Heron: Modern Hardware Graph Reduction

Craig Ramsay
Heriot-Watt University
Edinburgh, Scotland
craig.ramsay@hw.ac.uk

Robert Stewart
Heriot-Watt University
Edinburgh, Scotland
r.stewart@hw.ac.uk

ABSTRACT

FPGAs have enjoyed exponential growth of on-chip hardware resources — reason to reinvestigate *hardware* implementations of functional languages. This paper presents *Heron*, an FPGA-based special purpose processor core for pure, non-strict functional languages. We co-design its language semantics and parametrised design, gaining a high reductions-per-cycle performance metric. The Heron core is energy efficient, performing up to six times as many reductions per cycle as GHC. Despite its infancy, a 193 MHz Heron core outperforms wall-clock time for a mid-range Intel i3 1.9 GHz mobile CPU for 5 of these benchmarks and is competitive with an Alder Lake Intel i7 CPU. Its performance-per-Watt shows that the Heron core is a compelling solution for embedded applications. The simplicity of Heron's design results in just 2% FPGA resource usage, paving the way for future single-chip parallelism, further improving absolute performance.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; • **Computer systems organization → Architectures**; Multicore architectures.

KEYWORDS

hardware accelerators, functional languages, graph reduction, FPGAs

ACM Reference Format:
Craig Ramsay and Robert Stewart. 2023. Heron: Modern Hardware Graph Reduction. In *The 35th Symposium on Implementation and Application of Functional Languages (IFL 2023), August 29–31, 2023, Braga, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3652561.3652564

1 INTRODUCTION

Functional language implementations overwhelmingly target fixed CPU architectures. Stagnating clock frequencies in the mid 2000s sparked research into alternative speedup techniques such as exploiting parallelism, locality, and compiler heuristics. The semantic gap between functional languages and CPU assembly languages imposes limits on what these techniques can achieve. The conceptual mismatch between high-level functional execution models and low-level CPU instruction sets necessitates compiler transformations (Section 2.2). Continued exponential growth of FPGA logic density, including wide memories with multiple independent ports, rejuvenates 1980s questions about efficient functional language implementations using custom hardware. We argue that *co-designing language semantics and hardware closer to functional execution models will produce more performant systems versus general purpose CPUs*. We believe that co-designing graph reduction hardware architectures as custom logic enables three avenues for substantial progress:

1. *Low-level parallelism* within single β-reductions in the λ-calculus — not easily exploited on conventional CPUs due to the memory bottleneck caused by allocations for immutable data structures and *thunks* [1, 14].
2. Embedding runtime system tasks (e.g. garbage collection) as hardware units, running concurrently to reduction. More useful work is performed per cycle.
3. Exploiting the purity of functional languages by safely executing multiple reductions simultaneously.

Benefits 1 and 2 both significantly cut the number of required clock cycles, reducing energy consumption for carbon-efficient computing. This paper focuses on the first idea: the design of a single, sequential reduction core with good low-level parallelism. We hope to address benefits 2 and 3 in immediate future work. The contributions of this paper are:

- The co-design of Heron's native language and custom hardware architecture (Section 2).
- Three optimisations to Heron's language semantics resulting in decreases of 5.6% to clock cycles, 6.3% to heap allocations, and 22% to code size (ignoring positive outliers). Optimising Heron for a modern FPGA almost doubles the clock frequency versus a baseline Reduceron processor[13] and requires 1.88% of hardware resources versus 90% for Reduceron an older generation FPGA (Section 3).
- An evaluation of Heron's space and time performance trade-offs, and time and power performance comparisons against GHC running on embedded, desktop and high performance CPUs (Section 4).

2 GRAPH REDUCTION TECHNIQUES

Graph reduction implements lazy evaluation, where function arguments are not evaluated before the function body. There is a spectrum of graph reduction implementations, ranging from fixed general purpose CPUs (Section 2.2) to truly custom circuits (Section 2.3)[1]. Heron's foundations share ideas and its baseline operational semantics (Section 2.3.2) with the Reduceron project. Since the baseline Heron core is a faithful Reduceron re-implementation, Section 2.1 begins with a summary of its limiting factors, which are ameliorated by our own contributions in Sections 3 and 4.

[1]The authors provide a curated history of functional architectures at https://haflang.github.io/history.html

IFL 2023, August 29–31, 2023, Braga, Portugal

ACM ISBN 979-8-4007-1631-7/23/08
https://doi.org/10.1145/3652561.3652564

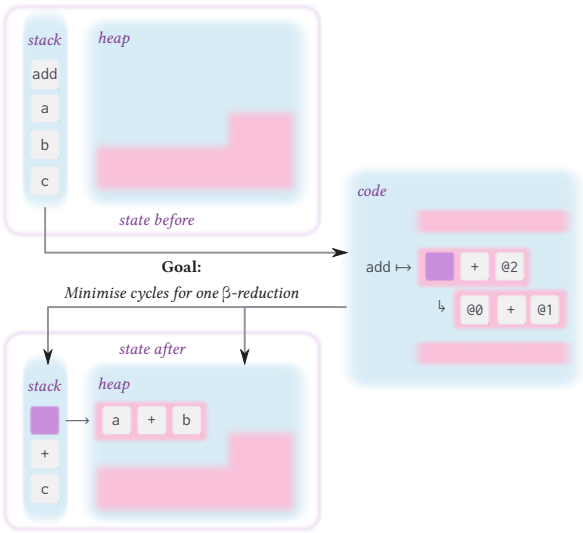


Figure 1: Abstract reduction machine state performing function application to `add x y z = (x + y) + z`

2.1 The Key Observation

The abstract graph reduction machine in Fig. 1 shows how function application can be performed by constructing fixed graph structures in stack and heap memories. These memories are accessed and updated according to the position of λ-bound variables in a function body. Narrow memory access on von Neumann CPU architectures sequentialises the many memory operations implicit in the function application in Fig. 1 into multiple CPU instructions (Fig. 2), and therefore multiple clock cycles. GHC uses compiled graph reduction to make this efficient (Section 2.2). However, narrow memory access on a single serial bus remains a fundamental bottleneck for software-based graph reduction implementations.

Reduceron showed that hardware implementation on a *single chip* is a promising avenue for graph reduction machines, when *very wide memory interfaces* can be custom made to parallelise the memory access patterns of a β-reduction. Unlike GHC's compiled graph reduction approach, Reduceron used simple template instantiation. It employs multiple wide, parallel memories to often perform these function applications in a single clock cycle. Our own implementation of these ideas performs function application with single-cycle access to more than 1 kb of data bus bits (Section 4.1.2), rather than the 64 bits common to stock hardware. The simplicity allows architecture choices to be easily evolved with a relatively small development effort and results in relatively low circuit areas — important for scaling up to multi-core implementations. However, Reduceron reached two hard ceilings enforced by the contemporary (pre-2012) FPGA devices:

Circuit area, especially in terms of memory resources.
Circuit timing, constrained by template instantiation.

These factors lead to a mixed conclusion in otherwise exciting research: "The Reduceron is on average 4–5 times slower than conventionally-compiled code running on a desktop PC" [13]. The design required 90% of FPGA BlockRAM resources — the most dense memory available in Virtex-5 devices. Limited memory capacity precluded efforts to explore parallel Reduceron architectures. Moreover, the simplicity of template instantiation (and the low-level parallelism therein) is counterbalanced by its resilience to pipelining techniques — the behaviour in each cycle is determined by the stack state from the previous cycle. This resulted in Reduceron's 96 MHz clock frequency, fairly low by modern FPGA standards.

The key insight of this paper is that these ceilings can be dramatically raised with new techniques and modern FPGA architectures. In particular, Section 3 demonstrates four new techniques that, when combined, arrive at an architecture with under 2% resource utilisation and double the clock frequency. These results stem from our *active* set of four incremental improvements to the architecture, and *passive* improvements enjoyed by riding the wave of exponential FPGA growth — analogous to how early compiled graph reduction techniques passively benefited from the rapid development of RISC processors.

