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

Implementations of general purpose functional languages overwhelmingly target CPUs. Stagnating clock frequencies in the mid 2000s sparked research into alternative speedup techniques such as parallelism, locality and compiler heuristics. The semantic gap between functional languages and CPU assembly imposes limits on what these techniques can achieve. Recent FPGA advances have seen step change improvements to density of on-chip hardware resources, including wide memories with multiple independent ports — reason to reinvestigate *hardware* implementations of functional languages. This paper presents *Heron*, a special purpose processor core for pure, non-strict functional languages. We co-design its language semantics and parameterised design, gaining a high reductions-per-cycle performance metric. The Heron core is energy efficient, requiring up to ×9.5 fewer clock cycles across 16 benchmarks compared with GHC in software. 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. Early measurement of its performance-per-Watt suggests that a single-core Heron system is already a compelling solution for embedded applications. The simplicity of Heron's design results in a maximum FPGA fabric use of just 1.88%, paving the way for future scalable 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 *Proceedings of IFL 2023.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

IFL 2023, August 29–31, 2023, Braga, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9831-X/XX/XX.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

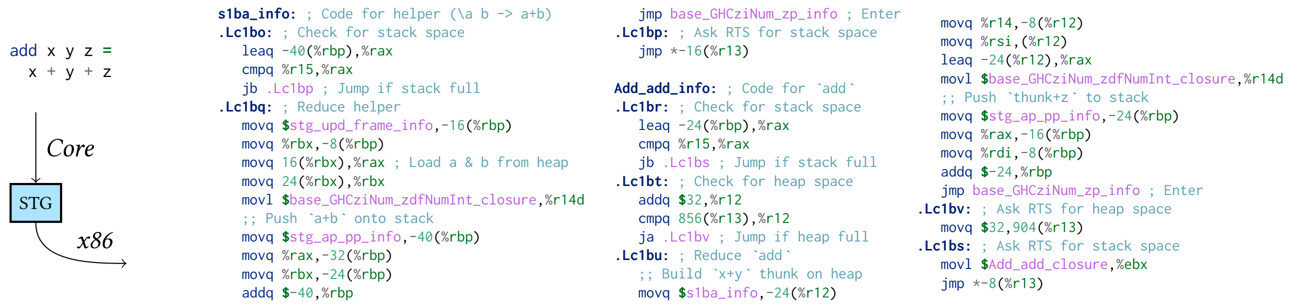
1 Introduction

Functional language implementations overwhelmingly target fixed CPU architectures. The conceptual mismatch between high-level functional execution models and low-level x86/AArch64/PowerPC CPU instruction sets necessitates compiler transformations (Section 2.2). Recent step changes in Field Programmable Gate Array (FPGA) logic density 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, 15].
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:

- Section 2 presents Heron's native language and custom hardware architecture in a co-design process.
- Section 3.2 re-evaluates previous work on Reduceron [13] in the context of modern FPGA architectures, over a decade later. We highlight substantial *passive* improvements due to modern FPGA technologies (+82% to clock frequency and −93% to per-chip memory resources) as result of a direct port.
- Section 3 contributes four further optimisations for the Heron core, resulting in *active* improvements of a cumulative +113% to reduction performance and a further ×3.5 reduction to total memory resources.
- Section 4.1 explores Heron core's trade-offs for space and runtime performance with 16 benchmarks.
- Section 4.2 compares the Heron core against other software and hardware graph reduction systems.



```
add x y z =
  x + y + z
```

```
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 1. Compiling a simple Haskell function to x86 CPU assembly, using closures, info tables, and explicit stack accounting

2 Graph Reduction Techniques

There is a spectrum of graph reduction implementations, ranging from fixed general purpose CPUs (Section 2.2) to truly custom circuits (Section 2.3). Heron's foundations share ideas and its baseline operational semantics (Section 2.3.2) with the Reduceron project [13]. Since the baseline Heron core is a faithful Reduceron re-implementation, Section 2.1 begins with a summary of its limiting factors, which are overcome by our own contributions in Sections 3 and 4.

