Native Offload of Haskell Repa Programs to Integrated GPUs

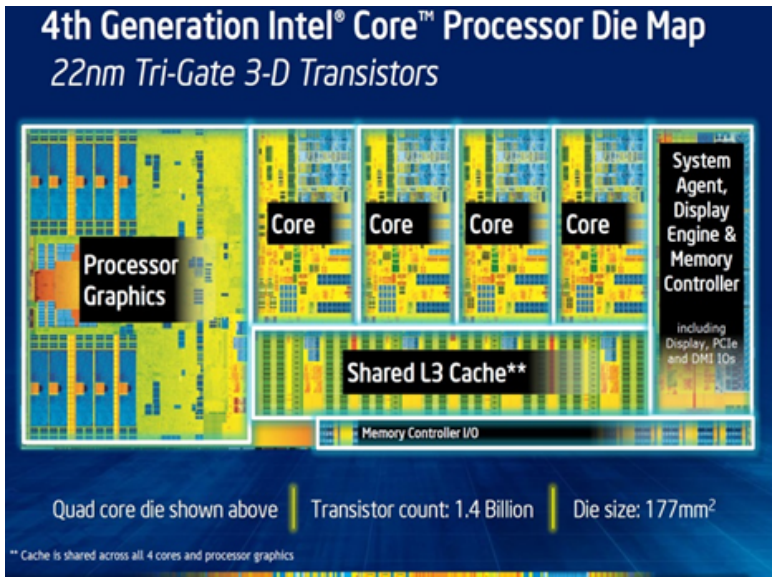
Hai (Paul) Liu
with Laurence Day, Neal Glew, Todd Anderson, Rajkishore Barik
Intel Labs. September 28, 2016

General purpose computing on integrated GPUs

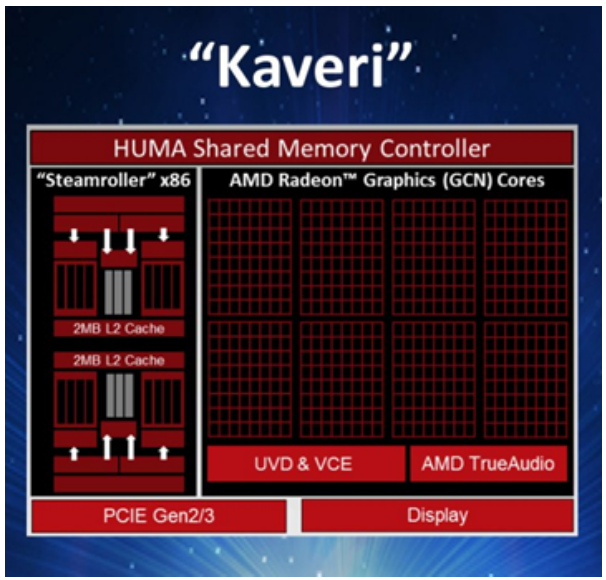
More than 90% of processors shipping today include a GPU on die.

Lower energy use is a key design goal.

The CPU and GPU share physical memory (DRAM), may share Last Level Cache (LLC).



(a) Intel Haswell



(b) AMD Kaveri

GPU differences from CPU

CPUs optimized for latency, GPUs for throughput.

- CPUs: deep caches, OOO cores, sophisticated branch predictors
- GPUs: transistors spent on many slim cores running in parallel

GPU differences from CPU

CPUs optimized for latency, GPUs for throughput.

- CPUs: deep caches, OOO cores, sophisticated branch predictors
- GPUs: transistors spent on many slim cores running in parallel

Single Instruction Multiple Thread (SIMT) execution.

- Work-items (logical threads) are partitioned into work-groups
- The work-items of a work-group execute together in near lock-step
- Allows several ALUs to share one instruction unit

GPU differences from CPU

CPUs optimized for latency, GPUs for throughput.

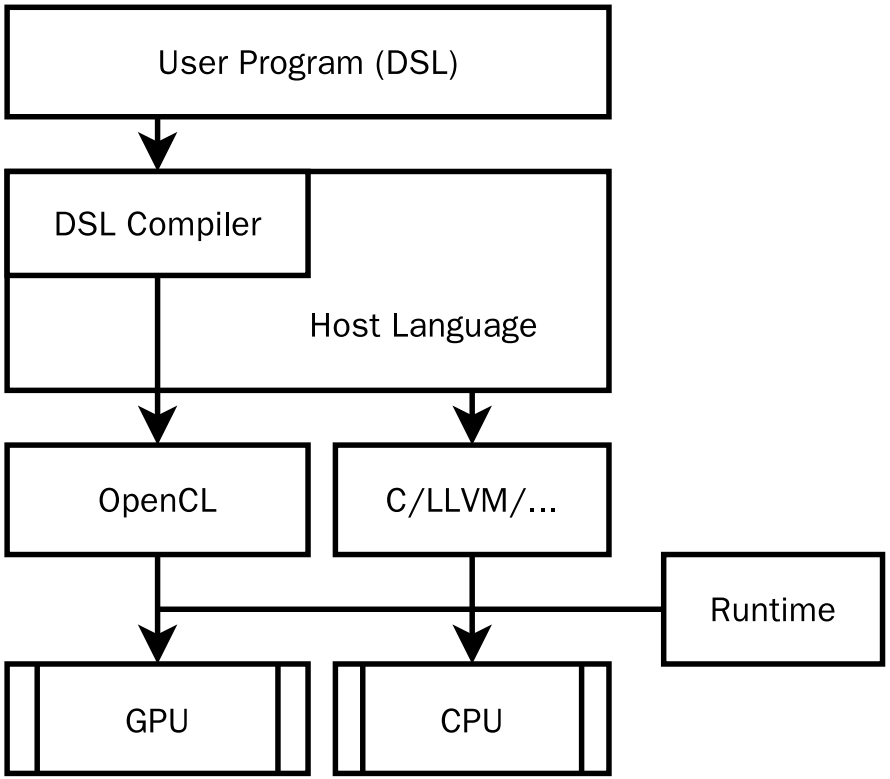
- CPUs: deep caches, OOO cores, sophisticated branch predictors
- GPUs: transistors spent on many slim cores running in parallel

Single Instruction Multiple Thread (SIMT) execution.

