Heron: Modern Hardware Graph Reduction

HAFLANG Project

Craig Ramsay & Rob Stewart

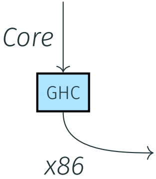
August 2023

Heriot-Watt University

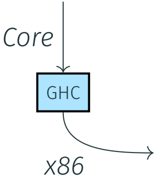
> *I wonder how popular Haskell needs to become for Intel to optimize their processors for my runtime, rather than the other way around.*
>
> — Simon Marlow, 2009

The Graph Reduction Problem

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

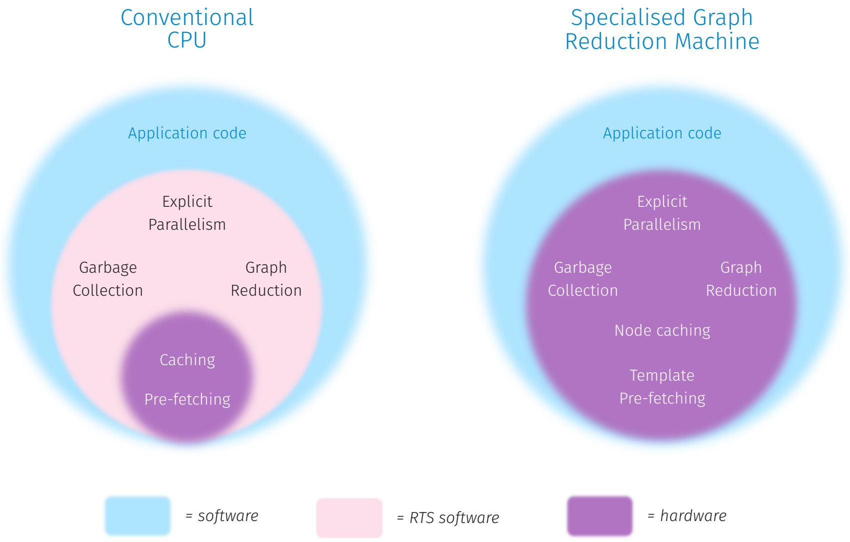


```
add x y z =
  x + y + z
```




```
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)
```

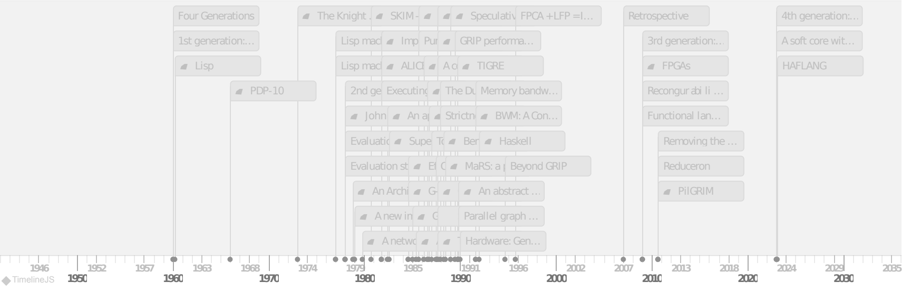
```
Add_add_info: ; Code for `add`
.Lc1br: ; Check for stack space
   leaq -24(%rbp),%rax
   cmpq %r15,%rax
   jb .Lc1bs ; Jump if stack full
.Lc1bt: ; Check for heap space
   addq $32,%r12
   cmpq 856(%r13),%r12
   ja .Lc1bv ; Jump if heap full
.Lc1bu: ; Reduce `add`
   ;; Build `x+y` thunk on heap
   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)
```


Conventional
CPU
Specialised Graph
Reduction Machine
Application code
Explicit
Parallelism
Garbage
Collection
Graph
Reduction
Caching
Pre-fetching
Application code
Explicit
Parallelism
Garbage
Collection
Graph
Reduction
Node caching
Template
Pre-fetching
= software
= RTS software
= hardware

http://haflang.github.io/history

FUNCTIONAL HARDWARE
1924 - 2023

A 100 year history of hardware implementations of functional languages.



by Rob Stewart.