2.1 The Key Observation

Reduceron [13] showed that hardware implementations on a *single chip* is a promising avenue for graph reduction machines. Unlike GHC's compiled graph reduction approach for CPUs, Reduceron took a simple approach with template instantiation. Simplicity is desirable because it allows architecture choices to be easily evolved with a relatively small development effort. It also allows for scaling simple graph reduction circuits for multi-core parallelism. However, the efforts with Reduceron reached a hard ceiling enforced by the contemporary (pre-2012) FPGA devices. This ceiling manifested in two directions:

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.

Moreover, the limit on memory capacity precluded efforts to explore parallel Reduceron architectures. Secondly, the simplicity of template instantiation (and the low-level parallelism therein) is counterbalanced by its resilience to pipelining techniques, because the behaviour in every cycle is determined by the stack state from the previous cycle. This resulted in a 96 MHz clock frequency for Reduceron, which is 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 almost 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 step change increase FPGA performance — 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 to the Spineless Tagless G-machine (STG) [8] as a stepping-stone to von Neumann architectures. This results in basic code blocks and heap representations of closures, e.g. Fig. 1 shows GHC's x86_64 output for a simple function. Software implementations have little choice but to use program transformations like STG or the G-Machine [12] (from which STG was derived), because a CPU's architecture is fixed and its only input is assembly code. This approach offers limited locality control within a CPU's memory hierarchy. Performance can suffer from memory contention and cache misses [15]. Reliance on indirect jumps can also damage pipeline utilisation. Moreover compilation to x86 or AArch64 sequentialises code into instruction streams for fetch-decode-execute cycles. Compilers try to efficiently map functional code into CPU architectures using heuristics [7, 9] in the absence of a precise model of specialised hardware.

Many specialised hardware architectures in the 1980s used stock processors for actually performing reductions, e.g. Motorola 68020 microprocessors in GRIP [10], so these often used programmed reduction. More recently, PilGRIM [4] moved towards the custom logic approach with a special purpose instruction set that lifted the circuitry to the intermediate representation of the G-Machine. 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 highly clocked design, it would be at the cost of a highly complex circuit. In contrast, Heron's relative simplicity offers much better potential for scaling to multi-core parallel graph reduction.

2.3 Hardware for Pure Graph Reduction

Heron, PilGRIM and Reduceron all borrowed 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 for reductions. Our choice of template instantiation (a form of *pure* reduction) is precisely to forego the STG-style translation to a sequential instruction stream. This means only in the cases of Heron and Reduceron, 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, which motivated our focus on alternative performance 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. 2, 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.

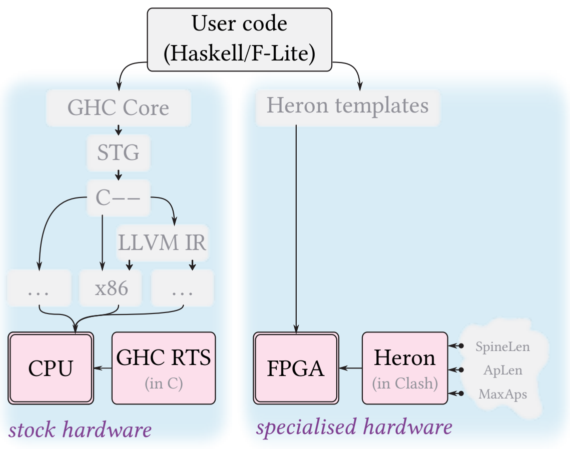


Figure 2. Compiler flows for CPU and Heron cores

Heron's native assembly is a non-strict functional language, with syntax shown in Fig. 3. The metavariables α, n, and s are used for arities, integers, and "possibly shared" flags respectively. A program is a series of supercombinators described as *Heron graph templates* — or simply *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, presented as part of [13], translates F-lite sources (an untyped subset of Haskell with case expressions, uniform pattern matching, and let-bindings) to this template format.

