NANDA: the Next Frontier

Wayne Luk
Imperial College London, United Kingdom

NANDA 2022

Acknowledgement: Jessica Vandebon, José G. F. Coutinho, Eriko Nurvitadhi, Stewart Denholm
EPSRC, SRC. AMD, Intel
Agenda

A. Context, Problem, Challenges

B. Approach: Meta-Programming Design-Flow Patterns

C. Design-Flow Pattern Catalogue For CPU and GPU Optimisation

D. Implementing Design-Flow Patterns as Meta-Programs

E. Evaluation: Automated Design-Flow Performance + Reusability

F. Ongoing Work and Big Picture
Processor Architecture: Heterogeneous Trend

A. Context, Problem, Challenges

NVIDIA GRACE HOPPER

AMD Instinct M1200
(source: Hotchips 2022)

Intel New Flexible Tile
Cloud Architecture: Heterogeneous Trend

AWS Compute Instance Types (source: AWS)
Design Automation: Support for Heterogeneity

Classical: Monolithic

- Algorithms
- High-Level Languages
- Compiler
- OS
- Architecture
- Modular hardware blocks: Gates, registers
- VLSI Circuits
- Semiconductor transistors

Current + Future: Diverse

- Algorithms
- High-Level Languages
- App-specific Approaches
- Implementation

- diverse tool-flow for high-level programs
- - compilation
- - run-time management

(adapted from: Martonosi)
Design Automation: Support for Heterogeneity

- Compiling for heterogeneous systems and processors
  - Mata-programming Design Flow Patterns
- Managing heterogeneous clouds
  - Function-as-a-Service
- Managing heterogeneous FPGA resources
  - Pool of functions
A. Context, Problem, Challenges

Design Automation: Support for Heterogeneity

- Compiling for *heterogeneous systems and processors*

- Managing *heterogeneous clouds*
  - Function-as-a-Service - Journal of Signal Processing Systems

- Managing *heterogeneous FPGA resources*
  - Pool of functions - FPL 2022
Heterogeneous Systems

- compute landscape is rapidly evolving → increasingly **parallel** and **heterogeneous**
  - potential of specialised accelerators (GPUs, FPGAs) for demanding applications, e.g. AI, HPC

- gap between software descriptions and optimised heterogeneous designs: getting larger
  - device-specific compilers: achieve high performance from high-level source-code
  - *but* significant code restructuring is required
Problem

- heterogeneous application optimisation: typically done manually
  - requires highly-skilled developers with in-depth target hardware understanding
- manual optimisation tasks:
  - identify computationally intensive hotspots
  - partition and map code across processing elements
  - annotate parallel computations
  - apply known datatype, throughput, and memory optimisations
  - systematic diagnosis to tune runtime parameters

⇒ this process is tedious, error-prone, and must be repeated for each new application
Current Design-Flow: State Of The Practice (SOP)

- current SOP: human developers **manually** perform source-level design-flows

- **design-flow:** explicit orchestration of manual and/or automated tasks
  - map and optimise a high-level software description onto hardware
Design-Flow Automation Challenges

C1. **Abstraction**: diverse components should be abstracted to hide implementation details
   - so they can be employed by non-experts

C2. **Efficiency**: automatically optimised code should be as efficient as manually optimisation
   - currently requires expertise, experience, and effort

C3. **Customisability**: automated design-flows should be flexible and extensible
   - support new techniques and technologies in the massive, evolving design space

C4. **Reusability**: design-flows should employ existing, reusable components
   - reduce time and development effort

C5. **Application-Agnosticity**: automated design-flows should operate on multiple applications
   - within a specific application domain
Contributions

➔ **design-flow patterns** to capture common and recurring elements of design-flows
  - for optimising high-level descriptions onto diverse hardware targets

➔ an initial catalogue of patterns
  - for accelerating CPU and GPU designs

➔ codify modular patterns as **Artisan meta-programs**
  - combine target-independent and -dependent patterns into automated design-flows
  - map unmodified sequential C++ descriptions into optimised CPU and GPU designs