Opportunities

Intra-function parallelism

Usually a victim of the von Neumann bottleneck. Worse for lazy, pure languages.

Inter-function parallelism

Exploiting the purity of functional languages.

Hardened, concurrent run-time system

Tasks like garbage collection usually halt reductions in software implementations.

Opportunities

Intra-function parallelism

Usually a victim of the von Neumann bottleneck. Worse for lazy, pure languages.

Inter-function parallelism

Exploiting the purity of functional languages.

Hardened, concurrent run-time system

Tasks like garbage collection usually halt reductions in software implementations.

A Template Instantiation Machine

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

Our *“assembly”* is a non-strict functional language

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

Tempates are functions:

- λ-lifted to top-level supercombinators
- In an administrative NF

$p ::= \overline{t}$ (Program)

$t ::=$ (Template)

  $\texttt{FUN}\ s\ \alpha\ n\ =$
  $\texttt{let}\ \overline{u}\ \texttt{in}\ v$

$u, v ::=$ (Applications)

  $\texttt{APP}\ \overline{e}$ (Normal application)
  $\mid\ \texttt{CASE}\ c\ \overline{e}$ (Application with case table)
  $\mid\ \texttt{PRIM}\ n\ \overline{e}$ (PRS candidate allocation)

$c ::= \texttt{TAB}\ n$ (Case table pointer)

$e ::=$ (Atoms)

  $\texttt{CON}\ \alpha\ n$ (Constructor tag)
  $\mid\ \texttt{INT}\ n$ (Primitive integers)
  $\mid\ \texttt{PRI}\ \alpha\ \otimes$ (Primitive operation)
  $\mid\ \texttt{FUN}\ s\ \alpha\ n$ (Function pointer)
  $\mid\ \texttt{ARG}\ s\ n$ (Argument pointer)
  $\mid\ \texttt{VAR}\ s\ n$ (Application pointer)
  $\mid\ \texttt{REG}\ n$ (Primitive register pointer)

Applications are wide and tagged

Case alternatives are lifted to contiguous templates

$p ::= \overline{t}$ (Program)

$t ::=$ (Template)

$\texttt{FUN}\ s\ \alpha\ n\ =$
$\texttt{let}\ \overline{u}\ \texttt{in}\ v$

$u, v ::=$ (Applications)

$\texttt{APP}\ \overline{e}$ (Normal application)
$|\ \texttt{CASE}\ c\ \overline{e}$ (Application with case table)
$|\ \texttt{PRIM}\ n\ \overline{e}$ (PRS candidate allocation)

$c ::= \texttt{TAB}\ n$ (Case table pointer)

$e ::=$ (Atoms)

$\texttt{CON}\ \alpha\ n$ (Constructor tag)
$|\ \texttt{INT}\ n$ (Primitive integers)
$|\ \texttt{PRI}\ \alpha\ \otimes$ (Primitive operation)
$|\ \texttt{FUN}\ s\ \alpha\ n$ (Function pointer)
$|\ \texttt{ARG}\ s\ n$ (Argument pointer)
$|\ \texttt{VAR}\ s\ n$ (Application pointer)
$|\ \texttt{REG}\ n$ (Primitive register pointer)

Usual atom suspects...
with four types of pointer[1]

[1]Foreshadowing is a literary device in which a writer gives an advance hint of what is to come later in the story

$p ::= \overline{t}$ (Program)

$t ::=$ (Template)

$\quad \texttt{FUN}\ s\ \alpha\ n\ =$
$\quad\ \ \texttt{let}\ \overline{u}\ \texttt{in}\ v$

$u, v ::=$ (Applications)

$\quad \texttt{APP}\ \overline{e}$ (Normal application)
$\quad |\ \texttt{CASE}\ c\ \overline{e}$ (Application with case table)
$\quad |\ \texttt{PRIM}\ n\ \overline{e}$ (PRS candidate allocation)