e ::=	(Atoms)
FUN s α n	(Function pointer)
| ARG s n	(Argument pointer)
| VAR s n	(Application pointer)
| CON α n	(Constructor tag)
| INT n	(Integer literal)
| PRI α $\otimes$	(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 α n = let $\overline{u}$ in v	
p ::= $\overline{t}$	(Program)

Figure 3. Syntax of Heron templates

The primitive operations includes `seq` to force evaluation of a primitive and the binary operations $(+)$, $(-)$, $(=)$, $(\neq)$, and $(\leq)$. 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` 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);
};
```

This can be trivially mapped to three templates after refactoring case expressions to case tables and translating to an A-normal form (Listing 1). 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

```
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

(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 overall heap memory capacity. One way to visualise the consequence of these three dimensions is shown in Fig. 4 using an expression graph directly.

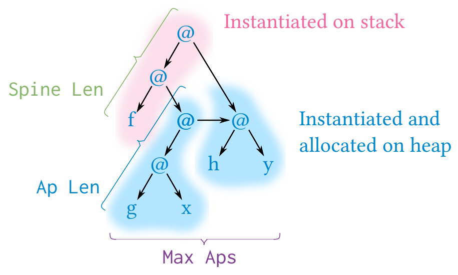


Figure 4. An example expression graph grouped by template resources and dimensions

For completeness, Fig. 3 also introduced an optional optimisation via the REG and PRIM constructs. This facilitates the Primitive-Redex Speculation (PRS) feature from Section 5.3 of [13], which the Heron core supports. 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.

2.3.2 An Operational Semantics.

This section introduces an operational semantics for the baseline Heron core. These rules are a reimplementation of [13] 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 using 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 subrules. 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 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

Here we detail our novel contributions towards specialised graph reduction hardware in two main categories. 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 solely concerns the hardware implementation for the semantics. We introduce a mapping of resources to modern UltraScale+ FPGA architectures and an approach to increasing heap performance 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 high-level structure of the baseline architecture is in Fig. 5. 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.

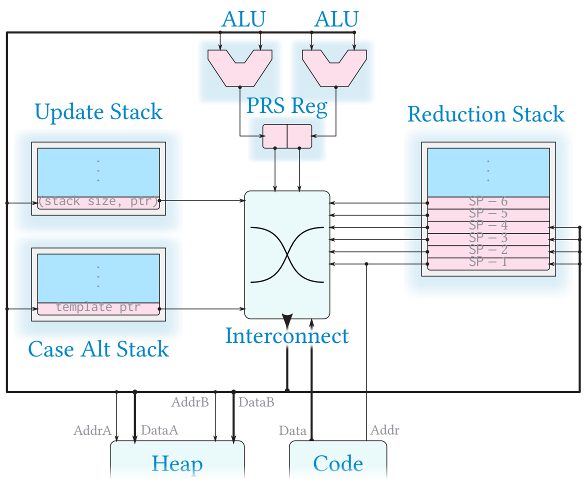


Figure 5. System architecture for the Heron core

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 hundreds of clock cycles for sequential accesses and stack accounting operations. Beyond this, our results also demonstrate the opportunity for multi-core implementations in which coarser implicit parallelism can be enjoyed, performing multiple reductions concurrently.

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 parallel memories.

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

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

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 disallows newly allocated heap pointers from appearing at the top of the spine. 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 might return invalid data. Heap pointers in the template format are `VAR` atoms, with an address relative to the current heap pointer. This restriction then applies to any `VAR n` with a non-negative `n`, including the top of the `fromTo` example's spine. Originally, the `fromTo` template would need to be split into two. Heron has no such constraint — we introduce extra forwarding logic to thread new applications through to the next cycle without the heap round-trip.

