Compiling for HSA accelerators with GCC

Martin Jambor

SUSE Labs

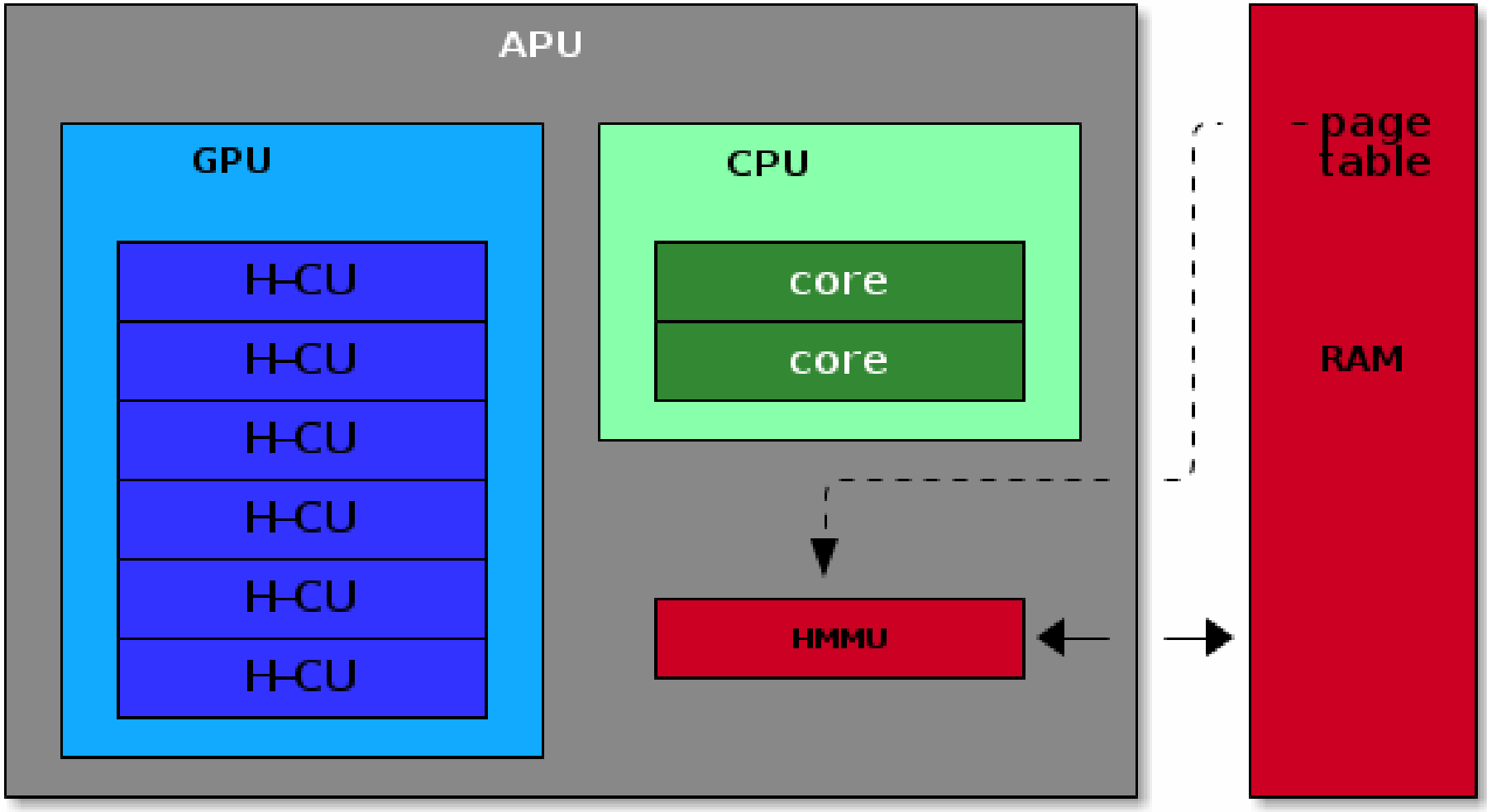
8th August 2015

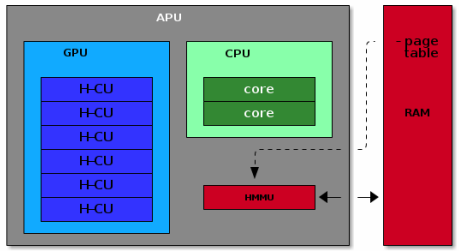
Outline

HSA branch: *svn://gcc.gnu.org/svn/gcc/branches/hsa*

Table of contents:

Heterogeneous System Architecture





HSA (extremely brief & imprecise overview for the purposes of this talk):

- Architecture developed by HSA Foundation (AMD, ARM, Qualcomm, Samsung, Texas instruments and many others. See www.hsafoundation.com.
- CPUs and GPUs on the same chip
- Sharing memory (cache coherent but with relaxed consistency)
- Unified virtual address space (devices can share data just by passing pointers)
- Dispatching through work queuing (also GPU→CPU and GPU→GPU)
- HSAIL...

HSA Intermediate Language (HSAIL)

```
prog kernel &__vector_copy_kernel(
        kernarg_u64 %a,
        kernarg_u64 %b)
{
        workitemabsid_u32 $s0, 0;
        cvt_s64_s32 $d0, $s0;
        shl_u64 $d0, $d0, 2;
        ld_kernarg_align(8)_width(all)_u64 $d1, [%b];
        add_u64 $d1, $d1, $d0;