$c ::= \texttt{TAB}\ n$ (Case table pointer)

$e ::=$ (Atoms)

$\quad \texttt{CON}\ \alpha\ n$ (Constructor tag)
$\quad |\ \texttt{INT}\ n$ (Primitive integers)
$\quad |\ \texttt{PRI}\ \alpha\ \otimes$ (Primitive operation)
$\quad |\ \texttt{FUN}\ s\ \alpha\ n$ (Function pointer)
$\quad |\ \texttt{ARG}\ s\ n$ (Argument pointer)
$\quad |\ \texttt{VAR}\ s\ n$ (Application pointer)
$\quad |\ \texttt{REG}\ n$ (Primitive register pointer)

Hardware is fixed size...
Beware of $\overline{u}$ and $\overline{e}$

```
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 ]
```

```
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 ]
```

SpineLen

Instantiate on stack

```
FUN ⊤ 2 2 =                              (fromTo#T)
                        ApLen
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 ]
                       SpineLen
```

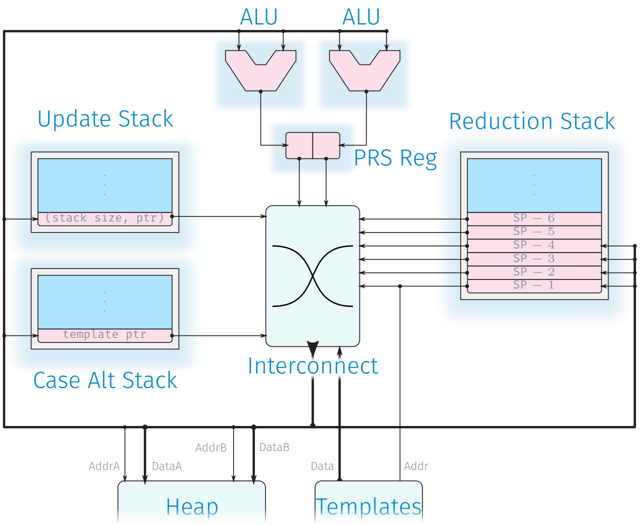
Instantiate on stack

Instantiate on heap

```
FUN ⊤ 2 2 =                    ApLen                        (fromTo#T)

  let   APP [   ARG ⊤ 0,    PRI 2 +,     INT 1 ]
        APP [ FUN ⊤ 2 0,    VAR ⊥ 0,     ARG ⊥ 1 ]    MaxAps

   in   APP [   CON 2 0,    ARG ⊤ 0,     VAR ⊥ 1 ]

                              SpineLen
```

Instantiate on stack

Instantiate on heap


ALU
ALU
Update Stack
PRS Reg
Reduction Stack
(stack size, ptr)
template ptr
SP — 6
SP — 5
SP — 4
SP — 3
SP — 2
SP — 1
Case Alt Stack
Interconnect
AddrA
DataA
AddrB
DataB
Data
Addr
Heap
Templates

New features in Heron:

Parameterised implementation in Clash, for UltraScale+

Zero-constraint templates

Inline case alternatives

Postfix primitive operations

New features in Heron:

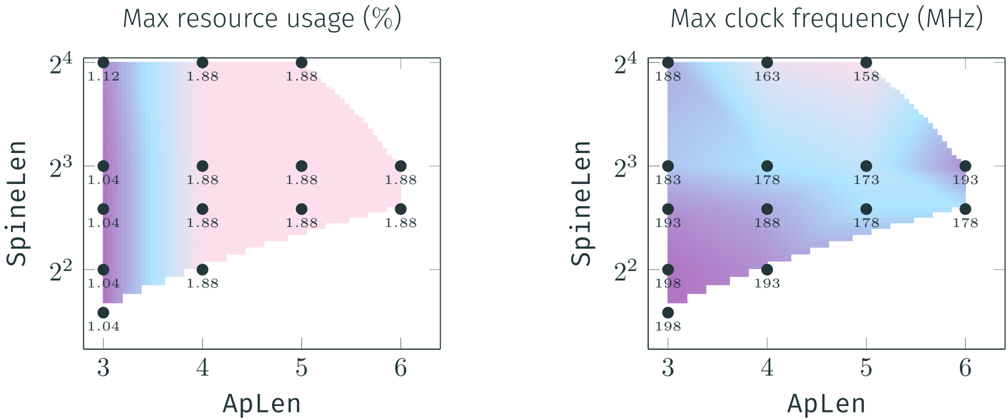
$\times 2\ f_{\text{max}}$, $< 2\%$ max usage on Alveo U280