The second related constraint guarantees that there will always be capacity to prefetch from the heap memory when needed. Unless a template's spine is of the form `[FUN _ _ x]`, guaranteeing that no heap prefetching is required, all templates are only allowed `MaxAps-1` heap applications! 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.

[1]Template size increase is accounted for in Fig. 7 and heap width falls within the same number of UltraRAM resources.

[2]Due to poor compiler heuristics only, and could be precluded.

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 with a small hardware addition, 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. 7 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 5.8% 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 5% reduction). Finally, we note that all benchmarks which have "hot" supercombinators with spines longer than `SpineLen` also benefit from reduced heap allocations — four of which see over 20% fewer allocations.

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 parameterised and 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. 6.

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 6. 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.

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:

```
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

```
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 visualised in Fig. 7. These numbers also 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 `Adjoxo` — 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. Our final compiler-level optimisation attempts to address this via an implementation of the idea presented in [14]. There, Naylor suggests that (binary) primitive operations could be transformed to postfix position and evaluated with the help of a primitive stack. This allows for longer, flatter spinal applications using Reverse Polish notation. Using an example from [14], we can rewrite the deep and narrow expression, `(+) (f x) (g y)`, into a single flat and wide expression, `f x g y (+)`. This lets us skip the cost of unwinding the `(f x)` and `(g y)` subexpressions from the heap. Doing so will often present a win 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 always flattens them, there are two scenarios where this causes damage:

- 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. 7 shows encouraging results when normalised against Reduceron. Some benchmarks require fewer cycles and fewer heap allocations — the most compelling being `Cichelli`'s 31% reduction in

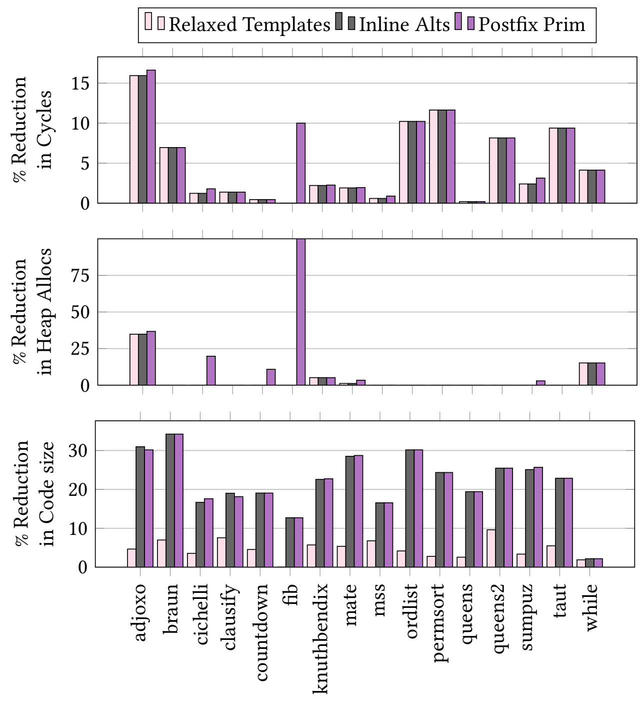


Figure 7. Heron compiler optimisations versus Reduceron

heap allocations. There are, however, a few instances of damage due to our compiler's poor decision making. In particular, the conflict with the PRS strategy damages the runtime of the `KnuthBendix` benchmark. These poor choices ought to be precluded by better compiler heuristics.

3.2 Hardware-Level Features

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

- Remapping the hardware design to resources available in modern UltraScale+ FPGAs.
- Reducing circuit area, enabling future work towards a multi-core architecture.
- Improving maximum clock frequency.

Section 3.2.1 considers the first two, while clock frequency improvements are investigated in Section 3.2.2. The latter is particularly challenging since many single-threaded pipelining techniques are not easily applicable to pure graph reduction cores.

3.2.1 Reimplementation with 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 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 similar to the semantics presented in Section 2.3.2 (in stark contrast to previous gate-level descriptions). 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 completely parameterised by the template dimensions (`SpineLen, ApLen, MaxAps`) allowing us to explore the design space in Section 4.1.