        ld_kernarg_align(8)_width(all)_u64 $d2, [%a];
        add_u64 $d0, $d2, $d0;
        ld_global_u32 $s0, [$d0];
        st_global_u32 $s0, [$d1];
        ret;
};
```

HSA Intermediate Language (HSAIL)

```
prog kernel &__vector_copy_kernel(
        kernarg_u64 %a,
        kernarg_u64 %b)
{
        workitemabsid_u32 $s0, 0;
        cvt_s64_s32 $d0, $s0;
        shl_u64 $d0, $d0, 2;
        ld_kernarg_align(8)_width(all)_u64 $d1, [%b];
        add_u64 $d1, $d1, $d0;
        ld_kernarg_align(8)_width(all)_u64 $d2, [%a];
        add_u64 $d0, $d2, $d0;
        ld_global_u32 $s0, [$d0];
        st_global_u32 $s0, [$d1];
        ret;
};
```

Compilation target: HSAIL

- Intermediate language
- Finalizer needed to translate it to the real GPU ISA
  - Based on LLVM
  - We have heard from a person who woks on making a GCC-based finalizer
  - AFAIK, the finalizer is still not opens-source, but we have been assured it will be (everything else such as drivers or run-time is).
- Textual and BRIG representation
- Close to assembly
- Explicitly parallel