➔ apply our design-flows to:
  - 3 case-study HPC applications in different domains (physics, graphics, mathematics)
  - evaluate performance of automatically generated OpenMP and HIP designs

➔ results:  
  - up to 18 times speedup on a CPU platform with 32-threads
  - up to 1184 times speedup on an NVIDIA GeForce RTX 2080 Ti GPU

**compared to a sequential single-threaded reference implementation**
Key Observation

- design-flows for diverse hardware targets often involve:
  - common, recurring, application-agnostic elements
  - elements that can be target and tool-independent or tool-dependent

⇒ can we capture and codify these recurring building blocks, highlighting the **branch points** for introducing diverse designs?
Proposed Solution: Meta-Programming Design-Flow Patterns

- **Design-Flow Patterns:**
  - capture, catalogue, and codify common and recurring design-flow tasks
  - for building customised, reusable, automated design-flows

- similar to design patterns:
  - abstract recurring solutions
  - provide reusable base of experience and a common vocabulary

- modular patterns implemented as meta-programs
  - can be coordinated into automated end-to-end design flows
Two Design-Flow Roles

(1) **application developer**: writes functionally correct high-level application description
B. Approach: Meta-Programming Design-Flow Patterns

Two Design-Flow Roles

(1) **application developer**: - writes functionally correct high-level application description

(2) **design-flow developer**: - uses design-flow patterns and meta-programs
- to automate design-flows for mapping and optimisation
B. Approach: Meta-Programming Design-Flow Patterns

Two Design-Flow Roles

Artisan meta-programs treat programs as data, enabling programmatic analysis and source-code manipulation.

- **Application developer**: writes functionally correct high-level application description

- **Design-flow developer**: uses design-flow patterns and meta-programs to automate design-flows for mapping and optimisation

(1) **Application developer**: - writes functionally correct high-level application description

(2) **Design-flow developer**: - uses design-flow patterns and meta-programs to automate design-flows for mapping and optimisation
B. Approach: Meta-Programming Design-Flow Patterns

Two Design-Flow Roles

Artisan meta-programs treat programs as data, enabling programmatic analysis and source-code manipulation

⇒ manual design-flow tasks are codified and coordinated to produce:
  - end-to-end design-flows operating on high-level software descriptions
  - optimised designs with little intervention from application developers
Addressing Design-Flow Automation Challenges

C1. **Abstraction**: high-level pattern descriptions
   - abstract implementation details for diverse targets

C2. **Efficiency**: source-to-source meta-programs
   - automate manual optimisation with static and dynamic analysis

C3. **Customisability**: pattern implementations as plug-and-play building blocks
   - can be parameterised, replaced, and extended

C4. **Reusability**: modular design-flows facilitate patterns
   - implemented once and reused in multiple design-flows (e.g. common analysis)

C5. **Application-Agnosticity**: optimisation is decoupled from application descriptions
   - so design-flows are application-agnostic
Design-Flow Pattern Catalogue: Overview

- current catalogue contains patterns for CPU and GPU parallel targets
- requirement: facilitate modular implementations and reasoning about coordination
- a uniform template* is used to describe design-flow patterns:
  - NAME: a succinct, descriptive name for the pattern
  - INTENT: what does the pattern do?
  - MOTIVATION: why is the pattern used?
  - APPLICABILITY: what conditions must be met to apply the pattern?
  - RELATED PATTERNS (OPTIONAL): are there related patterns? (e.g. components, often used together)

- text-based description should clearly capture intent and applicability
  - developers can unambiguously codify expected behaviour
  - ongoing work: formalise design-flow pattern specification

*a subset of design patterns for OOP from Gamma et al.
C. Design-Flow Pattern Catalogue For CPU and GPU Optimisation

Design-Flow Pattern Classification

- 4 types of design-flow patterns:
  