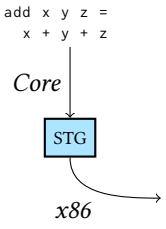
2.2 Compiled Graph Reduction

GHC is today's canonical software implementation of graph reduction. To implement non-strictness, it transforms Haskell code into Spineless Tagless G-machine (STG) [8] instructions for von Neumann architectures. This results in basic code blocks and heap representations of closures, e.g. Fig. 2 shows GHC's x86_64 output for a simple function without optimisation. This complexity contrasts to the direct, single-cycle function application achieved by Reduceron. Software implementations have little choice but to use program transformations like STG or the G-Machine [11]. Compilers try to efficiently map functional code into CPU architectures using heuristics [6, 7] in the absence of a precise model of specialised hardware. An optimising compiler can mask some of the reduction overhead, but it still offers limited locality control within a CPU's memory hierarchy. Performance can suffer from memory contention and cache misses [14]. Reliance on indirect jumps can also damage pipeline utilisation.

Many specialised hardware architectures in the 1980s used stock processors for performing reductions, e.g. Motorola 68020 microprocessors in GRIP [9], so these often used programmed reduction. More recently, PilGRIM [4] moved towards the custom logic approach with an instruction set tailored for programmed graph reduction. It therefore shared the fetch-decode-execute characteristic of CPUs. The authors had ambitions for a pipelined implementation to hide this latency. Whilst this could result in a design with high clock speeds, it would be at the cost of a highly complex circuit.

2.3 Hardware for Pure Graph Reduction

Heron, PilGRIM, and Reduceron all borrow a key idea from the Big Word Machine (BWM) [1]. They realise low-level parallelism by operating over multiple values from wide memories and a primary stack with a large crossbar interconnect. Heron and Reduceron are at the extreme end of the hardware spectrum, with truly custom reduction logic. Our choice of template instantiation is precisely to forgo the STG-style translation to a sequential instruction stream. Now, both memory accesses *and* reductions happen, in parallel, in the same cycle. The main drawback of template instantiation is that circuits prove resistant to many traditional pipelining techniques,




```
s1ba_info: ; Code for helper (\a b -> a+b)
.Lc1bo: ; Check for stack space
   leaq -40(%rbp),%rax
   cmpq %r15,%rax
   jb .Lc1bp ; Jump if stack full
.Lc1bq: ; Reduce helper
   movq $stg_upd_frame_info,-16(%rbp)
   movq %rbx,-8(%rbp)
   movq 16(%rbx),%rax ; Load a & b from heap
   movq 24(%rbx),%rbx
   movl $base_GHCziNum_zdfNumInt_closure,%r14d
   ;; Push `a+b` onto stack
   movq $stg_ap_pp_info,-40(%rbp)
   movq %rax,-32(%rbp)
   movq %rbx,-24(%rbp)
   addq $-40,%rbp
   jmp base_GHCziNum_zp_info ; Enter
.Lc1bp: ; Ask RTS for stack space
   jmp *-16(%r13)

Add_add_info: ; Code for `add`
.Lc1br: ; Check for stack space
   leaq -24(%rbp),%rax
   cmpq %r15,%rax
   jb .Lc1bs ; Jump if stack full
.Lc1bt: ; Check for heap space
   addq $32,%r12
   cmpq 856(%r13),%r12
   ja .Lc1bv ; Jump if heap full
.Lc1bu: ; Reduce `add`
   ;; Build `x+y` thunk on heap
   movq $s1ba_info,-24(%r12)
   movq %r14,-8(%r12)
   movq %rsi,(%r12)
   leaq -24(%r12),%rax
   movl $base_GHCziNum_zdfNumInt_closure,%r14d
   ;; Push `thunk+z` to stack
   movq $stg_ap_pp_info,-24(%rbp)
   movq %rax,-16(%rbp)
   movq %rdi,-8(%rbp)
   addq $-24,%rbp
   jmp base_GHCziNum_zp_info ; Enter
.Lc1bv: ; Ask RTS for heap space
   movq $32,904(%r13)
.Lc1bs: ; Ask RTS for stack space
   movl $Add_add_closure,%ebx
   jmp *-8(%r13)
```

Figure 2: Compiling a simple Haskell function to x86 CPU assembly language, using closures, info tables, and explicit stack accounting. Performs a state transition equivalent to Fig. 1

placing a ceiling on single-thread performance. This motivates our focus on alternative techniques with Heron's four optimisations to Reduceron (Section 3).

2.3.1 A Hardware-Amenable Template Language. The tool flow from a user's source code (in F-lite; Reduceron's non-strict functional language) to the Heron core hardware is shown in Fig. 3, and is contrasted to the equivalent process for GHC and stock CPUs. The only intermediate representation used by Heron is a template language, constructed by a simple compilation step, discussed below.

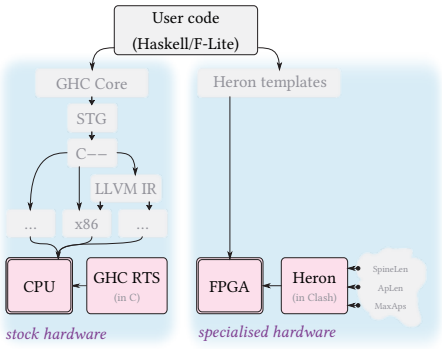


Figure 3: Compiler flows for CPU and Heron cores

Heron's assembly language is a non-strict functional language, with syntax shown in Fig. 4. The metavariables α, n, and s are used for arities, integers, and "possibly shared" flags. A program is a series of supercombinators described as *Heron graph templates*. Each template is an expression graph in an administrative-normal form. The language supports data constructors (eliminated by case expressions), and primitive integers (eliminated by primitive operations). The simple F-lite compiler [13] translates F-lite sources (an untyped subset of Haskell with case expressions, uniform pattern matching, and let-bindings) to this template format.

The primitive operations includes seq to force evaluation of a primitive and the binary operations (+), (−), (=), (≠), and (≤). Case expressions are represented using *case tables*, where each alternative is lifted to its own template placed contiguously in the program. The case's subject expression is placed in a CASE

$e ::=$	Atoms
FUN $s\ \alpha\ n$	Function pointer
| ARG $s\ n$	Argument pointer
| VAR $s\ n$	Application pointer
| CON $\alpha\ n$	Constructor tag
| INT n	Integer literal
| PRI $\alpha\ \otimes$	Primitive operation
| REG n	Primitive register pointer
$c ::=$ TAB n	Case table pointer
$u, v ::=$	Applications
APP $\overline{e}$	Normal application
| CASE $c\ \overline{e}$	Application with case table
| PRIM $n\ \overline{e}$	PRS candidate allocation
$t ::=$	Template
FUN $s\ \alpha\ n\ =$ let $\overline{u}$ in v	
$p ::= \overline{t}$	Program

Figure 4: Syntax of Heron templates

application, which indicates the location of the case table. Once evaluated, the subject's constructor tag is used to select one of the alternatives from the table.

Note the separate handling of the four pointer constructs for functions, arguments, applications, and case tables. This arises from a major opportunity of specialised hardware — each of the four memories are truly independent and can be accessed concurrently during a *single cycle*. There is also a stack for tracking to-be-updated nodes in the graph.