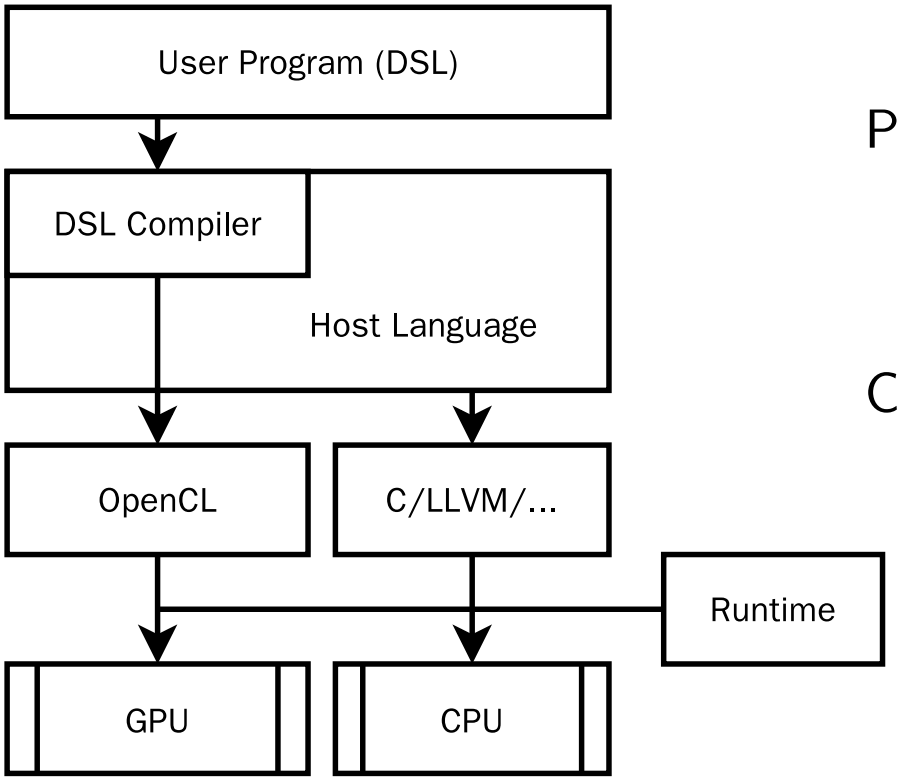
- Work-items (logical threads) are partitioned into work-groups
- The work-items of a work-group execute together in near lock-step
- Allows several ALUs to share one instruction unit

Shallow execution pipelines, highly multi-threaded, shared high-speed local memory, serial execution of branch codes, . . .

Programming GPUs with DSLs



Programming GPUs with DSLs



Pros:

- High-level constructs and operators.
- Domain-specific optimizations.

Cons:

- Barriers between a DSL and its host language.
- Re-implementation of general program optimizations.

Alternative approach: native offload

Directly compile a sub-set of host language to target GPUs.

- less explored, especially for functional languages.
- enjoy all optimizations available to the host language.
- target devices with shared virtual memory (SVM).

Alternative approach: native offload

Directly compile a sub-set of host language to target GPUs.

- less explored, especially for functional languages.
- enjoy all optimizations available to the host language.
- target devices with shared virtual memory (SVM).

This talk: native offload of Haskell Repa programs.

The Haskell Repa library

A popular data parallel array programming library.

```
import Data.Array.Repa as R

a :: Array U DIM2 Int
a = R.fromListUnboxed (Z :. 5 :. 10) [0..49]

b :: Array D DIM2 Int
b = R.map (^2) (R.map (*4) a)

c :: IO (Array U DIM2 Int)
c = R.computeP b
```

The Haskell Repa library

A popular data parallel array programming library.

```
import Data.Array.Repa as R

a :: Array U DIM2 Int
a = R.fromListUnboxed (Z :. 5 :. 10) [0..49]

b :: Array D DIM2 Int
b = R.map (^2) (R.map (*4) a)

c :: IO (Array U DIM2 Int)
c = R.~~computeP~~computeG b
```

Maybe we can run the same program on GPUs too!

Introducing computeG

```
computeS :: (Shape sh, Unbox e) ⇒
            Array D sh e → Array U sh e

computeP :: (Shape sh, Unbox e, Monad m) ⇒
            Array D sh e → m (Array U sh e)

computeG :: (Shape sh, Unbox e, Monad m) ⇒
            Array D sh e → m (Array U sh e)
```

In theory, all Repa programs should also run on GPUs.

Introducing computeG

```
computeS :: (Shape sh, Unbox e) ⇒
            Array D sh e → Array U sh e

computeP :: (Shape sh, Unbox e, Monad m) ⇒
            Array D sh e → m (Array U sh e)

computeG :: (Shape sh, Unbox e, Monad m) ⇒
            Array D sh e → m (Array U sh e)
```

In theory, all Repa programs should also run on GPUs.
In practice, only a restricted subset is allowed to compile and run.

Implementing computeG

We introduce a primitive operator offload#:

```
offload# :: Int → (Int → State# s → State# s)
            → State# s → State# s
```

that takes three parameters:

1. the upper bound of a range.
2. a kernel function that maps an index in the range to a stateful computation.
3. a state.

offload# is enough to implement computeG.

Implementation overview

HRC Intel Labs Haskell Research Compiler that uses GHC as frontend (Haskell'13).

Implementation overview

HRC Intel Labs Haskell Research Compiler that uses GHC as frontend (Haskell'13).

Concord C++ based heterogeneous computing framework that compiles to OpenCL (CGO'14).

Implementation overview

HRC Intel Labs Haskell Research Compiler that uses GHC as frontend (Haskell'13).

Concord C++ based heterogeneous computing framework that compiles to OpenCL (CGO'14).

1. Modify Repa to implement `computeG` in terms of `offload#`.

Implementation overview

HRC Intel Labs Haskell Research Compiler that uses GHC as frontend (Haskell'13).

Concord C++ based heterogeneous computing framework that compiles to OpenCL (CGO'14).

1. Modify Repa to implement `computeG` in terms of `offload#`.
2. Modify GHC to introduce the `offload#` primitive and its type.

Implementation overview

HRC Intel Labs Haskell Research Compiler that uses GHC as frontend (Haskell'13).

Concord C++ based heterogeneous computing framework that compiles to OpenCL (CGO'14).

1. Modify Repa to implement `computeG` in terms of `offload#`.
2. Modify GHC to introduce the `offload#` primitive and its type.
3. Modify HRC to intercept calls to `offload#`.