  I. *Analysis patterns*: perform static or dynamic app analysis
  
  II. *Code-generation patterns*: inject or generate new source-code
  
  III. *Transform patterns*: perform source-to-source transformation
  
  IV. *Optimisation patterns*: employ analysis and transform patterns
     - optimise a target metric (typically involving Design Space Exploration)
Analysis Design-Flow Pattern Example

- **NAME**: HOTSPOT LOOP DETECTION
- **INTENT**: identify computationally intensive parallel loops to accelerate
- **MOTIVATION**: - loops are often where most time is spent during execution
  - suitable for acceleration (Amdahl’s law)
- **APPLICABILITY**: applicable to any application source code
- **RELATED PATTERNS**: Loop Timing, Dependence Analysis
**Code-Generation Design-Flow Pattern Example**

- **NAME**: *HIP GPU MANAGEMENT CODE GENERATION*
- **INTENT**: insert code required to execute an identified kernel function on GPU
- **MOTIVATION**: device management code is required
  - to inform the runtime system what to run on the GPU vs CPU
  - to ensure data are where they need to be for application execution
- **APPLICABILITY**: applicable to application code with a specified kernel function
Transform Design-Flow Pattern Example

- **NAME**: SHARED MEMORY BUFFER
- **INTENT**: - copy the contents of a pointer argument
  - into shared memory in a GPU kernel
- **MOTIVATION**: on-chip shared memory
  - has limited size
  - has higher bandwidth and lower latency than global memory
- **APPLICABILITY**: applicable to any pointer argument for a GPU kernel
  - if pointer contents fit in shared memory
Optimisation Design-Flow Pattern Example

- **NAME**: TUNE KERNEL LAUNCH PARAMETERS
- **INTENT**: determine the kernel launch parameters
  - to minimise execution time, and/or
  - to maximise occupancy (e.g. block size)
- **MOTIVATION**: launching kernels with different thread configurations
  - can affect execution time and GPU occupancy
- **APPLICABILITY**: applicable to an application source with a GPU kernel
- **RELATED PATTERNS**: Set Blocksize, Kernel Timing, Calculate GPU Occupancy
C. Design-Flow Pattern Catalogue For CPU and GPU Optimisation

Table 1: Analysis (A1-A5), Code-Generation (G1-G2), Transform (T1-T9) and Optimisation (O1-O2) Design-Flow Patterns