As a concrete example, consider the fromTo function below, which constructs a list of integers from n to m (inclusive).

```
fromTo n m = case n <= m of {
  False -> Nil;
  True  -> Cons n (fromTo (n + 1) m);
};
```

First, the two case alternatives are lifted out to their own top-level function definitions: fromTo#F and fromTo#T. Note that they bind additional arguments, preventing free variable references in

any alternative expression. The case's subject is now applied to a case table, referencing the two alternatives.

```
fromTo   n m = (n <= m) <fromTo#F,fromTo#T> n m;
fromTo#F n m = Nil;
fromTo#T n m = Cons n (fromTo (n + 1) m);
```

Next, an A-normal form is obtained by lifting subexpressions to their own let-bound names.

```
fromTo n m   = let { a = n <= m; }
               in  a <fromTo#F,fromTo#T> n m;
fromTo#F n m = Nil;
fromTo#T n m = let { a = n + 1;
                    b = fromTo a m; }
               in  Cons n b;
```

This leaves a form with 1:1 equivalents in the Heron Core's native template language. Function, argument, and variable names are replaced with indices, and all applications and atoms are further annotated. All arguments and variable references are assigned a boolean flag (⊤/⊥) identifying the reference as possibly shared or locally non-shared. All function references and primitive operations carry their arities. Listing 1 shows the final template representation in full.

```
FUN ⊤ 2 0 =                                              (fromTo)
   let    APP [   ARG ⊤ 0,    PRI 2 <=,     ARG ⊤ 1  ]
    in    CASE (TAB 1)
             [   VAR ⊥ 0,     ARG ⊤ 0,      ARG ⊤ 1  ]
FUN ⊤ 2 1 =                                            (fromTo#F)
   let    ∅
    in    APP [    CON 0 1  ]
FUN ⊤ 2 2 =                                              (fromTo#T)
   let    APP [   ARG ⊤ 0,     PRI 2 +,       INT 1  ]
          APP [ FUN ⊤ 2 0,     VAR ⊥ 0,      ARG ⊥ 1  ]
    in    APP [    CON 2 0,    ARG ⊤ 0,      VAR ⊥ 1  ]
```

Listing 1: A template representation for `fromTo`

To facilitate a hardware implementation, template dimensions are bounded by the Heron core's architectural parameters. The three most fundamental dimensions are:

- `SpineLen`: Length of a spinal application (the scope of the let expression).
- `ApLen`: Length of a let-bound application.
- `MaxAps`: Number of let-bindings per template.

`SpineLen` and `ApLen` are not necessarily equal since the heap and stack memories are implemented independently (and accessible concurrently), as discussed later in Section 3. `SpineLen` dictates how parallel the stack implementation must be, while `MaxAps` sets the parallelism of the heap. `ApLen` contributes to the heap memory geometry. One way to visualise the consequence of these three dimensions is shown in Fig. 5 using an expression graph directly.

For completeness, Fig. 4 also introduced an optional optimisation via the `REG` and `PRIM` constructs. This facilitates the compatibility with Reduceron's Primitive-Redex Speculation (PRS). With PRS, let-bound primitive applications may be *evaluated* during their instantiation, precluding the need for their allocation on the heap and later dereferencing. Candidate applications for this process are stored as `PRIM` applications rather than an `APP`. This specifies a destination register for a fully-evaluated result. If any of the operands are unevaluated then this optimisation does not apply, and the application is instantiated as usual.

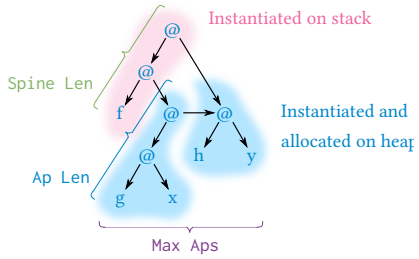


Figure 5: An example expression graph grouped by template resources and dimensions

2.3.2 An Operational Semantics. This section introduces an operational semantics for the baseline Heron core. These rules are a reimplementation of Reduceron's with superficial changes and the addition of their incremental refinements (case table stack, dynamic update avoidance, infix primitive operations, and primitive-redex speculation) included as standard. We continue with the same presentation style, making heavy use of Haskell code. This style is particularly appealing for the Heron core since it very closely aligns with the final hardware description written in Clash — approximating its control logic and memory structure.

A Heron program is evaluated by repeatedly performing reduction steps, where each corresponds to a single clock cycle in the hardware architecture. We model each cycle as a function, `step :: State->State`. The `State` type is a sextuple capturing each of the independent memory structures — a primary stack, program memory, heap memory, update stack, case table stack, and primitive registers. The update stack tracks references to heap nodes which, once evaluated, need updating with their normal form. The case table stack is used to select the correct case alternative template once the subject is evaluated. The primitive registers are used as local storage for the PRS scheme.

There are only five main reduction rules, each dispatched by inspecting the values on the top of the primary stack. When the top atom is a pointer into heap memory, we *unwind* it, moving its contents onto the primary stack.

```
step (VAR s n  : stk, p, h,       ustk,       cstk, rs)
  =  (es'     ++ stk, p, h, us ++ ustk, cs ++ cstk, rs)
  where
    (isNF, cs, es) = splitApp (h !! n)
    es'            = dashs s es
    us = if (s && not isNF)
           then [(length stk, n)]
           else []
```

We omit definitions of helper functions for brevity, leaving the full source accessible at [17]. Informally, the `splitApp` helper decomposes an application node into a triple of a flag for normal form, its case tables, and its atoms. The `dashs` helper is part of the dynamic

sharing analysis, marking the given atoms as "possibly shared" if its first argument is true.

Updates to possibly shared heap nodes are triggered by looking at the top stack atom and the top update stack word. When an update is required, the top stack elements are committed to the heap.

```
step (e     : stk, p, h , (sp,n) : ustk, cstk, rs)
  | arity e > length stk - sp
  =  (es1' ++ es2, p, h',          ustk, cstk, rs)
  where
    (es1, es2) = splitAt (arity e) (e:stk)
    es1'       = dashs True es1
    h'         = write h n (APP True es1)
```

Primitive operations are handled by a set of three sub-rules. There is a special case for a `seq` primitive, written as `(!)`, forcing the evaluation of a primitive argument. There are two remaining options for generic operations. When both arguments are fully evaluated, we directly construct the answer on the stack. If only the first argument is fully evaluated, we swap the order of the arguments and set the opcode to an equivalent one with flipped arguments.

```
step (INT n  : PRI _ "(!)"    : e      : stk, p, ...)
  =  (e      : INT n                   : stk, p, ...)
step (INT n0 : PRI 2 op       : INT n1 : stk, p, ...)
  =  (alu op n0 n1                     : stk, p, ...)
step (INT n : PRI 2 op        : e      : stk, p, ...)
  =  (e     : PRI 2 (swap op) : INT n  : stk, p, ...)
```

Whenever a constructor is at the top of the stack, it is treated as the subject of a case expression. The constructor tag is used as an offset into the program memory case table, with the base address popped from the case table stack. From this point, evaluation continues as for any normal function call. We seperate these two reductions for clarity, but they are handled by a single cycle in actuality.

```
step     (CON a n          : stk, p, h, ustk, c:cstk, rs)
  = step (FUN True 0 (c+n) : stk, p, h, ustk,   cstk, rs)
```

The final reduction is the template instantiation itself, often using all but one of our memory resources simultaneously. Instead of having a software runtime system manage *logically distinct* memories which physically inhabit a single, shared memory resource, we are free to design *physically distinct*, parallel memories for each type of data. This alleviates the issues of memory contention and slow, sequential accesses. A Heron core instantiates the spinal application onto the stack, and instantiates the internal applications between the heap and the primitive registers. The `instAtoms` and `instApp` helpers resolve ARG pointers, VAR pointers (relative the current heap pointer), and REG pointers. `instApp` is also responsible for the instantiation of PRS candidates.

```
step (FUN _ _ n : stk , p, h       , ustk,       cstk, rs )
  =  (            stk', p, h ++ us', ustk, cs ++ cstk, rs')
  where (a, us, v ) = p !! n
        (_, cs, es) = splitApp v
        es'         = instAtoms stk h rs es
        stk'        = es' ++ drop a stk
        (us', rs')  = foldl (instApp stk h) ([], rs) us
```

The remainder of this paper extends these semantics and explores a practical hardware implementation.

3 TECHNICAL IMPLEMENTATION

This section describes the Heron graph reduction hardware. Section 3.1 discusses our modifications to the template instantiation semantics, improving performance and memory footprints. These require modifications at both the compiler and hardware-level. Section 3.2 presents the hardware implementation of the semantics. We target modern UltraScale+ FPGA architectures and offer EDA tooling optimisations in lieu of traditional pipelining techniques. As an overview, table 1 summarises this section's additions and their relative impact on the final architecture.