Implementation overview

HRC Intel Labs Haskell Research Compiler that uses GHC as frontend (Haskell'13).

Concord C++ based heterogeneous computing framework that compiles to OpenCL (CGO'14).

1. Modify Repa to implement `computeG` in terms of `offload#`.
2. Modify GHC to introduce the `offload#` primitive and its type.
3. Modify HRC to intercept calls to `offload#`.
4. In HRC's outputter, dump the kernel function to a C file.

Implementation overview

HRC Intel Labs Haskell Research Compiler that uses GHC as frontend (Haskell'13).

Concord C++ based heterogeneous computing framework that compiles to OpenCL (CGO'14).

1. Modify Repa to implement `computeG` in terms of `offload#`.
2. Modify GHC to introduce the `offload#` primitive and its type.
3. Modify HRC to intercept calls to `offload#`.
4. In HRC's outputter, dump the kernel function to a C file.
5. Use Concord to compile C kernel to OpenCL.

Implementation overview

HRC Intel Labs Haskell Research Compiler that uses GHC as frontend (Haskell'13).

Concord C++ based heterogeneous computing framework that compiles to OpenCL (CGO'14).

1. Modify Repa to implement `computeG` in terms of `offload#`.
2. Modify GHC to introduce the `offload#` primitive and its type.
3. Modify HRC to intercept calls to `offload#`.
4. In HRC's outputter, dump the kernel function to a C file.
5. Use Concord to compile C kernel to OpenCL.
6. Replace `offload#` with call into Concord runtime.

What is the catch?

What is the catch?

Not all Repa functions can be offloaded.

What is the catch?

Not all Repa functions can be offloaded.

The following restrictions are enforced at compile time:

- kernel function must be statically known.
- no allocation/thunk evals/recursion/exception in the kernel.
- only function calls into Concord or OpenCL are allowed.

What is the catch?

Not all Repa functions can be offloaded.

The following restrictions are enforced at compile time:

- kernel function must be statically known.
- no allocation/thunk evals/recursion/exception in the kernel.
- only function calls into Concord or OpenCL are allowed.

Additionally:

- All memory are allocated in the SVM region.
- No garbage collection during offload call.

Benchmarking

A Variety of 9 embarrassingly parallel programs written using Repa.
A majority come from the "Haskell Gap" study (IFL'13).

Hardware:

Processor	Cores	Clock	Hyper-thread	Peak Perf.
HD4600 (GPU)	20	1.3GHz	No	432 GFLOPs
Core i7-4770	4	3.4GHz	Yes	435 GFLOPs
Xeon E5-4650	32	2.7GHz	No	2970 GFLOPs

Benchmarking

A Variety of 9 embarrassingly parallel programs written using Repa.
A majority come from the "Haskell Gap" study (IFL'13).

Hardware:

Processor	Cores	Clock	Hyper-thread	Peak Perf.
HD4600 (GPU)	20	1.3GHz	No	432 GFLOPs
Core i7-4770	4	3.4GHz	Yes	435 GFLOPs
Xeon E5-4650	32	2.7GHz	No	2970 GFLOPs

Average relative speed-up (bigger is better):

	HD4600 (GPU)	Core i7-4770	Xeon E5-4650
Geometric Mean	6.9	7.0	18.8

What we have learned

Laziness is not a problem most of the time for Repa programs.

Sample: ANormStrict IR

```
lv311252_ia2NL_tslam^* = \ <; lv311232_ia2NL> →
  let
    <lv311233_s1a2NM_tsscr> = ghczmprim:GHCziPrim.noDuplicatezh
        <lv5772_main:Main.ghczmprim:GHCziPrim.RealWorld0>
    lv311245_v8896^ = thunk <; >
      let
        <lv311234_v8896_tsscr> = ghczmprim:GHCziPrim.remIntzh
            <lv311232_ia2NL, lv236843_main:Main.y1s36S>
        <lv311235_v8896_tsscr> = ghczmprim:GHCziPrim.quotIntzh
            <lv311232_ia2NL, lv236843_main:Main.y1s36S>
        <lv311236_atmp> = n22_ghczmprim:GHCziTypes.Izh <lv311235_v8896_tsscr>
        lv311237_v8893^ = thunk <; > <lv311236_atmp>
        <lv322918_atmp> = n15_repazm3zi2zi2zi2:DataziArrayziRepaziIndex.ZCzi
            <lv5929_main:Main.repazm3zi2zi2zi2:DataziArrayziRepaziIndex.ZZ111,
            lv311237_v8893>
        lv311240_v8894^ = thunk <; > <lv322918_atmp>
        <lv311241_atmp> = n22_ghczmprim:GHCziTypes.Izh <lv311234_v8896_tsscr>
        lv311242_v8895^ = thunk <; > <lv311241_atmp>
        <lv322921_atmp> = n15_repazm3zi2zi2zi2:DataziArrayziRepaziIndex.ZCzi
            <lv311240_v8894, lv311242_v8895>
      in <lv322921_atmp>
    <lv311247_v8904_tsscr> = lv332264_main:Main.fa1ZZM_ubx <lv311245_v8896>
    <lv311250_v8904> =
      case lv311247_v8904_tsscr of
        {n22_ghczmprim:GHCziTypes.Izh lv311248_xzha30Q →
          let <lv311249_atmp> = ghczmprim:GHCziPrim.initUnboxedIntArrayzh
                  <lv311225_ipv1a222, lv311232_ia2NL, lv311248_xzha30Q,
     lv311233_s1a2NM_tsscr>
          in <lv311249_atmp>}
    <lv311251_atmp> = (0 :: primtype #int)
  in <lv311251_atmp>
lv311253_v8908^ = thunk <; > <lv311252_ia2NL_tslam>
<lv311254_sa1ZZT_tsscr> = ghczmprim:GHCziPrim.offloadzh
    <lv236850_main:Main.nzhs36W, lv311253_v8908, lv311230_ipv2a2NE>
```