The underlying UltraScale+ architecture offers several improvements over the Virtex-5 device used in [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. However, the fixed geometry of the UltraRAM blocks imposes a new performance overhead for our 32K-word heap. Section 3.2.2 attempts to mitigate this effect, and further improve circuit timing.

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%	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%

3.2.2 FPGA tooling optimisations.
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. For the Heron core, this process results in a best-case 10% improvement to maximum clock frequency, compared to a direct port of Reduceron.

By default, the heap memory is synthesised as two independent cascades of 8 UltraRAM blocks: one cascade for the least-significant bits, and one for the most-significant. Within each cascade, signals are routed through all 8 memories sequentially, providing a total depth of 32K. This implementation minimises circuit area, but at the cost of maximum clock frequency. We instead force the tools to avoid these long cascades, implementing each as a bank of 8 interleaved blocks, as shown in Fig. 8. This substantially improves the timing results for deep, latency-sensitive memories.

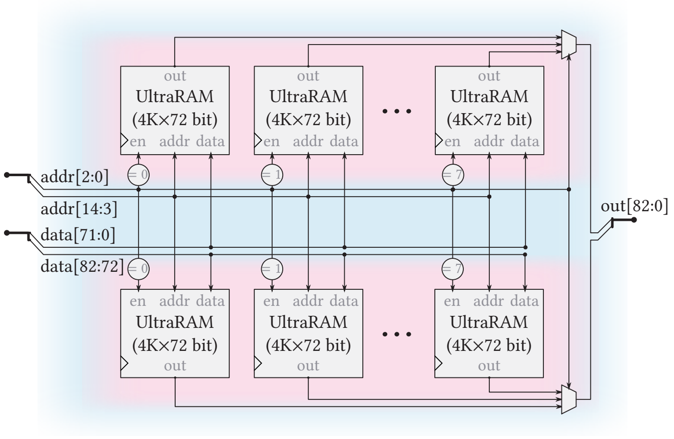


Figure 8. Simplified heap memory architecture with `CASCADE_HEIGHT=1` (showing only one of two ports)

Additionally, the timing results improve when we allow *retiming* to occur during synthesis. Here, the combinatorial logic that performs accounting for the heap pointer is automatically redistributed across its register, balancing the distribution of complexity more evenly between consecutive cycles.

During placement, Vivado's `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.

4 System Results

4.1 Exploring the Parameter Space

The template parameters `SpineLen`, `ApLen` and `MaxAps` (Section 2.3.1) offers a design space to explore performance characteristics of generated Heron cores. They offer a balance between hardware size and template bounds that best match code characteristics of real-world programs. This section justifies one choice of parameters for a given benchmark suite, before it is compared to GHC implementations in Section 4.2. For the following results we fix the `MaxAps` parameter at two, since our implementation currently 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 in a future single-chip multi-core architecture,
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. This dictates the upper bound on the number of cores for a future single-chip multi-core architecture.