The baseline architecture is in Fig. 6. As in its semantics, we have a mostly combinatorial control circuit hidden within the *interconnect* block. Attached to this is the set of six main memory components, all of which are accessible concurrently due to the tight co-design between the template language and the hardware architecture.

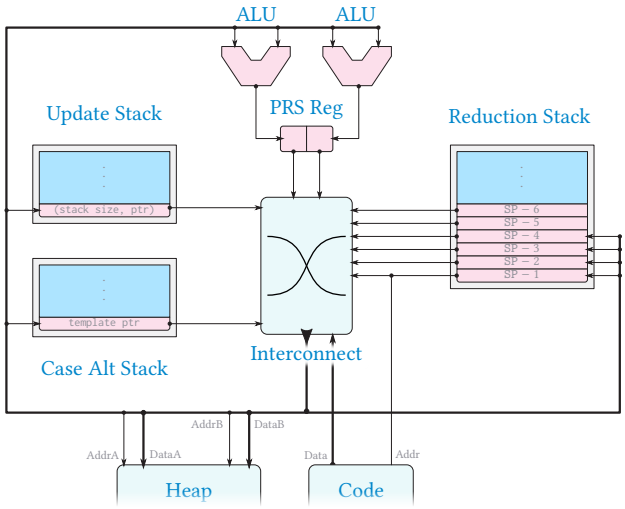


Figure 6: System architecture for the baseline Heron core

It is important to note that, although we present a *sequential* graph reduction core, we fundamentally exploit low-level parallelism in the execution model. Instantiating a template might touch five of these memories (and the primary stack demands parallel access itself) and our architecture facilitates this in a single clock cycle. A conventional von Neumann architecture could take tens of clock cycles for sequential accesses and stack accounting operations.

3.1 Compiler-Level Features

Three concerns at the compiler-level are:

- Increasing utilisation of (split) template resources.
- Improving case expression representations.
- Enabling longer applications to better exploit wide memories.

Table 1: Summary of Heron Core Features

Optimisation technique	Required support		Intended goals				Negative Impact
	Compiler	Hardware	cycles	allocs	code size	frequency	
Relaxing template constraints	✔	✔	↓	↓	↓		Control complexity
Small case table inlining	✔	✔			↓		Template size and heap width[1]
Postfix primitive operations	✔	✔	↓	↓	↓		Rare damage to allocs and code size[2]
Device & EDA tool specialisation		✔				↑	Implementation time increase

While previously handled at a software-level, we aim to elide existing overheads at a hardware-level with minimal additional cost. Sections 3.1.1 to 3.1.3 attempt to address these concerns in turn, each proposing a small change to the template semantics without modifying the source language.

3.1.1 Relaxing Template Constraints. Large supercombinators are unavoidably split over several Heron templates. In the Reduceron implementation, application length and the number of allocations are not the only constraints that trigger splitting of templates. These extra split templates have a direct impact on code size because each template is a fixed-size structure occupying hundreds of bits. Recall that memory resources were one limiting factor for the Reduceron implementation, so any reduction of memory footprint is valuable. This section introduces a set of architectural tweaks which relieve us from these extra splits.

Consider the compilation of our `fromTo` example for an architecture with `SpineLen=3`, `ApLen=3`, and `MaxAps=2`, instead of the Heron core's respective default parameters of 8, 6, and 2. We might expect the three-template solution shown in Listing 1. However, the original architecture flags two different hazards for these templates.

The first constraint precludes the use of a heap pointer *immediately* after its allocation. The issue here is that the circuit might attempt to prefetch the contents of the heap node during the same cycle it is instantiated. Depending on the heap architecture, this can return invalid data. The restriction identifies any template who's spine application starts with a heap pointer which is also allocated by that template (`VAR n`, for any non-negative `n`) — including the `fromTo` template in Listing 1. Originally, the `fromTo` template would need to be split into two. Heron has no such constraint — we introduce extra logic to forward such nodes through to the next cycle instead of prefetching from the heap.

The second constraint restricts heap use more harshly. It guarantees that whenever prefetching from the heap *is* needed, it can always succeed. To ensure there is always a port on the heap memory available for prefetching, most templates are restricted to one fewer applications than advertised (`MaxAps-1`). The full number of applications is only permitted in templates with no useful spine application. While this enables particularly dense representation of templates, it has substantial consequences for benchmark runtimes and number of templates — including the splitting of the `fromTo#T` template. Heron does not have this constraint, instead opting to dynamically detect the hazard at runtime. An extra cycle is required only in the rare case when a template instantiation allocates `MaxAps` heap applications, requires heap prefetching, *and* the forwarding technique above is not applicable.

These two techniques allow a Heron core to instantiate the intuitive template from Listing 1 in, at most, two cycles, rather than the four cycles required originally.

Finally, Heron enhances the original approach to splitting spinal applications longer than `SpineLen`. As standard, the initial portion of a long spine is redistributed as heap applications, which require extra unwinding. A consequence of this is, for a long chain of split templates, only the final template will have useful atoms in the spine. We could, in most circumstances, better utilise our templates if we can divide particularly long spines *between* split template spines. The challenge of this lies in handling instantiation of arguments if we incrementally construct the spine — the argument indices will change, and some arguments might be pushed past the reach of our stack addressing. To counter this, we freeze a copy of the top of the stack for use during instantiation of split templates chains. The small-step semantics require only minor adjustments, after adding the new frozen stack copy, `frz`, to our state. In particular, we perform instantiation relative to the frozen version of the stack:

```
step
    (FUN b _ n:stk , p, h     , ustk,     cstk, rs , frz )
  = (          stk', p, h++us', ustk, cs++cstk, rs', frz')
  where
    (a, us, v ) = p !! n
    (_, cs, es) = splitApp v
    frz'        = if b then stk else frz
    es'         = instAtoms frz' h rs es
    stk'        = es' ++ drop a stk
    (us', rs')  = foldl (instApp frz' h) ([], rs) us
```

The cumulative effects of these three modifications are shown in Fig. 8 with results gathered from the Heron core simulator. We note improvements across code size, reduction runtime, and the number of heap allocations (influencing demands on the garbage collector, once implemented). We see modest reductions to the runtime of all benchmarks except `Fib` and `Queens`. This effect is due to two factors: more efficient template packing leads to fewer cycles spent on instantiation, and better use of spinal applications avoids the overhead of unnecessary dereferencing. We see an average of a 4.8% reduction in runtime, with a maximum of 16% for `Adjoxo`. Next, we highlight the first of these factors in isolation by comparing the number of templates required for each benchmark. For this metric, we also see improvements nearly across the board (averaging to a 4.7% reduction). All benchmarks which have "hot" supercombinators with spines longer than `SpineLen` also benefit from reduced heap allocations — with `Adjoxo` seeing 35% fewer allocations.

[1]Template size increase is accounted for in Fig. 8 and heap width falls within the same number of UltraRAM resources.
[2]Due to poor compiler heuristics only, and could be precluded.

3.1.2 Small Case Table Inlining. Our next optimisation focuses solely on minimising code size via small adjustments to the compiler and Heron core architecture. All case expressions in F-lite programs infer multiple templates:

- At least one to evaluate the case scrutinee, and...
- At least one per case alternative.