Sample: ANormStrict IR

```
lv311252_ia2NL_tslam^* = \ <; lv311232_ia2NL> →
  let
    <lv311233_s1a2NM_tsscr> = ghczmprim:GHCziPrim.noDuplicatezh
        <lv5772_main:Main.ghczmprim:GHCziPrim.RealWorld0>
    lv311245_v8896^ = thunk <; >
      let
        <lv311234_v8896_tsscr> = ghczmprim:GHCziPrim.remIntzh
            <lv311232_ia2NL, lv236843_main:Main.y1s36S>
        <lv311235_v8896_tsscr> = ghczmprim:GHCziPrim.quotIntzh
            <lv311232_ia2NL, lv236843_main:Main.y1s36S>
        <lv311236_atmp> = n22_ghczmprim:GHCziTypes.Izh <lv311235_v8896_tsscr>
        lv311237_v8893^ = thunk <; > <lv311236_atmp>
        <lv322918_atmp> = n15_repazm3zi2zi2zi2:DataziArrayziRepaziIndex.ZCzi
            <lv5929_main:Main.repazm3zi2zi2zi2:DataziArrayziRepaziIndex.ZZ111,
            lv311237_v8893>
        lv311240_v8894^ = thunk <; > <lv322918_atmp>
        <lv311241_atmp> = n22_ghczmprim:GHCziTypes.Izh <lv311234_v8896_tsscr>
        lv311242_v8895^ = thunk <; > <lv311241_atmp>
        <lv322921_atmp> = n15_repazm3zi2zi2zi2:DataziArrayziRepaziIndex.ZCzi
            <lv311240_v8894, lv311242_v8895>
      in <lv322921_atmp>
    <lv311247_v8904_tsscr> = lv332264_main:Main.fa1ZZM_ubx <lv311245_v8896>
    <lv311250_v8904> =
      case lv311247_v8904_tsscr of
        {n22_ghczmprim:GHCziTypes.Izh lv311248_xzha30Q →
          let <lv311249_atmp> = ghczmprim:GHCziPrim.initUnboxedIntArrayzh
                  <lv311225_ipv1a222, lv311232_ia2NL, lv311248_xzha30Q,
     lv311233_s1a2NM_tsscr>
          in <lv311249_atmp>}
    <lv311251_atmp> = (0 :: primtype #int)
  in <lv311251_atmp>
lv311253_v8908^ = thunk <; > <lv311252_ia2NL_tslam>
<lv311254_sa1ZZT_tsscr> = ghczmprim:GHCziPrim.offloadzh
    <lv236850_main:Main.nzhs36W, lv311253_v8908, lv311230_ipv2a2NE>
```

Sample: ANormStrict IR

```
lv311252_ia2NL_tslam^* = \ <; lv311232_ia2NL> →
  let
    <lv311233_s1a2NM_tsscr> = ghczmprim:GHCziPrim.noDuplicatezh
        <lv5772_main:Main.ghczmprim:GHCziPrim.RealWorld0>
    lv311245_v8896^ = thunk <; >
      let
        <lv311234_v8896_tsscr> = ghczmprim:GHCziPrim.remIntzh
            <lv311232_ia2NL, lv236843_main:Main.y1s36S>
        <lv311235_v8896_tsscr> = ghczmprim:GHCziPrim.quotIntzh
            <lv311232_ia2NL, lv236843_main:Main.y1s36S>
        <lv311236_atmp> = n22_ghczmprim:GHCziTypes.Izh <lv311235_v8896_tsscr>
        lv311237_v8893^ = thunk <; > <lv311236_atmp>
        <lv322918_atmp> = n15_repazm3zi2zi2zi2:DataziArrayziRepaziIndex.ZCzi
            <lv5929_main:Main.repazm3zi2zi2zi2:DataziArrayziRepaziIndex.ZZ111,
            lv311237_v8893>
        lv311240_v8894^ = thunk <; > <lv322918_atmp>
        <lv311241_atmp> = n22_ghczmprim:GHCziTypes.Izh <lv311234_v8896_tsscr>
        lv311242_v8895^ = thunk <; > <lv311241_atmp>
        <lv322921_atmp> = n15_repazm3zi2zi2zi2:DataziArrayziRepaziIndex.ZCzi
            <lv311240_v8894, lv311242_v8895>
      in <lv322921_atmp>
    <lv311247_v8904_tsscr> = lv332264_main:Main.fa1ZZM_ubx <lv311245_v8896>
    <lv311250_v8904> =
      case lv311247_v8904_tsscr of
        {n22_ghczmprim:GHCziTypes.Izh lv311248_xzha30Q →
          let <lv311249_atmp> = ghczmprim:GHCziPrim.initUnboxedIntArrayzh
                  <lv311225_ipv1a222, lv311232_ia2NL, lv311248_xzha30Q,
     lv311233_s1a2NM_tsscr>
          in <lv311249_atmp>}
    <lv311251_atmp> = (0 :: primtype #int)
  in <lv311251_atmp>
lv311253_v8908^ = thunk <; > <lv311252_ia2NL_tslam>
<lv311254_sa1ZZT_tsscr> = ghczmprim:GHCziPrim.offloadzh
    <lv236850_main:Main.nzhs36W, lv311253_v8908, lv311230_ipv2a2NE>
```

Sample: MIL IR

```
a2NL_tslam_code =
Code^*(CcCode; lv344572_ia2NL_tslam, lv311232_ia2NL){PIw} : (SInt32)
{
  Entry L12630
  L12630()[]
    lv344570_ipv1a222 = lv344572_ia2NL_tslam [sf:1];
    lv344571_main:Main.fa1ZZM_ubx = lv344572_ia2NL_tslam [sf:2];
    Call(ev340941_ihrNoDuplicate) ?{} () → () L5152 {I}
  L5152()[L12630]
    lv344549_main:Main.rbs366 = lv344571_main:Main.fa1ZZM_ubx [sf:1];
    lv344551_main:Main.arrzhs36y = lv344571_main:Main.fa1ZZM_ubx [sf:2];
    lv333435_v8860 = SInt32Plus(lv344549_main:Main.rbs366, lv311232_ia2NL);
    lv333436_v8861 = lv344551_main:Main.arrzhs36y [sv:lv333435_v8860];
    lv352231_a7s356 = SInt32Times(lv333436_v8861, lv333436_v8861);
    lv333439_v8865 = SInt32Times(lv352231_a7s356, S32(16));
    !lv344570_ipv1a222 [sv:lv311232_ia2NL] ← lv333439_v8865;
    Return(S32(0))
}

{
  ....
  L10195()[L5150]
    lv311252_ia2NL_tslam = <<L; b32+, r+, r+>; gv344568_ia2NL_tslam_code,
                            lv344566_, lv255299_xa1dW_tslam>;
    lv311253_v8908 = ThunkMkVal(lv311252_ia2NL_tslam);
    Call(ev344585_pLsrPrimGHCOffloadzh) ?{} (S32(50), lv311253_v8908) → ()
  L5158 {Agrw}
  ....
}
```