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. Therefore, if the characteristics of compiled code could be ignored, smaller Heron cores are more suited for meet these non-functional requirements. The

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

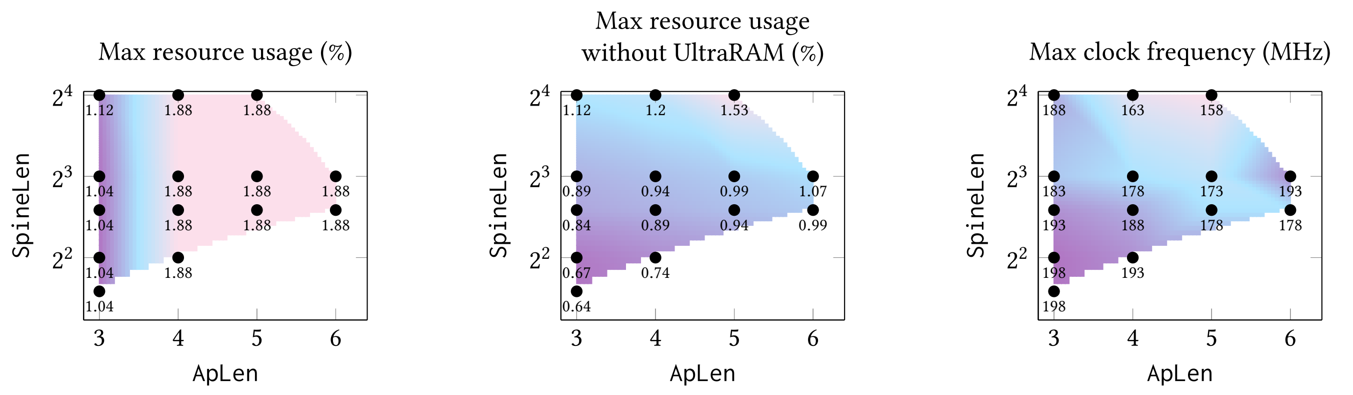


Figure 9. Non-functional circuit properties over the Heron core parameter space

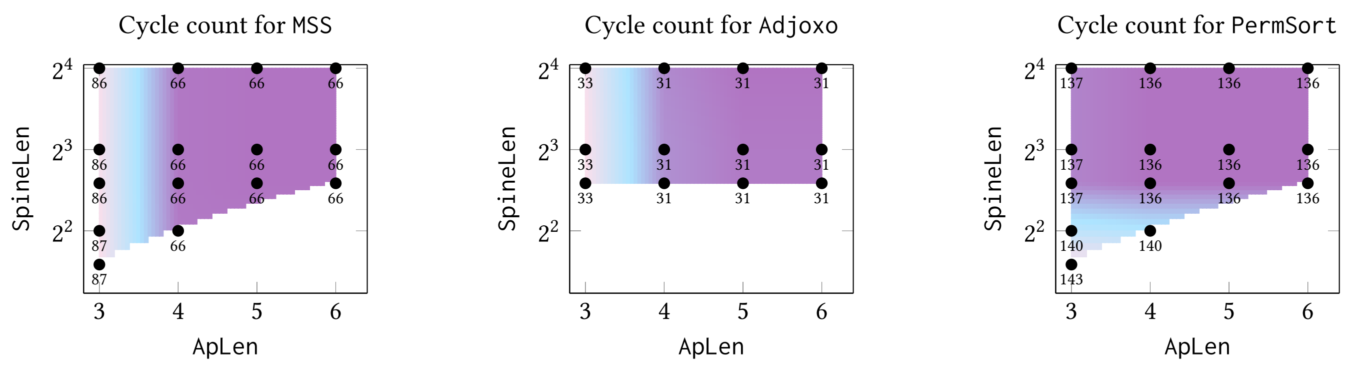


Figure 10. Benchmark cycle counts (in millions) over the Heron core parameter space

`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` $\geq$ `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 may be 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 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. The fewer cycle counts the 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 such benchmarks on small cores, although [11] suggests that this is possible.

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.

Fig. 11 shows a histogram of the 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 having a lower clock frequency of 193 MHz. We therefore use (6, 8) for comparing with Reduceron and two CPUs 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%.

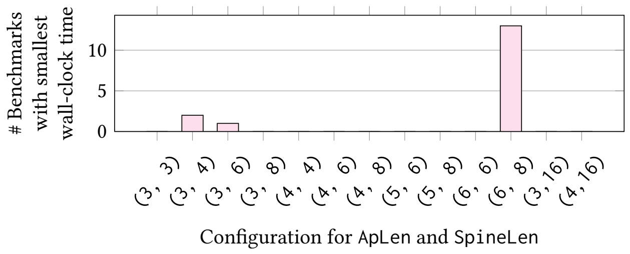


Figure 11. Histogram of Heron core `ApLen` and `SpineLen` parameters which minimise benchmark wall-clock times

4.2 Comparison of Graph Reduction Systems

4.2.1 Reductions per clock cycle. Fig. 12 shows a comparison of clock cycle counts required by 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 scaling settings at 1.9 GHz. We omit results for the Reduceron here since a comparison was already shown in Fig. 7.