This scheme keeps the representation of case table pointers in `CASE` applications small, but at significant cost to code size. The number of *trivial* templates generated for this purpose is particularly worrying. Instead, we propose an optional *inline* case table representation, used to avoid introducing sparse, trivial templates. We consider *trivial* alternatives to be any alternative with a *small* single atom. Since any decrease in number of templates could be offset by a growth in case table representation, the definition of *small* is chosen to appeal to compact bit representations of both code and heap memories. We also need to track the number of arguments which are popped from the stack by the alternative. The new general form for case tables is shown in Fig. 7.

c ::=		Case table
	`Offset` n	Pointer to template memory
	| `Inline` a a	Binary choice of alts
a ::=		Inline alternative
	`AFun` n	Pointer to template
	| `AInt` α n	Integer with arity
	| `AArg` α n	Argument index with arity
	| `ACon` α_1 α_2 n	Constructor with arity

Figure 7: Syntax for new case tables

We can compile our example `fromTo` program quite simply. We follow the same process creating templates for each alternative. After desugaring of pattern matching, translation to case tables, and inlining, we identify any 2-way case tables that have at least one trivial alternative. Both alternatives are then inlined into the new `Inline` case table construct. This reduces the `fromTo` example from four unique Reduceron templates to just the two demonstrated below.

```
FUN ⊤ 2 0 =                                                  (fromTo)
   let    APP [   ARG ⊤ 0,    PRI 2 <=,     ARG ⊤ 1 ]
    in    CASE (LInline (ACon 2 0 1) (AFun 1))
             [   VAR ⊥ 0,     ARG ⊤ 0,      ARG ⊤ 1 ]
FUN ⊤ 2 1 =                                                  (fromTo#T)
   let    APP [   ARG ⊤ 0,     PRI 2 +,       INT 1 ]
          APP [ FUN ⊤ 2 0,     VAR ⊥ 0,      ARG ⊥ 1 ]
    in    APP [    CON 2 0,    VAR ⊥ 0,      VAR ⊥ 1 ]
```

Listing 2: Templates for `fromTo` with inlined case tables

The semantics for reducing case subjects also requires modification. Any alternatives resolved to a `FUN` atom are handled as before, but any `AInt`, `AArg`, or `ACon` alternatives are instantiated directly:

```
step      (CON _ n : stk , p, h, ustk, c:cstk, rs, frz)
  | isFUN e
  =  step (e       : stk , p, h, ustk,   cstk, rs, frz)
  | otherwise
  =       (          stk', p, h, ustk,   cstk, rs, frz)
  where
    (d, e) = splitAlt (pickAlt n c)
    pickAlt n (Offset n') = AFUN (n+n')
    pickAlt n (Inline as) = as !! n
    es   = instAtoms stk h regs [e]
    stk' = es ++ drop d stk
```

The consequences of inlining trivial case alternatives, including the default 4.7% template size growth, is shown in Fig. 8. These numbers include the cumulative effects of the previous template optimisations from Section 3.1.1. This, by design, only impacts code size and not runtimes or heap allocations. Encouragingly, this does not negatively impact the code size for any of our benchmarks. We enjoy an average code size reduction of 22% and a maximum of 34% for `Braun` — a clear win with only modest architectural changes.

3.1.3 Postfix Primitive Operations. Although it is cheap in hardware for us to support long spines with a parallel stack implementation, the compiler often struggles to utilise these long spine applications. Naylor suggests that utilisation of spine applications might be improved by transforming (binary) primitive operations into postfix position and evaluating them with the help of a primitive stack [12]. This allows for longer, flatter spinal applications using Reverse Polish notation. We can rewrite the deep and narrow expression, `(+) (f x) (g y)`, into a single flat and wide expression, `f x g y (+)`. Note that only *primitive* operations appear in postfix; the supercombinators `f` and `g` remain prefix. This lets us skip the cost of allocating and unwinding the `(f x)` and `(g y)` subexpressions from the heap. Doing so often presents wins in terms of runtime and heap allocations.

Iterating on our language/architecture co-design, this feature requires modified reduction rules and a new stack for primitives, storing the fully evaluated arguments of primitive operations. The amended runtime reduction rules for binary primitives are shown below, and are now relative to both the primary stack and the new primitive stack, `pstk`:

```
step (INT n0 : INT n1 : PRI 2 op : stk, ...,      pstk)
  =  (alu op n0 n1               : stk, ...,      pstk)
step (INT n1          : PRI 2 op : stk, ..., n0 : pstk)
  =  (alu op n0 n1               : stk, ...,      pstk)
step (INT n0                     : stk, ...,      pstk)
  =  (                             stk, ..., n0 : pstk)
```

The first rule directly evaluates primitive applications with both arguments already evaluated. The second evaluates a primitive application whose first argument has already been placed on the stack of primitives. The third moves a primitive application's first argument to the stack of primitives in the case that we still need to evaluate the second.

With these new rules in place, we are faced with two options during compilation. Do we flatten or preserve these subexpressions? While our current implementations only flattens primitive applications onto the spine, there are two scenarios where this could cause damage:

Table 2: Implementations of Reduceron/Heron cores (mostly) for Alveo U280 (`SpineLen` = 6, `ApLen` = 4, `MaxAps` = 2)

Design	Clock Frequency	CLB LUTS	CLB Registers	BlockRAMs	UltraRAMs
Reduceron (on Virtex-5 LX110T) [1]	96 MHz	———	14% ———	90%	—
Direct Reduceron port[1]	175 MHz	0.9%	0.04%	6.55%	—
Heron core reimplementation	173 MHz	0.94%	0.03%	4.94%	—
+ device/tooling optimisation	180 MHz	0.70%	0.03%	0.92%	1.77%
+ new compiler support	188 MHz	0.73%	0.05%	0.89%	1.88%

- Edge cases of template splitting. Increasing application width beyond the hardware's upper bound may damage code size after template splitting.
- The interaction with PRS strategy — subexpressions flattened onto the spine are not valid PRS candidates. A static PRS scheme would enable us to make the right choice here.

Even with our simple implementation, Fig. 8 shows encouraging results when normalised against Reduceron. Some benchmarks require fewer cycles and fewer heap allocations — the most compelling being `Cichelli`'s 20% reduction in heap allocations. There are, however, a few instances of damage due to our compiler's poor decision making. In particular, conflicts with the template splitting strategy damage the code size of the `Adjoxo` and `Clausify` benchmarks. These poor choices ought to be precluded by better compiler heuristics.

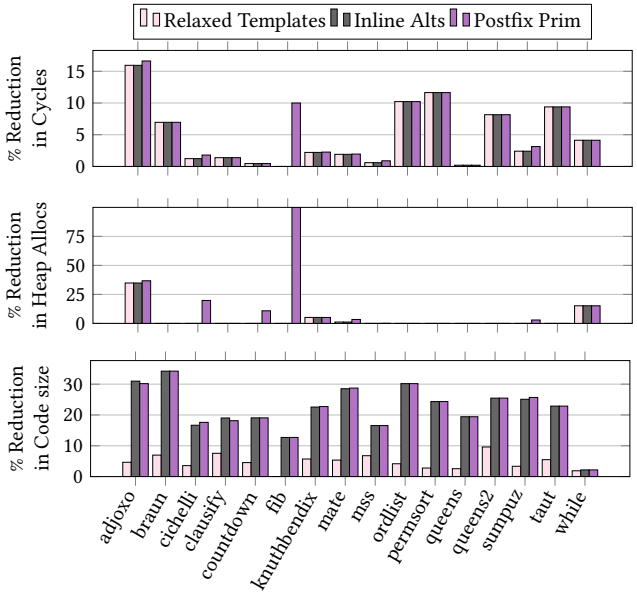


Figure 8: Heron compiler optimisations versus Reduceron

3.2 Hardware-Level Design

Section 3.1 detailed the functional aspects of the Heron core's new semantics. We now consider the *implementation* of its circuit. Two concerns at the hardware-level are:

- Remapping the hardware design to resources available in modern UltraScale+ FPGAs.
- Improving maximum clock frequency.

Improving the maximum clock frequency is particularly challenging for *pure* graph reduction schemes, including template instantiation. The next step's control flow is always derived from the current expression under reduction. This introduces a very tight data dependence between consecutive clock cycles, and proves resistant to many single-threaded pipelining techniques.