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME (RELATED)</th>
<th>INTENT</th>
<th>MOTIVATION</th>
<th>APPICABILITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Hotspot Loop</td>
<td>Identify computationally intensive loops to accelerate.</td>
<td>Loop is often a region that most time is spent in the program's execution.</td>
<td>Application code</td>
</tr>
<tr>
<td>A2</td>
<td>Loop Tuning</td>
<td>Measure execution time for all loops in the application.</td>
<td>To identify application bottlenecks and regions worth optimizing.</td>
<td>Application code</td>
</tr>
<tr>
<td>A3</td>
<td>Dep. Analysis</td>
<td>Identify dependencies in a program.</td>
<td>To parallelise and transform loops.</td>
<td>Loop</td>
</tr>
<tr>
<td>A4</td>
<td>Pointer Analysis</td>
<td>Determine if pointer arguments could alias within a function scope.</td>
<td>Certain compiler optmisations can only be applied if it is indicated that pointers do not alias.</td>
<td>Function definition</td>
</tr>
<tr>
<td>A5</td>
<td>Kernel Tuning</td>
<td>Time all GPU kernels in an event application.</td>
<td>To understand the impact of code changes, identify bottlenecks and compare performance.</td>
<td>Application code + (GPU kernel)</td>
</tr>
<tr>
<td>A6</td>
<td>Calculate GPU Occupancy</td>
<td>Determine the occupancy of a kernel on a target GPU.</td>
<td>Calculating occupancy helps to understand performance and tune GPU launch parameters.</td>
<td>Application code + (GPU kernel)</td>
</tr>
<tr>
<td>C1</td>
<td>Loop-to-Function Extraction</td>
<td>Extract a program loop into an isolated function.</td>
<td>To enable analysis and annotation to indicate it should be offloaded to an accelerator.</td>
<td>Loop</td>
</tr>
<tr>
<td>C2</td>
<td>Multi-Threaded Code Generation</td>
<td>Insert the framework-specific code required to multi-thread a loop.</td>
<td>Loop annotation, header file inclusion, and runtime parameter specification is encoded for runtime system to use multiple parallel threads.</td>
<td>Application code + Loop</td>
</tr>
<tr>
<td>C3</td>
<td>GPU IMGt Code Generation</td>
<td>Insert the framework-specific code required to execute a kernel on a GPU.</td>
<td>Device management code is required to inform the runtime system what to run on the GPU vs CPU and to ensure data was already available.</td>
<td>Application code + Function</td>
</tr>
<tr>
<td>T1</td>
<td>Restrict Pointer Arguments (A4)</td>
<td>Indicate to the compiler that pointer arguments do not alias.</td>
<td>Device compiler that cannot determine if pointers could alias conservatively assume that they might, limiting the scope for optimisation.</td>
<td>Non-aliasing function args + target with restricted keyword</td>
</tr>
<tr>
<td>T2</td>
<td>Shared Memory Buffer</td>
<td>Copy the contents of a pointer argument into shared memory in a GPU kernel.</td>
<td>Limited shared memory has higher bandwidth and lower latency than global memory.</td>
<td>App code + GPU kernel target with page-locked memory</td>
</tr>
<tr>
<td>T3</td>
<td>Page-Locked Memory</td>
<td>Allocate memory as page-locked.</td>
<td>Limited page-locked memory has the highest bandwidth between host and device, but higher weight allocations than regular memory</td>
<td>App code + GPU kernel target with page-locked memory</td>
</tr>
<tr>
<td>T4</td>
<td>Single-Precision Math Functions</td>
<td>Use single-precision versions of math functions (e.g. sqrtf).</td>
<td>Avoid implicit intermediate rounding to double-precision operations.</td>
<td>GPU kernel + library math function call</td>
</tr>
<tr>
<td>T5</td>
<td>Single-Precision FP Literals</td>
<td>Employ single-precision floating point literals.</td>
<td>Explicitly use single precision literals (e.g. 0.0, 0.5) so compiler does not assume double precision.</td>
<td>Expansions with single-precision types</td>
</tr>
<tr>
<td>T6</td>
<td>Specialised Math Operations</td>
<td>Use available specialised math operations.</td>
<td>Framework-provided specialised math functions are more optimised than generic equivalents.</td>
<td>Consult tool documentation (e.g. pow(x, 2) to exp2(x))</td>
</tr>
<tr>
<td>T7</td>
<td>Remove Loop Dep (A3)</td>
<td>Remove dependent array accesses in loops by introducing intermediate variables.</td>
<td>To ease loop dependency bottlenecks.</td>
<td>Loops with dependent array accesses</td>
</tr>
<tr>
<td>T8</td>
<td>Set Blocksize</td>
<td>Specify the thread block size for GPU kernel execution.</td>
<td>Run-time GPU thread configurations must be set when launching a kernel.</td>
<td>GPU kernel</td>
</tr>
<tr>
<td>T9</td>
<td>Set Num Threads</td>
<td>Set the number of parallel threads for loop execution.</td>
<td>To control the number of threads used for multi-threaded execution.</td>
<td>Loop + multi-threaded target</td>
</tr>
<tr>
<td>O1</td>
<td>Tune Number of Threads (T1A2)</td>
<td>Determine the number of threads that minimises loop execution time.</td>
<td>The number of threads can affect performance depending on available cores and workload size</td>
<td>Loop(s) + multi-threaded target</td>
</tr>
<tr>
<td>O2</td>
<td>Tune Kernel Launch (T1A5, A6)</td>
<td>Determine the kernel launch parameters that minimises execution time and/or maximises occupancy.</td>
<td>Application code + (GPU kernel)</td>
<td></td>
</tr>
</tbody>
</table>