The GHC results are profiled using `perf` [16] and are compiled with the `-O2` flag. The Heron results are gathered via hardware simulation and exclude garbage collection since this is yet to be implemented. We believe this is an acceptable omission since recent work on FPGA accelerated garbage collection [3] shows that a simple implementation can work almost entirely concurrently to mutations/reductions.

Fig. 12 shows that the Heron core requires substantially fewer cycles than both CPU across all 16 benchmarks. The i7-1250U system requires ×6.9 Heron's cycles on average, and the i3-4030U requires ×9.5. This does not imply that the

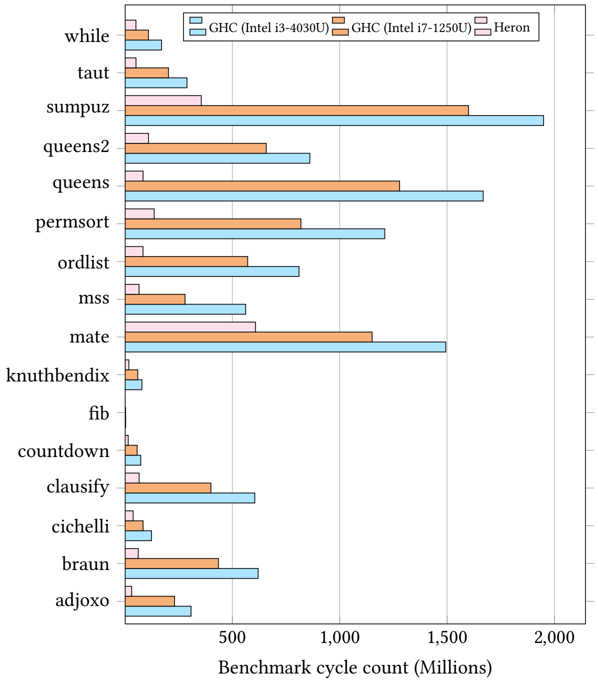


Figure 12. Comparison of the clock cycle counts required for several graph reduction systems

Heron core has a faster wall-clock time, but it does highlight a substantial *reductions-per-cycle* 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 power consumption in Section 4.2.2.

4.2.2 Performance-per-Watt. We now compare absolute 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. Both the Heron results exclude garbage collection, as in Section 4.2.1. [13] includes a breakdown of time spent in GC for Reduceron's simple two-space collector.

The power usage results for Intel CPUs are gathered using `perf`'s `energy-pkg` counter and the ARM/FPGA results are

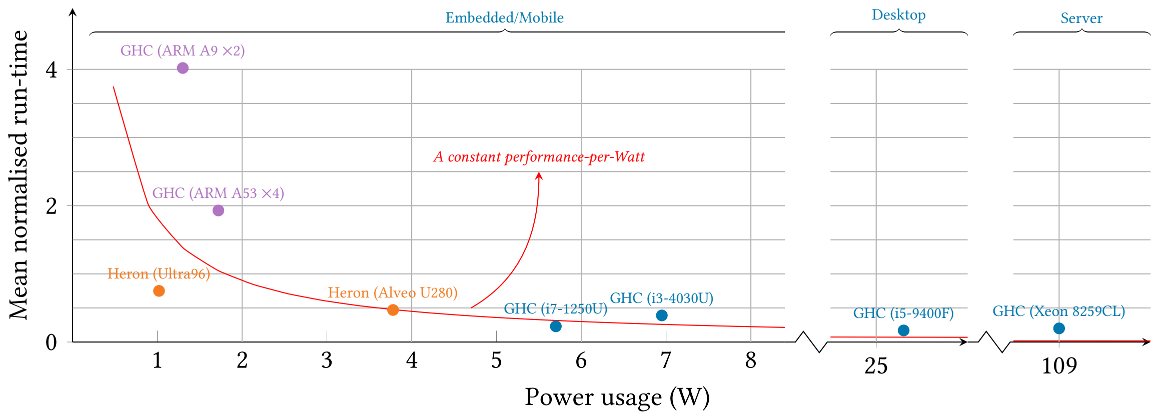


Figure 13. A comparison of performance-per-Watt across different graph reduction systems

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 power budget in an analogy for idle CPU cores. We also include a Heron implementation for Ultra96, 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 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 than the i3 CPU (193 MHz vs 1.9 GHz). GHC on the Intel i7-1250U CPU are all faster than the Alveo Heron core, as expected, but 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 believe that these early wall-clock results are still encouraging due to Heron's clear roadmap towards exploiting implicit parallelism present in pure functional programs. All of this paper's implementation results point towards a future massively multi-core implementation. Since each Heron core occupies under 2% of our target device, an implementation of a parallel graph reduction machine becomes low-hanging fruit. 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 slower clock frequencies, the efficiency of custom graph reduction logic brings the Heron into competition with mature CPUs. We believe that similar levels of efficiency can be achieved in the context of parallelism, i.e. lower overheads than GHC's software runtime system for context switching, load balancing and scheduling. In [6], GRIP's early automatic scheduling gives a ×9 speed-up of workloads with very fine-grain parallelism given 18 reduction cores.

5 Conclusion & Future Work