3.2.1 Remapping for UltraScale+. The implementation results in Table 2 show that a direct port of Reduceron to an Alveo U280 [18] UltraScale+ FPGA already offers substantial gains. The maximum clock frequency increases from 96 MHz to 175 MHz and the hugely restrictive 90% use of BlockRAM resources decreases to only 6.55% with the improved memory density of modern reconfigurable devices.

The underlying UltraScale+ architecture offers several improvements over the Virtex-5 device used previously [13], including *UltraRAM* resources. These are particularly dense memories — 288 Kb, structured into 4K words of 72 bits each. This directly appeals to our serial stacks, each with 4K words. To further reduce the footprint of the Heron core, we also map the dense heap memory to multiple UltraRAM resources, instead of BlockRAMs. This choice reduces the overall device utilisation to below 2% (Table 2) — a huge juxtaposition against the 90% reported in 2012.

The Heron core's implementation uses the functional hardware description language, Clash [2], which represents circuits as plain functions, sharing the syntax and frontend of Haskell. Table 2 demonstrates that this workflow generates essentially equivalent circuits with a clean behavioural source using quite idiomatic Haskell (similar to the semantics presented in Section 2.3.2). The final row in Table 2 reassures us that the additional support for the compiler optimisations from Section 3.1 does not substantially damage circuit area or timing. Our Clash implementation is also parametrised by the template dimensions (`SpineLen`, `ApLen`, `MaxAps`) allowing us to explore the design space in Section 4.1.

3.2.2 Timing Closure. Since the choice of *pure* graph reduction proves resistant to (single-threaded) hardware pipelining, Heron gains good performance by operating over wide structures. This section comments on some tweaks towards reclaiming part of this lost clock frequency potential.

The circuit synthesis, placement, and routing tasks performed by FPGA tooling are fundamentally challenging. We, as digital designers, can substantially improve the quality of these results by offering the tooling a guiding hand. During placement, Vivado's

[1]The Reduceron implementation includes a garbage collector, while the current Heron core implementation does not.

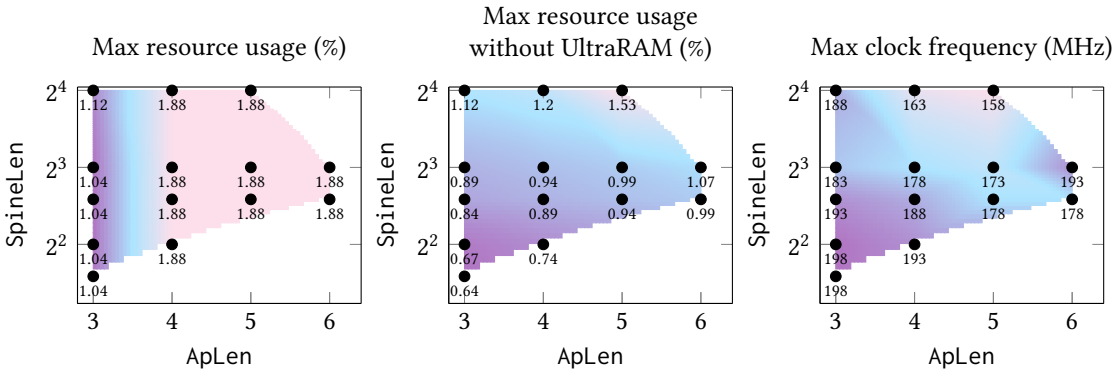


Figure 9: Non-functional circuit properties over the Heron core parameter space. Plot background shading is interpolated between data points to highlight overall trends

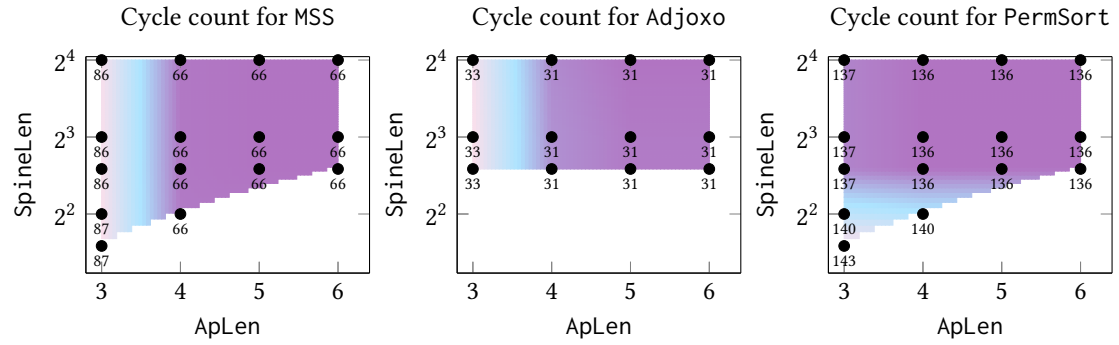


Figure 10: Benchmark cycle counts (in millions) over the Heron core parameter space

`EarlyBlockPlacement` directive encourages a compact layout for the Heron core. This directive prioritises location selection for BlockRAMs and UltraRAMs, and only then places the remaining logic. This appeals to the Heron core architecture since it is an (almost entirely combinatorial) control system spanning between columns of BlockRAMs and UltraRAMs. Timing is further improved by avoiding deep UltraRAM cascades in the low-latency heap memory, via the `MAX_URAM_CASCADE_HEIGHT` attribute. Finally, we observe that *retiming* during synthesis often helps rebalance the combinatorial control logic by pushing the state registers forward/backward along their paths.

The above tooling tweaks contrast rows 3 and 4 in Table 2, benefiting clock frequency without modifying the design.

4 SYSTEM RESULTS

4.1 Exploring the Parameter Space

The template parameters `SpineLen`, `ApLen` and `MaxAps` (Section 2.3.1) offer a design space to explore performance characteristics of generated Heron cores[2]. They offer a balance between hardware size and template bounds that best match code characteristics of real-world programs.

[2]Scripts to reproduce the results in Section 4 are all available as an open access dataset [16].

This section justifies one choice of template parameters to compare with GHC implementations in Section 4.2. For the following results we fix the `MaxAps` parameter at two, since our implementation uses dual-port UltraRAM resources for the heap memory. In this design space we explore:

1. *Resource use*, setting the upper bound on the number of cores realisable on a single chip,
2. *Clock frequency*, directly impacting wall-clock time,
3. *Clock cycle counts*, reflecting the suitability of a parameter set for a particular benchmark's characteristics.

4.1.1 Non-functional requirements. We measure the area as the maximum percentage use of any resource type on the Alveo U280 device. The hardware resources we consider are Flip-Flops, Look-Up-Tables, BlockRAMs, and UltraRAMs. Fig. 9 shows the impact of the `ApLen` and `SpineLen` template parameters on resource use and clock frequency. As these values increase, the circuit area increases, from 1.04% to 1.88%. This is always bound by the heap memory's UltraRAM resources. The trends caused by other circuit parameters (not just heap size!) are visible when ignoring UltraRAM resources. The maximum clock frequency scales inversely, from 158 MHz to 198 MHz. The `SpineLen` axis is logarithmic since the efficient implementation of the stack requires a 2^n parallelism. If `SpineLen` is not a power of two, this only reduces the number of spine atoms stored in each template and does not limit the stack implementation

or maximum arity. We report results only when SpineLen ≥ ApLen since we need to unwind heap nodes onto the stack in a single cycle. Finally, the (6, 16) configuration is omitted due to high routing congestion.

4.1.2 Benchmark Characteristics. From the results in Section 4.1.1 it is tempting to generate Heron cores to be as small as possible. However, during compilation, code is split into multiple templates when hardware bounds are exceeded (Section 2.3.1). This results in a trade-off between the desire to have small, fast cores and the desire to minimise the number of templates to execute. Fig. 10 shows the impact of the ApLen and SpineLen template parameters on clock cycle counts for three benchmarks (fewer is better).

The horizontal striping in the MSS cycle count is, in part, due to the benchmark's exclusive use of small recursive functions over lists. A SpineLen of three is enough to support all function arities and almost all spinal applications. Increasing ApLen from three to four precludes need for splitting some template heap applications. Beyond this, MSS does not benefit from increased SpineLen or ApLen.