Mean 6% reduction in cycles (max 17%)

Mean 22% reduction in code size (max 34%)

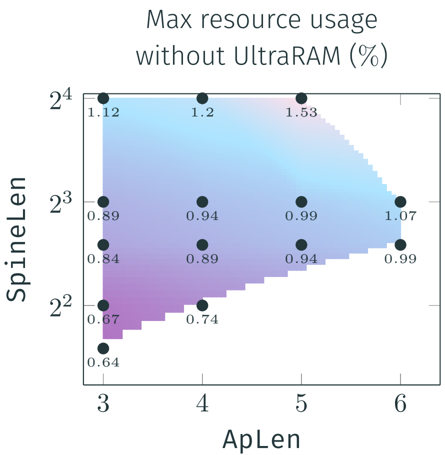
Mean 12% reduction in heap allocs (max 100%)

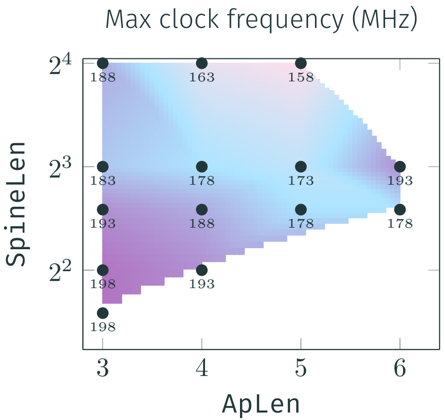
Circuit Results



We’ve now got a parameter space.

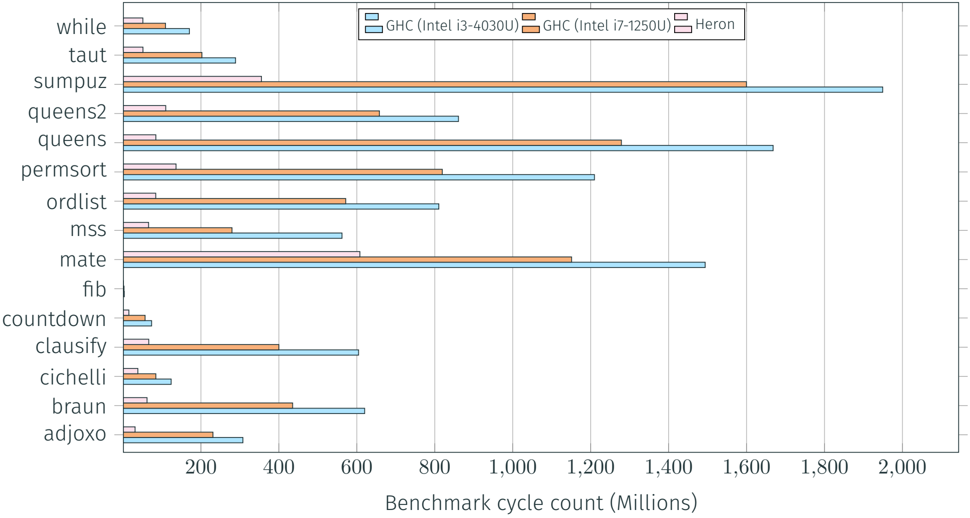
Which, if any, is best?