This paper presents the co-design of a template instantiation language and an FPGA-based hardware architecture, resulting in a small, parameterised graph reduction core. Three changes to the operational semantics (Section 3.1) of the graph templates, combined with circuit-level and memory architecture improvements (Section 3.2), reduces: (1) runtime to 47%, (2) code size by 22% and (3) heap allocations by 16%. A 193 MHz Heron core outperforms a mid-range Intel i3 1.9 GHz CPU for 5 of 16 benchmarks and is competitive with a high-end mobile Intel i7 CPU. Improved resource density of modern FPGAs removes hard ceilings faced by the previous generation of implementations [4, 13]. For example, maximum FPGA resource use for Heron is 1.88% compared with 90% for Reduceron, which paves the way for future parallel architectures with tens of Heron cores.

Immediate further work is fourfold: (1) adding hardware accelerated garbage collection to Heron, based on recent work on FPGA-based concurrent collection [3], (2) a multi-core architecture using the Heron core, and exploiting its parameterised design (Section 4.1) by (3) exploring the possibilities of a heterogeneous architecture, and (4) optimisation of core parameters via static analysis of workloads.

References

[1] Lennart Augustsson. 1992. BWM. 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] Kevin Hammond and Simon L. Peyton Jones. 1992. Profiling Scheduling Strategies on the GRIP Multiprocessor. In *International. Workshop on the Parallel Implementation of Functional Languages.* RWTH Aachen, Germany, 73–98.

[7] 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.

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

[10] SL Peyton Jones, Chris Clack, Jon Salkild, Mark Hardie, and Simon Peyton Jones. 1987. GRIP - a high-performance architecture for parallel graph reduction. In *Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland* (proc ifip conference on functional programming languages and computer architecture, portland ed.). Springer Verlag LNCS 274, 98–112. https://www.microsoft.com/en-us/research/publication/grip-a-high-performance-architecture-for-parallel-graph-reduction/

[11] 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

[12] 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.

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

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

[15] 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 and The International Symposium on Memory Management (ISMM 2002), June 20-21, 2002, Berlin, Germany*, Hans-Juergen Boehm and David Detlefs (Eds.). ACM, 44–55.

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

[17] Craig Ramsay and Robert Stewart. 2023. *Source Code for Heron Core.* https://github.com/haflang/heron

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

Received 1 January 1980; revised 1 January 1980; accepted 1 January 1980