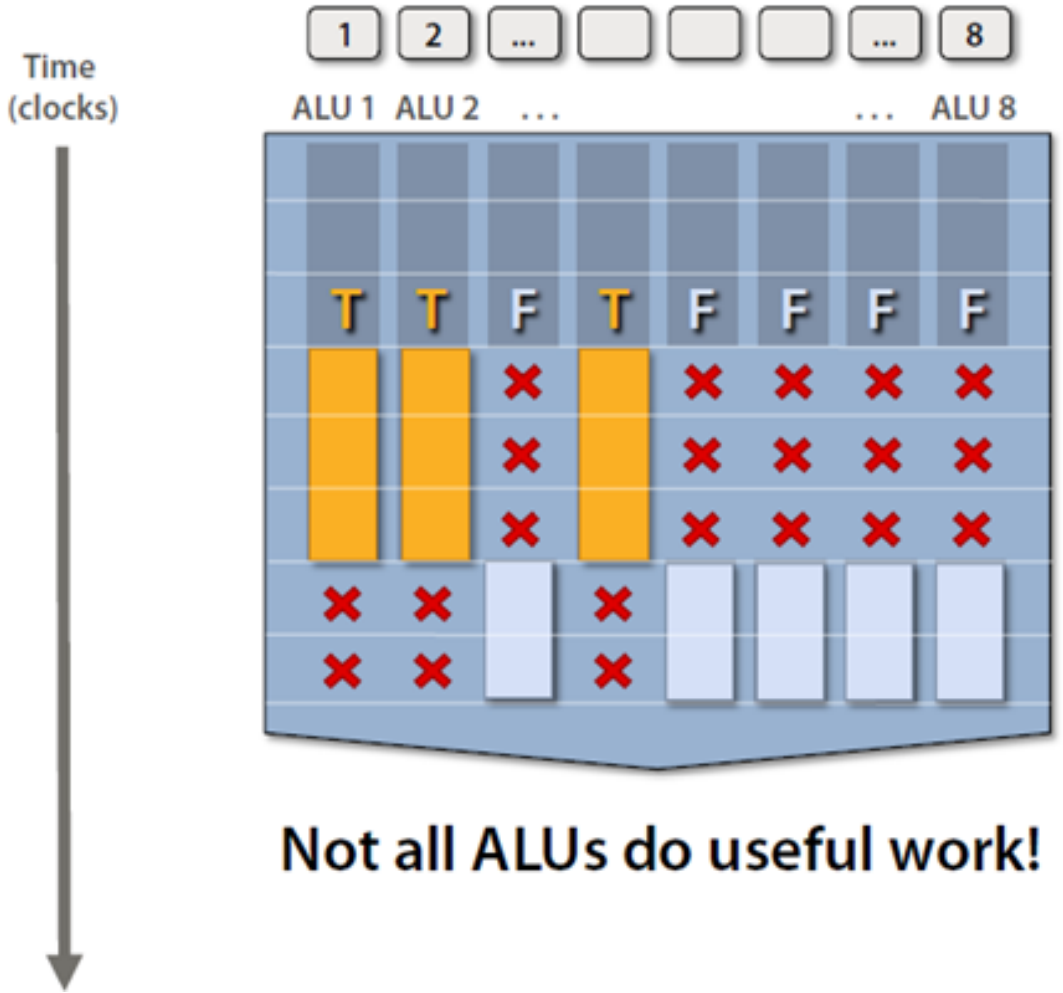
Sample: kernel code in C

```
static sint32 v344568_ia2NL_tslam_code(PlsrObjectB v344572_ia2NL_tslam,
                                       sint32 v311232_ia2NL)
{
  sint32 v333435_v8860;
  sint32 v333436_v8861;
  sint32 v333439_v8865;
  sint32 v344549_mainZCMainzirbs366;
  PlsrPAny v344551_mainZCMainziarrzzhs36y;
  PlsrPAny v344570_ipv1a222;
  PlsrPAny v344571_mainZCMainzifa1ZZZZM_ubx;
  sint32 v352231_a7s356;
  v344570_ipv1a222 = pLsrObjectField (v344572_ia2NL_tslam, 8, PlsrPAny (*));
  v344571_mainZCMainzifa1ZZZZM_ubx =
        pLsrObjectField (v344572_ia2NL_tslam, 12, PlsrPAny (*));
  ihrNoDuplicate ();
  v344549_mainZCMainzirbs366 =
        pLsrObjectField (v344571_mainZCMainzifa1ZZZZM_ubx, 8, sint32 (*));
  v344551_mainZCMainziarrzzhs36y =
        pLsrObjectField (v344571_mainZCMainzifa1ZZZZM_ubx, 12, PlsrPAny (*));
  pLsrPrimPSInt32Plus(v333435_v8860, v344549_mainZCMainzirbs366, v311232_ia2NL);
  v333436_v8861 = pLsrObjectExtra (v344551_mainZCMainziarrzzhs36y, 8,
        sint32 (*), 4, v333435_v8860);
  pLsrPrimPSInt32Times (v352231_a7s356, v333436_v8861, v333436_v8861);
  pLsrPrimPSInt32Times (v333439_v8865, v352231_a7s356, 16);
  pLsrObjectExtra (v344570_ipv1a222, 8, sint32 (*), 4, v311232_ia2NL) =
        v333439_v8865;
  return 0;
}
static void v344568_ia2NL_tslam_code_kernel(void (*env), size_t i, void (*p))
{
  v344568_ia2NL_tslam_code ((PlsrObjectB)env, (sint32)i);
}
void v344568_ia2NL_tslam_code_offload(sint32 size, PlsrObjectB env)
{
  offload ((size_t)size, (void (*))env, v344568_ia2NL_tslam_code_kernel, 0);
}
```

What we have also learned

Many optimizations for CPUs also help GPUs.

Branch divergence hurts GPU performance




```
<unconditional
 shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
 shader code>
```

Branching problem with GHC

Cause:

GHC tends to inline aggressively into leaves,

Branching problem with GHC

Cause:

GHC tends to inline aggressively into leaves,

. . . which creates branches that has many lines of code,

Branching problem with GHC

Cause:

GHC tends to inline aggressively into leaves,

. . . which creates branches that has many lines of code,

. . . but mostly identical (modulo renaming).