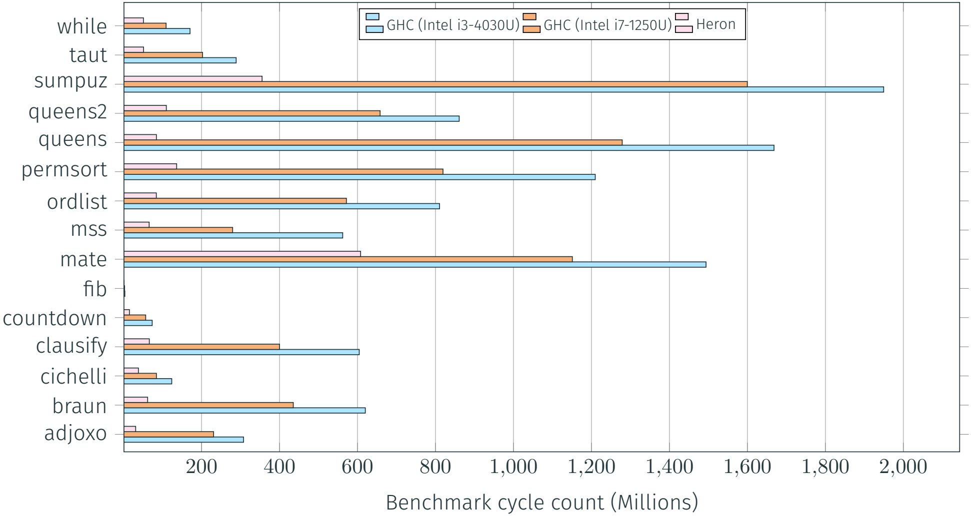


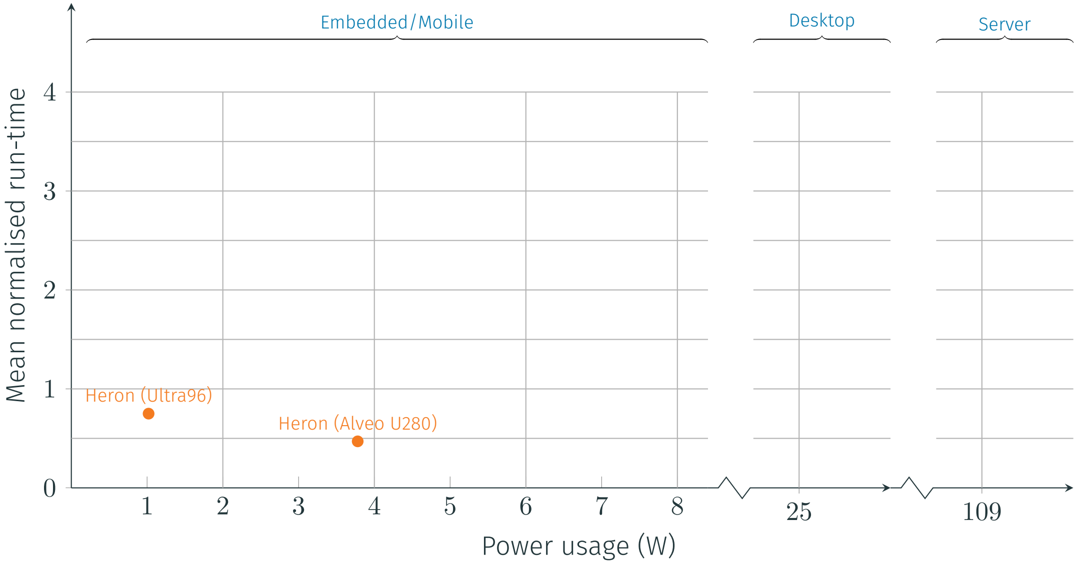
We've now got a parameter space.