- refer to our HERRT’22 paper for the full catalogue
- a starting point to demonstrate scope and value: recurring, application-agnostic design-flow
- not an exhaustive list of GPU/CPU patterns
Design-Flow Patterns as Meta-Programs

- codify patterns using the Artisan meta-programming framework
  - based on libclang, supports C++ parsing and manipulation
  - unified Python environment for code analysis, instrumentation, and execution
  - true source-to-source: no progressive lowering
- key Artisan features
  - *query* and *instrument*
    - enables static source-code analysis and manipulation
  - application *execution* and runtime *reporting*
    - enables application self-reporting for dynamic analyses
D. Implementing Design-Flow Patterns as Meta-Programs

Design-Flow Patterns as Meta-Programs

- example meta-programs:
  1. GPU shared memory buffer (transform)
  2. parallel hotspot loop detection (dynamic analysis)

- for more details on Artisan, refer to our paper in *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2043-2055, 1 Dec. 2021

*Enhancing High-Level Synthesis using a Meta-Programming Approach*

Jessica VandeBon*, Jose G. F. Coutinho*, Wayne Luk*, Eriko Nurvitadhi†

*Imperial College London, United Kingdom
Email: {jessica.vandeBon17, gabriel.figueiredo, w.luk}@imperial.ac.uk
†Intel Corporation, San Jose, USA
Email: eriko.nurvitadhi@intel.com

Abstract—In today’s increasingly heterogeneous compute landscape, there is high demand for design tools that offer seemingly contradictory features: portable programming abstractions that hide underlying architectural detail, and the capability to optimise and exploit architectural features. Our meta-programming approach, Artisan, decouples application functionality from optimisation concerns to address the complexity of mapping high-level application descriptions onto heterogeneous platforms from which they are abstracted. With
Hotspot Detection

D. Implementing Design-Flow Patterns as Meta-Programs

```python
1 def identify_hotspots(ast, threshold):
2     # clone ast for instrumentation & execution
3     ast_clone = ast.clone()
4     # query for parallel for-loops to time
5     par_loops = ast_clone.query("loopStmt", where=lambda loop: is_par(loop))
6     # instrument loops and main function with timers
7     instrument_app_timer(ast_clone, par_loops)
8     # execute instrumented code and receive report
9     report = ast_clone.exec(reports=True)
10    # discard clone
11    ast_clone.discard()
12    # extract main timing from report (e.g. main_t = 404.9)
13    main_t = report['main']; del report['main']
14    # filter and return loop list that satisfies given threshold
15    hotspots = [loop for loop in report[loop] > main_t * threshold]
16    # in our example, returns ['loop0312']
17    return hotspots
```

```
1  #include <artisan>
2  using namespace artisan;
3  int main(int argc, char *argv[]){
4      Report::start();
5      int ret;
6      { Timer timer_main([](double t){
7          Report::write("'main':%f", t);
9        ret = [](auto argc, auto argv){
10        { Timer timer_ltag([](double t){
11            Report::write("'loop0312','%f'"), t);
13            for (int i = 0; i < N; i++) {
15            z[i] = x[i] * y[i];
17            }...
18            for (int j=0; j<T; j++) {
20            z[j] = x[j] * z[j-1];
22            }...
24            return 0;
25            })(argc, argv);
27        } Report::emit();
30    } report sent via network socket to metaprogram
32    return ret;
33 }
34 Instrumented app (C++)
```

Artisan meta-program (Python)

仪器化程序 (C++)

报告通过网络套接字发送到元程序

此循环不平行，因此未仪器化
Automated End-To-End Design Flows

- implemented end-to-end HIP GPU and OpenMP CPU design-flows
  - comprised modular meta-programs codifying patterns from our catalogue