Branching problem with GHC

Cause:

GHC tends to inline aggressively into leaves,

. . . which creates branches that has many lines of code,

. . . but mostly identical (modulo renaming).

Consequence:

No significant cost when executing sequntially on CPU,

Branching problem with GHC

Cause:

GHC tends to inline aggressively into leaves,

. . . which creates branches that has many lines of code,

. . . but mostly identical (modulo renaming).

Consequence:

No significant cost when executing sequntially on CPU,

. . . but bad for both:

- SIMD vectorization on CPU, and
- SIMT execution on GPU.

Branching problem with GHC

Cause:

GHC tends to inline aggressively into leaves,

. . . which creates branches that has many lines of code,

. . . but mostly identical (modulo renaming).

Consequence:

No significant cost when executing sequntially on CPU,

. . . but bad for both:

- SIMD vectorization on CPU, and
- SIMT execution on GPU.

Solution:

Branch to `CMOV` conversion that helps both CPU and GPU.

But not all is rosy . . .

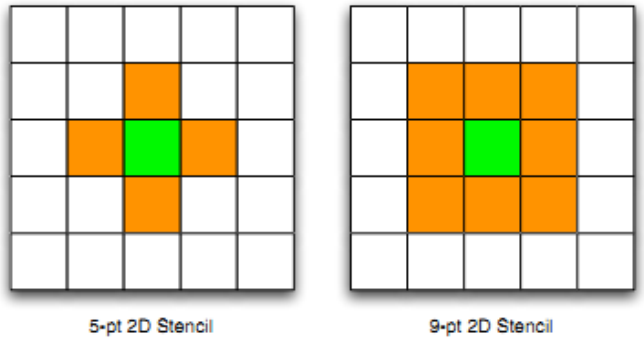
Sometimes we must optimize differently!

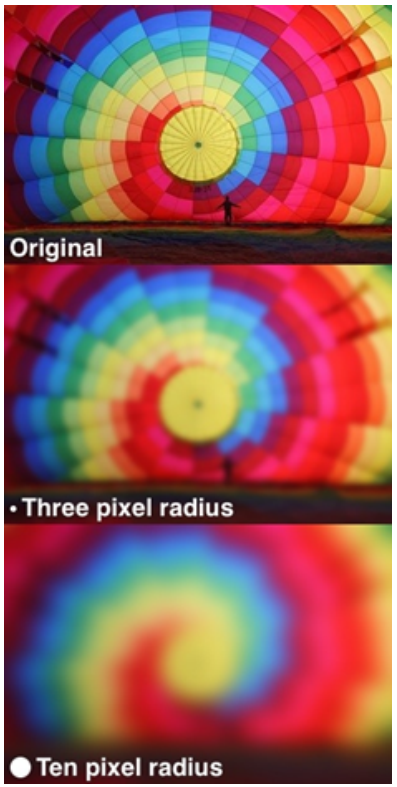
Example: 2D Convolution

Operation $\star$ on 2D image is defined by:

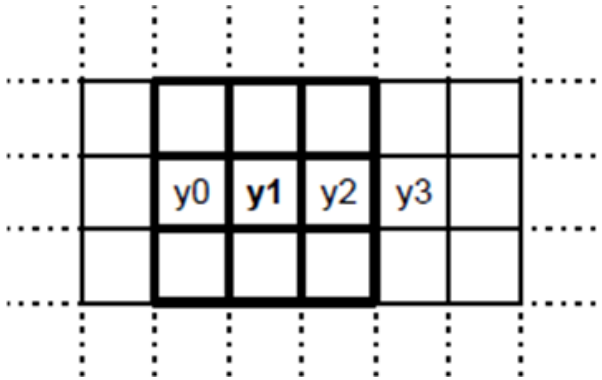
$$(A \star K)(x, y) = \sum_i \sum_j A(x + i, y + j)K(i, j)$$

A is the image being processed.
K is the stencil kernel, 3×3, 1×5, etc.





How Repa handles blocking

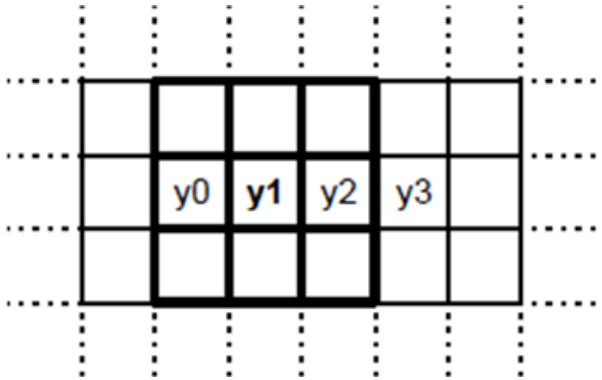


B. Lippmeier and G. Keller (Haskell'11)

- group block-reads of adjacent input pixels
- Global Value Numbering (GVN)

Good sequential speed-up for CPU.

How Repa handles blocking



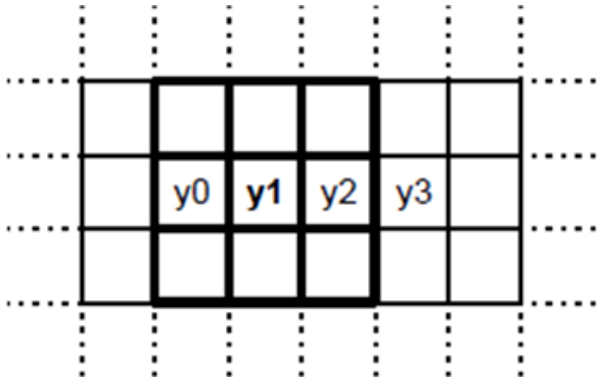
B. Lippmeier and G. Keller (Haskell'11)

- group block-reads of adjacent input pixels
- Global Value Numbering (GVN)

Good sequential speed-up for CPU.

For SIMD?