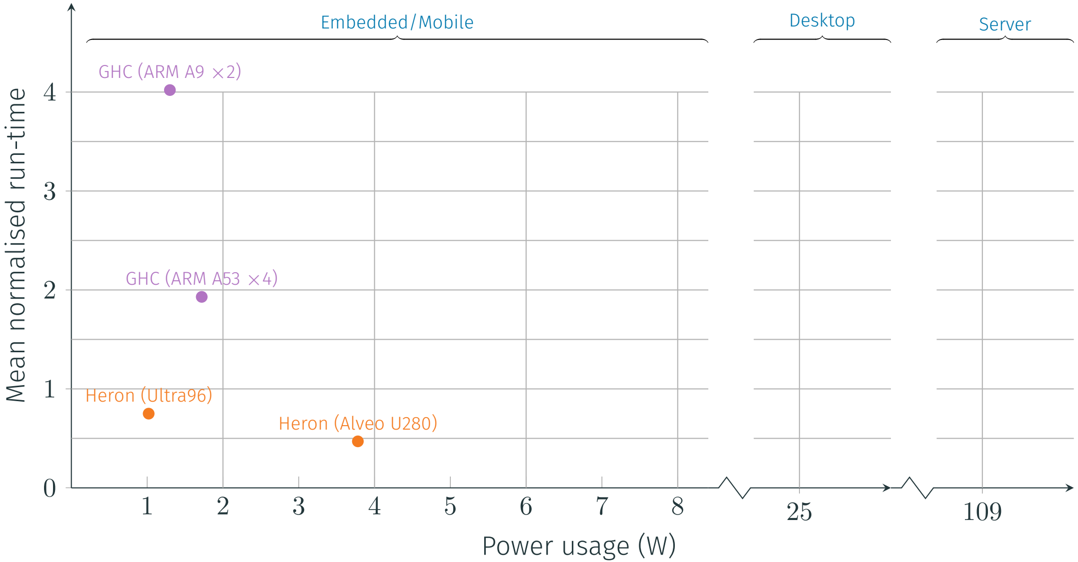
Which, if any, is best?


GHC (Intel i3-4030U)
GHC (Intel i7-1250U)
Heron
while
taut
sumpuz
queens2
queens
permsort
ordlist
mss
mate
fib
countdown
clausify
cichelli
braun
adjoxo
200
400
600
800
1,000
1,200
1,400
1,600
1,800
2,000
Benchmark cycle count (Millions)

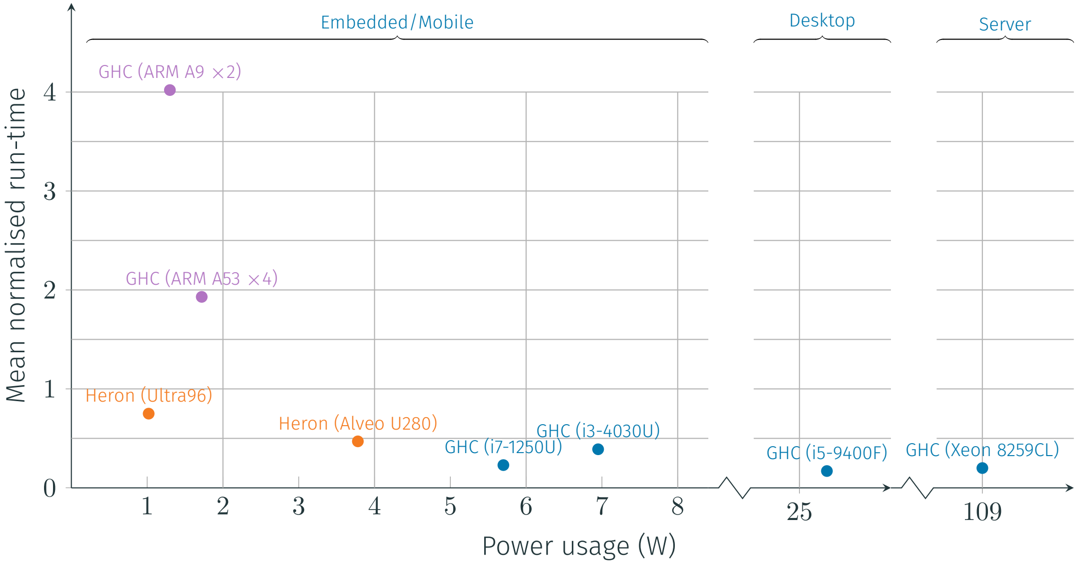
GHC + i3-4030U ≈ ×9.5 cycles and GHC + i7-1250U ≈ ×6.9 cycles