The results for Adjoxo suggest a global pattern identical to that of MSS, but results are truncated below a SpineLen of six. This is because Adjoxo's maximum function arity is four (a result of pattern matches in foldr1) and we cannot instantiate functions with arguments past SpineLen-1. The maximum function arity enforces a hard lower bound on the SpineLen for a given program, which impacts the Braun, Clausify, KnuthBendix, Queens2, SumPuz, and Taut benchmarks. Our compiler currently does not have an arity reduction mechanism to allow evaluation of these benchmarks on small cores, but such schemes are well-researched [10].

Finally, PermSort, OrdList, and Fib exhibit vertical striping. These have long spinal applications (either from the source or introduced by the compiler) well above the benchmark's maximum function arity. The While benchmark is the only one with substantial vertical *and* horizontal striping — demonstrating both spine lengths beyond the maximum function arity and wide internal applications.

Wall-clock times are found by combining the clock frequencies (Fig. 9) and the cycle counts (Fig. 10). These results demonstrate the tension between hardware size derived from (ApLen/SpineLen) and generated code size (the number of templates). For example, a (4, 16) configuration clocks at 158 MHz whilst a (3, 4) configuration clocks at 198 MHz. However, switching from the former to the latter increases the MSS cycle count from 66 to 87 M.

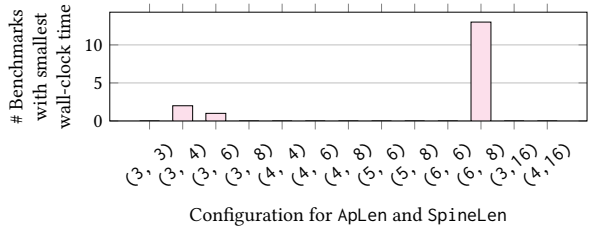


Figure 11: Histogram of Heron core ApLen and SpineLen parameters which minimise benchmark wall-clock times

Fig. 11 shows a histogram of Heron template configurations with the best wall-clock times across 16 benchmarks. *The highest clock frequency does not necessarily result in the fastest wall-clock runtime*: the highest achieved clock is 198 MHz with configuration (3, 4) however Fig. 11 shows that, for this benchmark suite, (6, 8) is the best configuration despite a lower clock frequency of 193 MHz. We therefore use (6, 8) for Heron configurations in Section 4.2. Comparing to the 2012 Reduceron's circuit implementation, this choice offers more than twice the frequency and a substantial reduction in maximum resource usage from 90% to just 1.88%. In contrast to the logical 64 bit data buses implied by Fig. 2, one cycle of Heron's template instantiation has over 1 kb at hand — 129 for each heap port, 19 for each stack atom (both pop & push), 454 for the template, and 23 for spine case table.

4.2 Comparison of Graph Reduction Systems

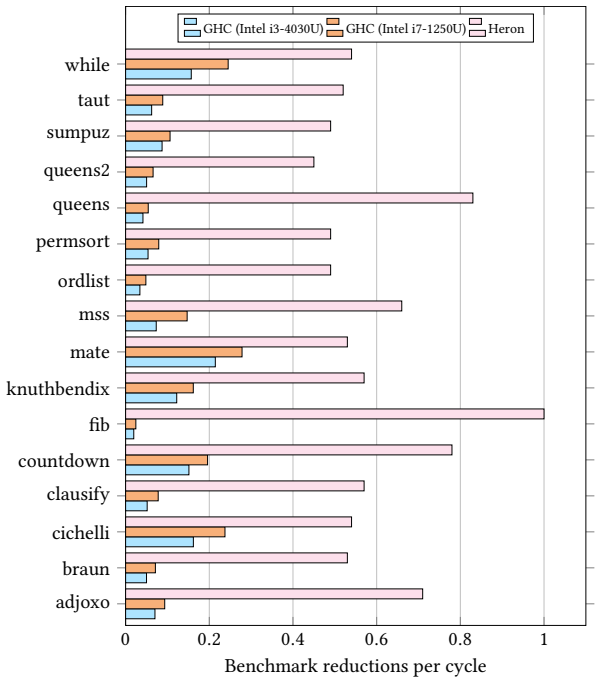


Figure 12: Comparison of the hand-reduction counts per clock cycle required for several graph reduction systems

4.2.1 Reductions per clock cycle. Fig. 12 shows that the Heron core performs substantially more reductions per cycle than both CPUs across all 16 benchmarks. The i7-1250U system performs 22% of Heron's reductions per cycle on average, and the i3-4030U only 15%.

Fig. 12 compares hand-reductions (as defined in [13]) performed per clock cycle for the Heron core and two general purpose CPUs running code generated by GHC 8.6.5. The F-lite benchmark code was ported to Haskell for the performance comparison. Since F-lite is a subset of Haskell, the porting effort was trivial. The CPU configurations include an Intel i7-1250U (Max Turbo frequency of 4.70 GHz) and a more modest Intel i3-4030U, both with default frequency

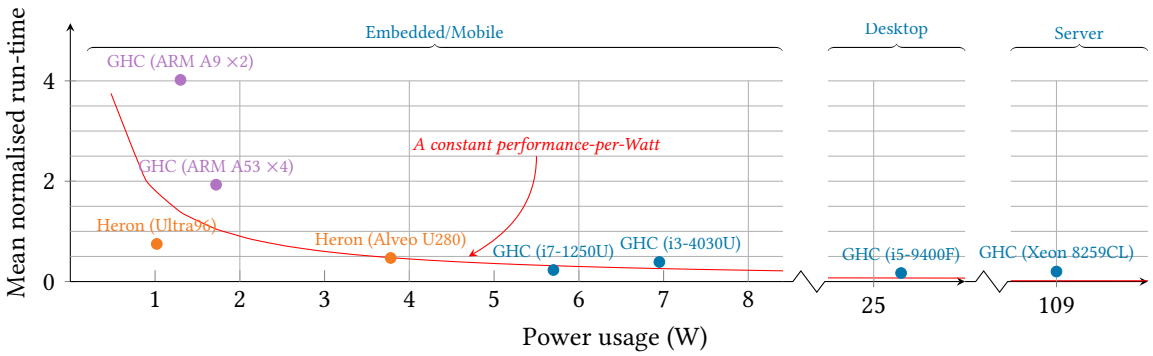


Figure 13: A comparison of performance-per-Watt across different graph reduction systems

scaling settings at 1.9 GHz. We omit results for the Reduceron here since a cycle-count comparison was already shown in Fig. 8.

The Heron results are from hardware simulation and exclude garbage collection since this is yet to be implemented. The GHC results are profiled using `perf` [15], compiled with the `-O2` flag, and also exclude garbage collection.

This does not imply that the Heron core has a faster wall-clock time, but it does highlight a substantial efficiency advantage. Although the i3-4030U is competitive with the wall-clock time of the Heron core (Section 4.2.2), it comes with a significant carbon cost. The poor reductions-per-cycle is masked by a ×10 increase in frequency (from 193 MHz to 1.9 GHz). Since modern CPUs appeal to *diverse* workloads compiled from any programming paradigm, they require extremely complex superscalar implementations, using out-of-order execution and deep pipelines for good performance. Such complexity also comes with a switching power cost. This is contrasted to Heron's energy efficiency (Section 4.2.2).

4.2.2 Performance-per-Watt. We now compare wall-clock runtime performance and power consumption across eight architectures, including two Heron implementations and embedded, mobile, desktop, and server CPU configurations. Fig. 13 captures these results, where lower is better in both axes. The wall-clock results are normalised against the 2012 implementation of Reduceron. The GHC results are gathered using the Criterion library [5] with the `-O2` compiler flag. The ARMv7 A9 configuration is from Xilinx Zynq-7000 SoCs, and the ARMv8 A53s are from the Ultra96 board. All CPU times here include garbage collection overhead, while the Heron cores currently do not yet have hardware garbage collection. For an upper-bound estimate of this discrepancy, Naylor includes a breakdown of time spent in GC for Reduceron's simple two-space collector [13].

