f2( ui18 %y_01 , ui18 %a_01 , ui18 %b_01 , ui18 %c_01 ) pipe
  call @f2( ui18 %y_02 , ui18 %a_02 , ui18 %b_02 , ui18 %c_02 ) pipe
  call @f2( ui18 %y_03 , ui18 %a_03 , ui18 %b_03 , ui18 %c_03 ) pipe
  call @f2( ui18 %y_04 , ui18 %a_04 , ui18 %b_04 , ui18 %c_04 ) pipe
  ret void
}

; ** Main **

define void @main () {
  call @f3 ( ui18 @y_01 , ui18 @y_02 , ui18 @y_03 , ui18 @y_04 ,
    ui18 @a_01 , ui18 @a_02 , ui18 @a_03 , ui18 @a_04 ,
    ui18 @b_01 , ui18 @b_02 , ui18 @b_03 , ui18 @b_04 ,
    ui18 @c_01 , ui18 @c_02 , ui18 @c_03 , ui18 @c_04 ) par
}
```
Version 3 – Scalar Instruction Processor (C4)
Version 3 – Scalar Instruction Processor (C4)

The ALU would be customized for the instructions mapped to this PE at compile-time.
Version 3 – Single Sequential Processor

```c
; ** top level seq block 
;
define void @f1 ( ui18 %y, ui18 %a, ui18 %b, ui18 %c) seq {
    ui18 %1 = add ui18 %a, %b
    ui18 %2 = add ui18 %c, %c
    ui18 %3 = mul ui18 %1, %2
    ui18 %y = add ui18 %3, %k
    ret void
}
;

; ** Main ** 
;
define void @main () {
    call @f1( ui18 %y, ui18 %a, ui18 %b, ui18 %c) seq
    ret void
}
```
Version 4 – Multiple Processors / Vectorization (C5)
Version 4 – Multiple Processors / Vectorization (C5)
Note the continued use of stream abstractions even through the PEs are Instruction Processors now.