- applied to three HPC case-study applications:
  - N-Body Simulation (physics),
  - Bezier Surface Generation (graphics),
  - Rush Larsen ODE Solver (maths)
Evaluating Design-Flow Pattern Reuse

- 20 patterns implemented
  - 10 employed by OpenMP design-flow
  - 17 employed by HIP GPU design-flow

E. Evaluation: Automated Design-Flow Performance + Reusability
Evaluating Design-Flow Pattern Reuse

- 20 patterns implemented
  - 10 employed by OpenMP design-flow
  - 17 employed by HIP GPU design-flow
- 7/20 patterns shared by both design flows
Evaluating Design-Flow Pattern Reuse

- 20 patterns implemented
  - 10 employed by OpenMP design-flow
  - 17 employed by HIP GPU design-flow
- 7/20 patterns shared by both design flows
- 17/20 patterns applicable to all three case-study applications
Evaluating Design-Flow Pattern Reuse

- 20 patterns implemented
  - 10 employed by OpenMP design-flow
  - 17 employed by HIP GPU design-flow
- 7/20 patterns shared by both design flows
- 17/20 patterns applicable to all three case-study applications
- 20/20 patterns are application agnostic
Evaluating Design-Flow Performance

OpenMP CPU Experiments:

- experimental set-up:
  - 2 Intel Xeon Silver 4110 CPUs, 16 cores with SMT
  - g++ -O2
  - consider 8, 16, 32 available threads

- performance results:
  - generally: increasing threads decreases execution time (nonlinear due to scheduling/mgmt overhead)
  - above 16 threads: speedup limited by SMT support
  - 12X-18X maximum speedup across case-studies

Figure 6: Performance of multi-threaded CPU and HIP GPU designs generated by automated Artisan design-flows compared to the input unoptimised sequential implementation (single-threaded).
Evaluating Design-Flow Performance

**HIP GPU Experiments:**

- **experimental set-up:**
  - 2 NVIDIA GeForce GPUs:
    - NVIDIA GeForce GTX 1080 Ti
    - NVIDIA GeForce RTX 2080 Ti
  - `hipcc -O2`

- **performance results:**
  - generally: RTX 2080 faster than GTX 1080 (wider cores with advanced features)
  - **87X-1184X** maximum speedup across case-studies

---

**Figure 6:** Performance of multi-threaded CPU and HIP GPU designs generated by automated Artisan design-flows compared to the input unoptimised sequential implementation (single-threaded).
Evaluating Design-Flow Performance

→ performance comes free
- little or no intervention from application developer

→ generated code is human-readable
- same level of abstraction as original code
- can be further hand-tuned

Figure 6: Performance of multi-threaded CPU and HIP GPU designs generated by automated Artisan design-flows compared to the input unoptimised sequential implementation (single-threaded).
Ongoing and Future Work

➔ formalising the specification and description of design-flow patterns
  ◆ using functional programming

➔ extending our design-flow pattern catalogue:
  ◆ FPGA OneAPI mapping and optimisation patterns
  ◆ patterns to support more advanced GPU optimisations
  ◆ application-domain specific patterns
Big Picture: Automating Design

design space exploration, goals and constraints

partition, compile

system-specific programming interface

system-specific adaptation: clouds to edge devices

Application description

optimisation/synthesis

Compiler

Machine code
Run-time interface
Configuration information

Fixed processor
Custom processor

Custom computing system
Big Picture: Automating Design + Debug + Verify

design space exploration, goals and constraints
partition, compile, analysis, verify
system-specific programming interface
system-specific adaptation: clouds to edge devices

Application description
optimisation/synthesis
analysis/verify

Compiler
Machine code
Run-time interface
Configuration information

Fixed processor
Custom processor
Custom computing system

Meta-program

FCCM 2021: Flexible Instrumentation for Live On-Chip Debug of Machine Learning Training on FPGAs
JSA 2021: In-Circuit Tuning of Deep Learning Designs