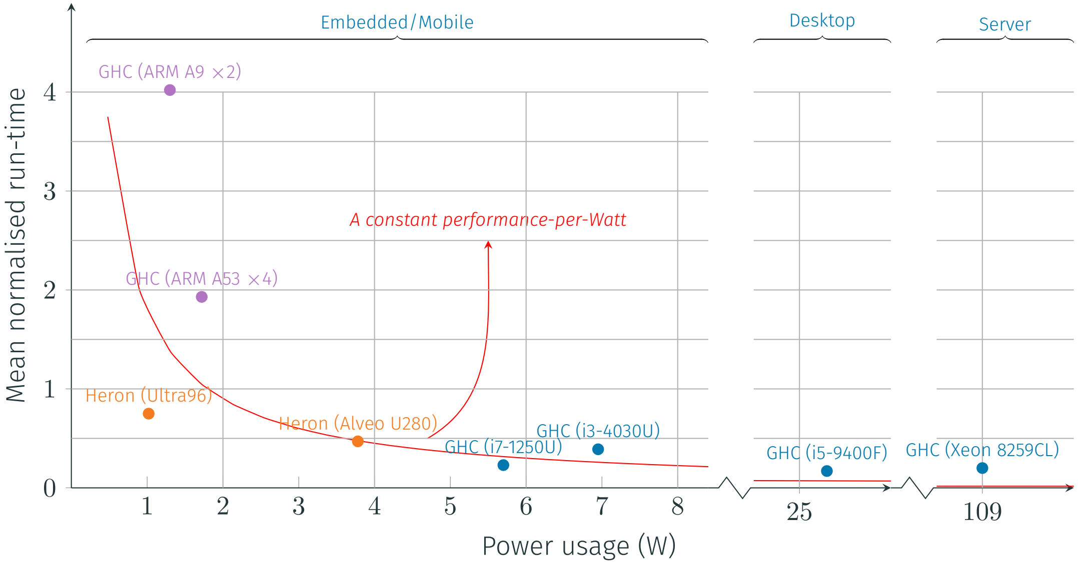



Embedded/Mobile
Desktop
Server
Mean normalised run-time
0
1
2
3
4
Heron (Ultra96)
Heron (Alveo U280)
1
2
3
4
5
6
7
8
25
109
Power usage (W)



Embedded/Mobile
Desktop
Server
GHC (ARM A9 ×2)
GHC (ARM A53 ×4)
Heron (Ultra96)
Heron (Alveo U280)
Mean normalised run-time
0
1
2
3
4
1
2
3
4
5
6
7
8
25
109
Power usage (W)


Embedded/Mobile
Desktop
Server
GHC (ARM A9 ×2)
GHC (ARM A53 ×4)
Heron (Ultra96)
Heron (Alveo U280)
GHC (i7-1250U)
GHC (i3-4030U)
GHC (i5-9400F)
GHC (Xeon 8259CL)
Mean normalised run-time
0
1
2
3
4
1
2
3
4
5
6
7
8
25
109
Power usage (W)



Embedded/Mobile
Desktop
Server
Mean normalised run-time
Power usage (W)
0
1
2
3
4
1
2
3
4
5
6
7
8
25
109
GHC (ARM A9 ×2)
GHC (ARM A53 ×4)
Heron (Ultra96)
Heron (Alveo U280)
A constant performance-per-Watt
GHC (i7-1250U)
GHC (i3-4030U)
GHC (i5-9400F)
GHC (Xeon 8259CL)

What’s next?