The power usage results for Intel CPUs are gathered using `perf`'s `energy-pkg` counter and the ARM/FPGA results are reported by the vendor tooling's static analysis. Both metrics include static power consumed by any idle cores — a common real-world cost since GHC does not exploit SMP parallelism for Haskell sources without annotation. Since our default Heron target, Alveo U280, is large enough to host upwards of 50 cores, it also wastes some of the static power budget in an analogy for idle CPU cores. We also include an Ultra96 implementation, a smaller device capable of hosting ≈ 3 cores, with a more commensurate power usage.

The Ultra96 Heron core is a clear outlier in terms of performance-per-Watt. This suggests that even a single-core graph reduction architecture could be appealing when cost of operation, financial or environmental, is a concern. For power-constrained applications, the Ultra96 Heron core obtains a ×2.5–5 performance increase compared to the embedded ARM configurations while staying within the same power budget. When absolute performance is the primary concern, high-end desktop and server CPUs still outperform an Alveo Heron core by approximately ×2.5 — at the expense of up to an order of magnitude increase in power usage.

A stronger comparison can be made to the mobile Intel processors. The Alveo Heron core evaluates five of the benchmark programs *faster* than GHC with the Intel i3-4030U CPU, despite the Heron core clocking ×10 slower. The GHC results on the Intel i7-1250U CPU are all faster than the Alveo Heron core, as expected, with a similar performance-per-Watt. However, if an application can accept an increase in wall-clock times, the Ultra96 Heron core offers an average performance-per-Watt of ×1.7 that of the Intel i7-1250U.

We also expect these early results to scale well to a parallel implementation, exploiting implicit parallelism in pure functional programs. This could substantially improve the Alveo Heron core performance without major damage to its power usage (as it stands, 83% is for the whole device's *static power* rather than *dynamic power* required by Heron). As we have shown, despite lower clock frequencies, the efficiency of custom graph reduction logic already brings the Heron into competition with mature CPUs.

5 CONCLUSION & FUTURE WORK

We present the co-design of a template instantiation language and an FPGA-based hardware architecture, resulting in a small, parametrised graph reduction core. Three changes to the operational semantics (Section 3.1) of the graph templates, and remapping to modern FPGA devices (Section 3.2), reduces: (1) runtime to 48%, (2) code size by 22% and (3) heap allocations by 6.3%. A 193 MHz Heron core outperforms a mobile Intel i3 1.9 GHz CPU for 5 of 16 benchmarks. Improved resource density removes hard ceilings faced by the previous generation of implementations [4, 13]; maximum FPGA resource use for Heron is 1.88% compared to 90% for

Reduceron, enabling future parallel architectures with tens of cores. Further work is threefold: (1) adding a concurrent hardware garbage collector to Heron (e.g. [3]), (2) a multi-core Heron architecture with dense off-chip memories, and exploiting its parametrised design (Section 4.1) by (3) optimisation of core parameters via static analysis of workloads.

ACKNOWLEDGMENTS

The authors thank Sven-Bodo Scholz, Hans-Wolfgang Loidl, and Greg Michaelson for their advice. This work was supported by the Engineering and Physical Sciences Research Council (EP/W009447/1).

REFERENCES

[1] Lennart Augustsson. 1992. BWM: A Concrete Machine for Graph Reduction. In *Functional Programming, Glasgow 1991*, Rogardt Heldal, Carsten Kehler Holst, and Philip Wadler (Eds.). Springer London, London, 36–50.

[2] C.P.R. Baaij. 2015. *Digital circuit in CλaSH: functional specifications and type-directed synthesis*. Ph. D. Dissertation. University of Twente, Netherlands. https://doi.org/10.3990/1.9789036538039

[3] Martha Barker, Stephen A. Edwards, and Martha A. Kim. 2022. Synthesized In-BRAM Garbage Collection for Accelerators with Immutable Memory. In *32nd International Conference on Field-Programmable Logic and Applications, FPL 2022, Belfast, United Kingdom, August 29 - Sept. 2, 2022*. IEEE, 47–53. https://doi.org/10.1109/FPL57034.2022.00019

[4] Arjan Boeijink, Philip K. F. Hölzenspies, and Jan Kuper. 2010. Introducing the PilGRIM: A Processor for Executing Lazy Functional Languages. In *Proceedings of the 22nd International Conference on Implementation and Application of Functional Languages* (Alphen aan den Rijn, The Netherlands) *(IFL'10)*. Springer-Verlag, Berlin, Heidelberg, 54–71.

[5] Bryan O'Sullivan. 2014. *criterion: a Haskell microbenchmarking library*. http://www.serpentine.com/criterion/

[6] Celeste Hollenbeck, Michael F. P. O'Boyle, and Michel Steuwer. 2022. Investigating magic numbers: improving the inlining heuristic in the Glasgow Haskell Compiler. In *Haskell '22: 15th ACM SIGPLAN International Haskell Symposium, Ljubljana, Slovenia, September 15 - 16, 2022*, Nadia Polikarpova (Ed.). ACM, 81–94.

[7] Simon L. Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler inliner. *J. Funct. Program.* 12, 4&5 (2002), 393–433.

[8] SL Peyton Jones. 1992. Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-machine. *Journal of Functional Programming* 2 (July 1992), 127–202. https://www.microsoft.com/en-us/research/publication/implementing-lazy-functional-languages-on-stock-hardware-the-spineless-tagless-g-machine/

[9] SL Peyton Jones, Chris Clack, Jon Salkild, and Mark Hardie. 1987. GRIP - a high-performance architecture for parallel graph reduction. In *Proceedings of the conference on Functional Programming languages and Computer Architecture, Portland*. Springer Verlag LNCS 274, 98–112. https://www.microsoft.com/en-us/research/publication/grip-a-high-performance-architecture-for-parallel-graph-reduction/

[10] Richard Kennaway and Ronan Sleep. 1988. Director Strings as Combinators. *ACM Trans. Program. Lang. Syst.* 10, 4 (oct 1988), 602–626. https://doi.org/10.1145/48022.48026

[11] Richard B. Kieburtz. 1985. The G-Machine: A Fast, Graph-Reduction Evaluator. In *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings (Lecture Notes in Computer Science, Vol. 201)*, Jean-Pierre Jouannaud (Ed.). Springer, 400–413.

[12] Matthew Naylor. 2009. *Reduceron Project Memo 38: Benefits of a primitive-value stack*. University of York. https://www.cs.york.ac.uk/fp/reduceron/memos/Memo38.txt

[13] Matthew Naylor and Colin Runciman. 2012. The Reduceron reconfigured and re-evaluated. *Journal of Functional Programming* 22, 4-5 (2012), 574–613. https://doi.org/10.1017/S0956796812000214

[14] Nicholas Nethercote and Alan Mycroft. 2002. The cache behaviour of large lazy functional programs on stock hardware. In *Proceedings of The Workshop on Memory Systems Performance (MSP 2002), June 16, 2002, Berlin, Germany*, Hans-Juergen Boehm and David Detlefs (Eds.). ACM, 44–55.

[15] `perf` Project. 2023. *`perf`: Linux profiling with performance counters*. https://perf.wiki.kernel.org/

[16] Craig Ramsay and Robert Stewart. 2024. *Dataset for results in "Heron: Modern Hardware Graph Reduction"*. https://doi.org/10.17861/b9ab6ca4-a86c-4bf0-b6f3-e462129b6ebb

[17] Craig Ramsay and Robert Stewart. 2024. *Dataset for the Heron Core in "Heron: Modern Hardware Graph Reduction"*. https://doi.org/10.17861/f4fab5ef-ae98-4300-8328-ea59e47ff8c6

[18] Xilinx, Inc. 2023. *DS963 — Alveo U280 Data Center Accelerator Card Data Sheet (v1.6)*. https://docs.xilinx.com/r/en-US/ds963-u280