How Repa handles blocking



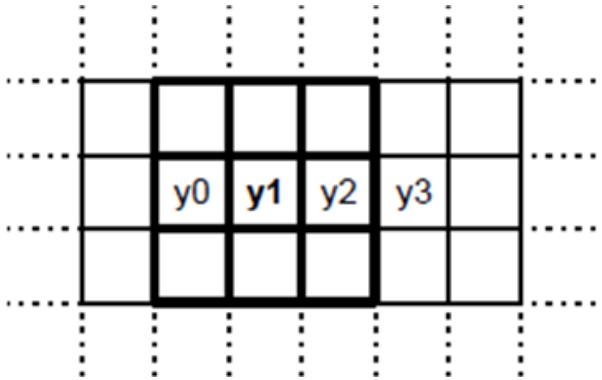
B. Lippmeier and G. Keller (Haskell'11)

- group block-reads of adjacent input pixels
- Global Value Numbering (GVN)

Good sequential speed-up for CPU.

For SIMD? Block vertically instead.

How Repa handles blocking



B. Lippmeier and G. Keller (Haskell'11)

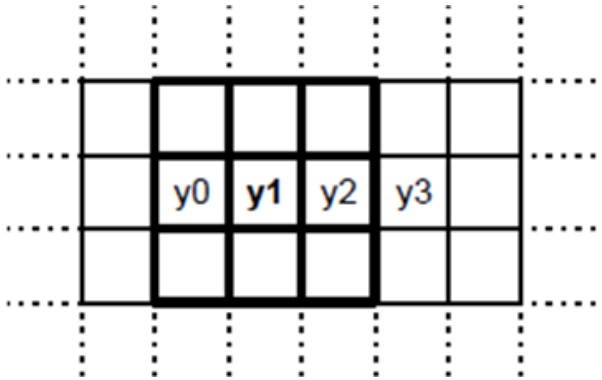
- group block-reads of adjacent input pixels
- Global Value Numbering (GVN)

Good sequential speed-up for CPU.

For SIMD? Block vertically instead.

For GPU?

How Repa handles blocking



B. Lippmeier and G. Keller (Haskell'11)

- group block-reads of adjacent input pixels
- Global Value Numbering (GVN)

Good sequential speed-up for CPU.

For SIMD? Block vertically instead.

For GPU? HUGE slowdown!

Conclusion and Take Away

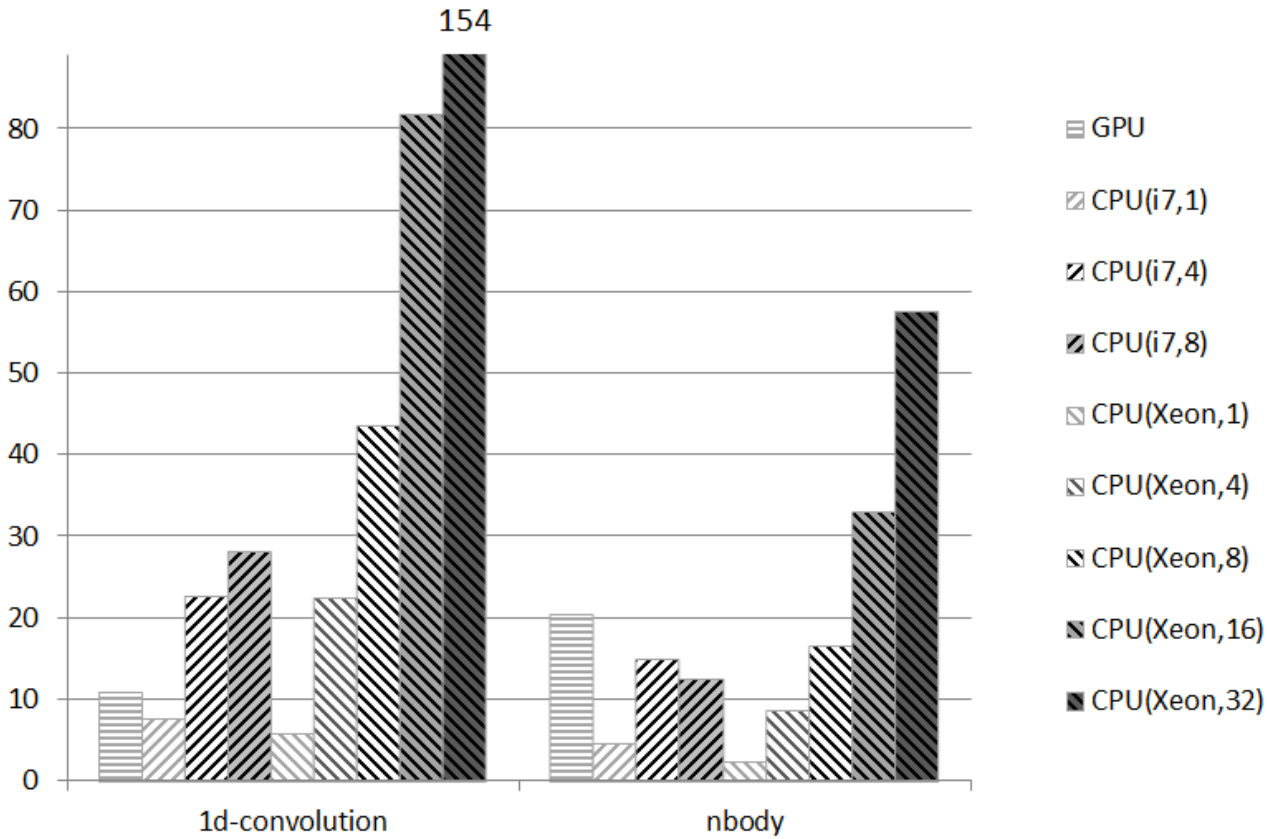
- The advance in hardware and OpenCL standard (e.g., SVM) gives new opportunities to explore alternatives.
- Native offload is a promising approach towards GPGPU.
- Optimizing for GPUs is challenging and fun.