Call To Action

CAFP mailing list

(Computer Architectures for Functional Programming)

`https://groups.google.com/g/cafp`

Appendix

```
add x y z = (x + y) + z;
```

$$\Downarrow$$

```
add x y z = (x + y) + z;
```

$$\Downarrow$$

```
FUN ⊤ 3 0 =                                                  (add)
  let   APP [  ARG ⊥ 0,    PRI 2 +,    ARG ⊥ 1 ]
   in   APP [  VAR ⊥ 0,    PRI 2 +,    ARG ⊥ 2 ]
```

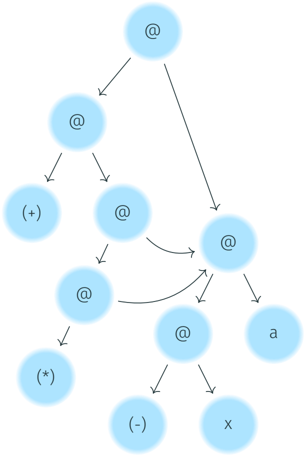
```
fromTo n m = case n <= m of {
  False -> Nil;
  True  -> Cons n (fromTo (n + 1) m);
};
```

$$\Downarrow$$

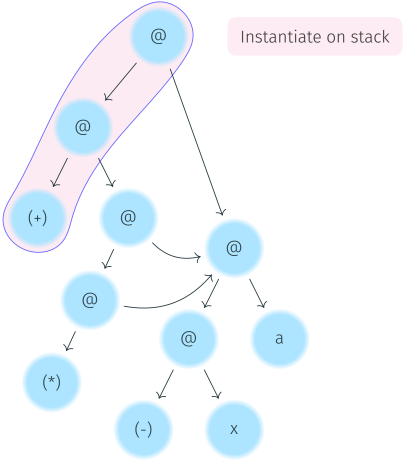
```
fromTo n m = case n <= m of {
  False -> Nil;
  True  -> Cons n (fromTo (n + 1) m);
};
```

$$\Downarrow$$

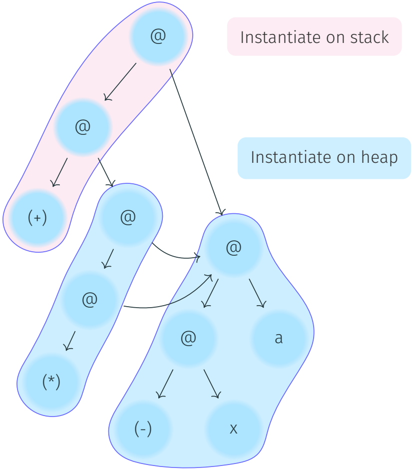
```
FUN ⊤ 2 0 =                                                    (fromTo)
   let   APP [   ARG ⊤ 0,    PRI 2 <=,     ARG ⊤ 1 ]
    in   CASE (TAB 1)
            [   VAR ⊥ 0,    ARG ⊤ 0,     ARG ⊤ 1 ]

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 ]
```



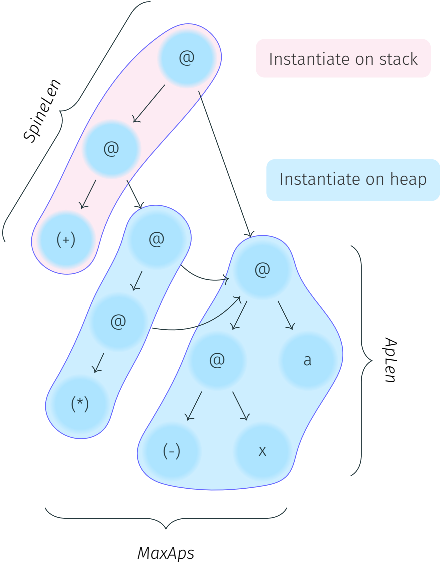

```
y a x = let b = (-) x a
            c = (*) b b
        in (+) c b
```




```
y a x = let b = (-) x a
            c = (*) b b
        in (+) c b
```




```
y a x = let b = (-) x a
            c = (*) b b
        in (+) c b
```




```
y a x = let b = (-) x a
            c = (*) b b
        in (+) c b
```