Backup Slides

Haskell Repa Benchmark Programs

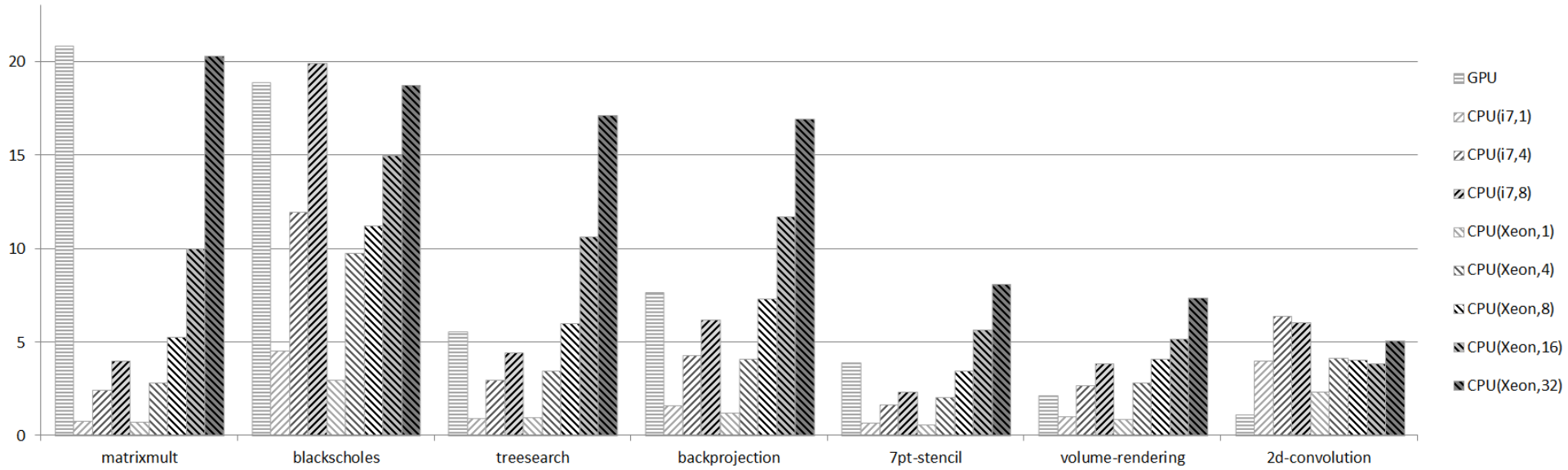
Name	Parameter	iteration	Description
1d-convolution	3M pixels	10	1D convolution with 8192-point stencil
2d-convolution	3200×4000 pixels	100	2D convolution with a 5x5 stencil
7pt-stencil	256×256×160 pixels	100	3D convolution with 7-point stencil
backprojection	256×256×256 pixels	100	2D to 3D image projection
blackscholes	10M options	100	Black Scholes algorithm for put and call options
matrix-mult	2K×2K matrix	1	Matrix multiplication
nbody	200K bodies	1	Nbody simulation
treesearch	16-level tree, 20M inputs	50	Binary tree search
volume-rendering	1M input rays	1000	Volumetric rendering

Benchmarking result: GPU vs CPU (2/9)



Kernel speedups relative to non-vectorized single-thread Core i7.
(bigger is better)

Benchmarking result: GPU vs CPU (7/9)



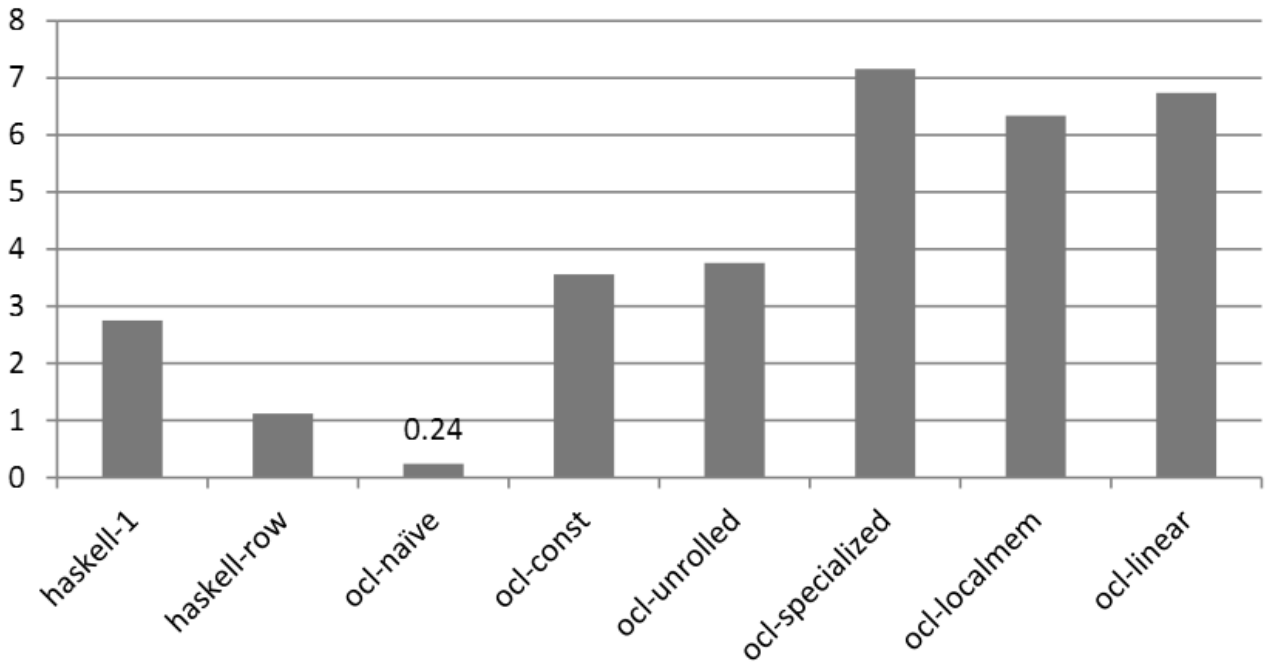
Kernel speedups relative to non-vectorized single-thread Core i7. (bigger is better)

Haskell vs OpenCL Performance (2D Convolution)

Benchmark	Description
haskell-1	Haskell program with a kernel that computes only one output pixel
haskell-row	Haskell program with a kernel that computes an entire output row
ocl-naive	native OpenCL that reads 5x5 stencil from an array
ocl-const	Similar to ocl-naive, specifies constant memory for stencil array
ocl-unrolled	Similar to naive-const, with stencil loop unrolled
ocl-specialized	Similar to ocl-unrolled, with stencil values specialized
ocl-localmem	Similar to ocl-specialized, uses a 20x20 local memory for blocking
ocl-linear	OpenCL ported from the generated kernel of haskell-1

OpenCL and Haskell benchmarks for 2D convolution

Haskell vs OpenCL (2D Convolution)



2D convolution kernel speedups relative to Core i7 (bigger is better)

- ocl-localmem is slower than ocl-specialized.
- ocl-linear is a direct port of haskell-1, yet more than 2X faster.
- haskell-row is optimized for CPU, but got worse on GPU.