HSAIL is explicitly parallel

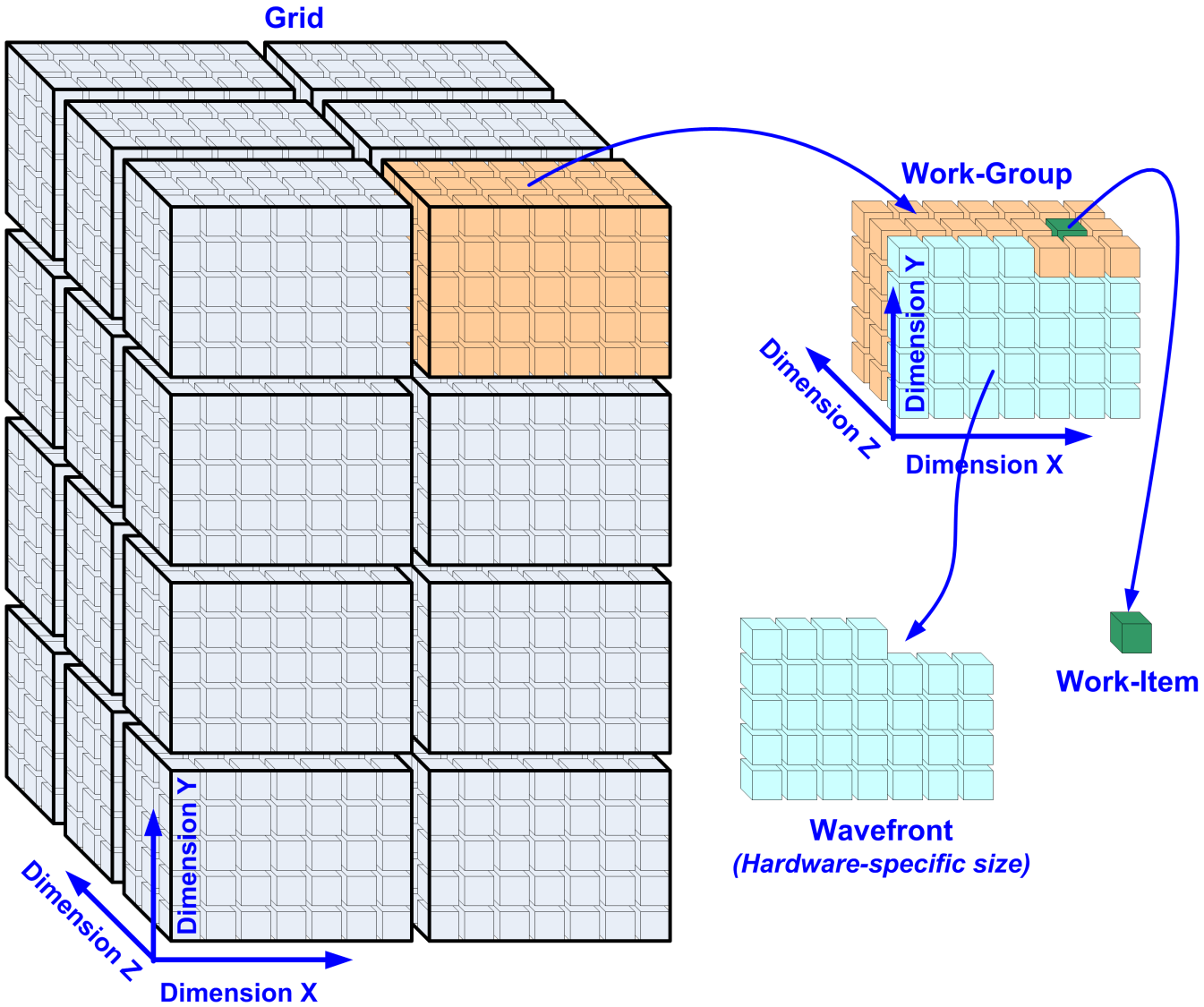


Image from www.hsafoundation.com

HSAIL is explicitly parallel

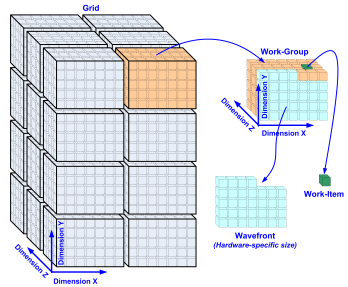
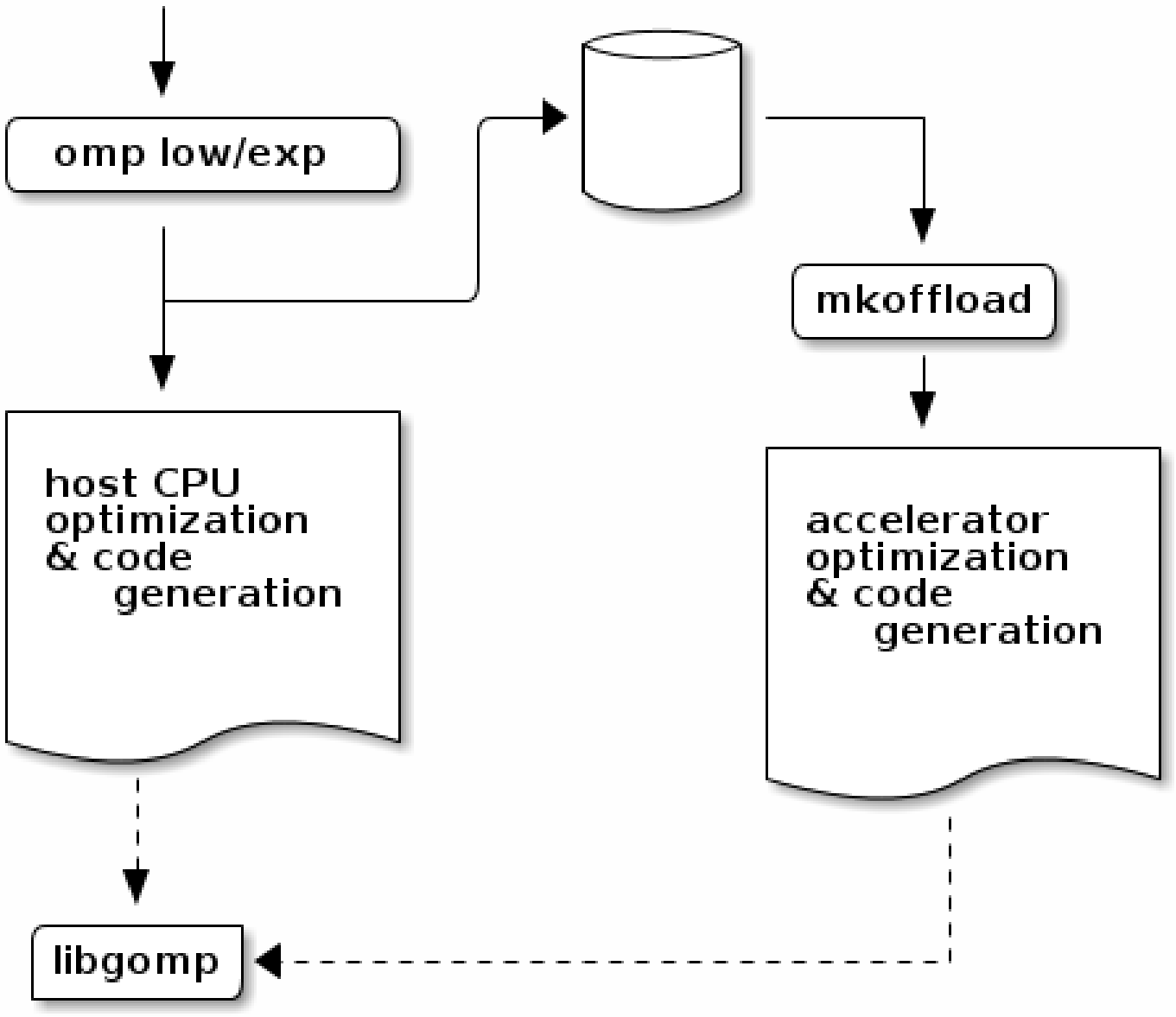
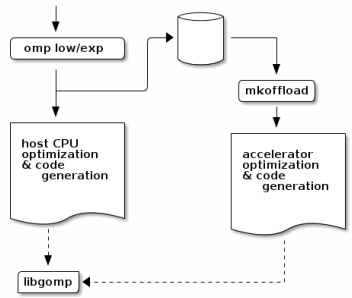


Image from www.hsafoundation.com

- Many **work-items**, ideally each roughly corresponding to one iteration of a loop
- Each has own register set and a private bit of memory
- Grouped in **work groups** which can synchronize and communicate through group-private memory
- Together they form a 3D grid (but we currently only use one dimension)

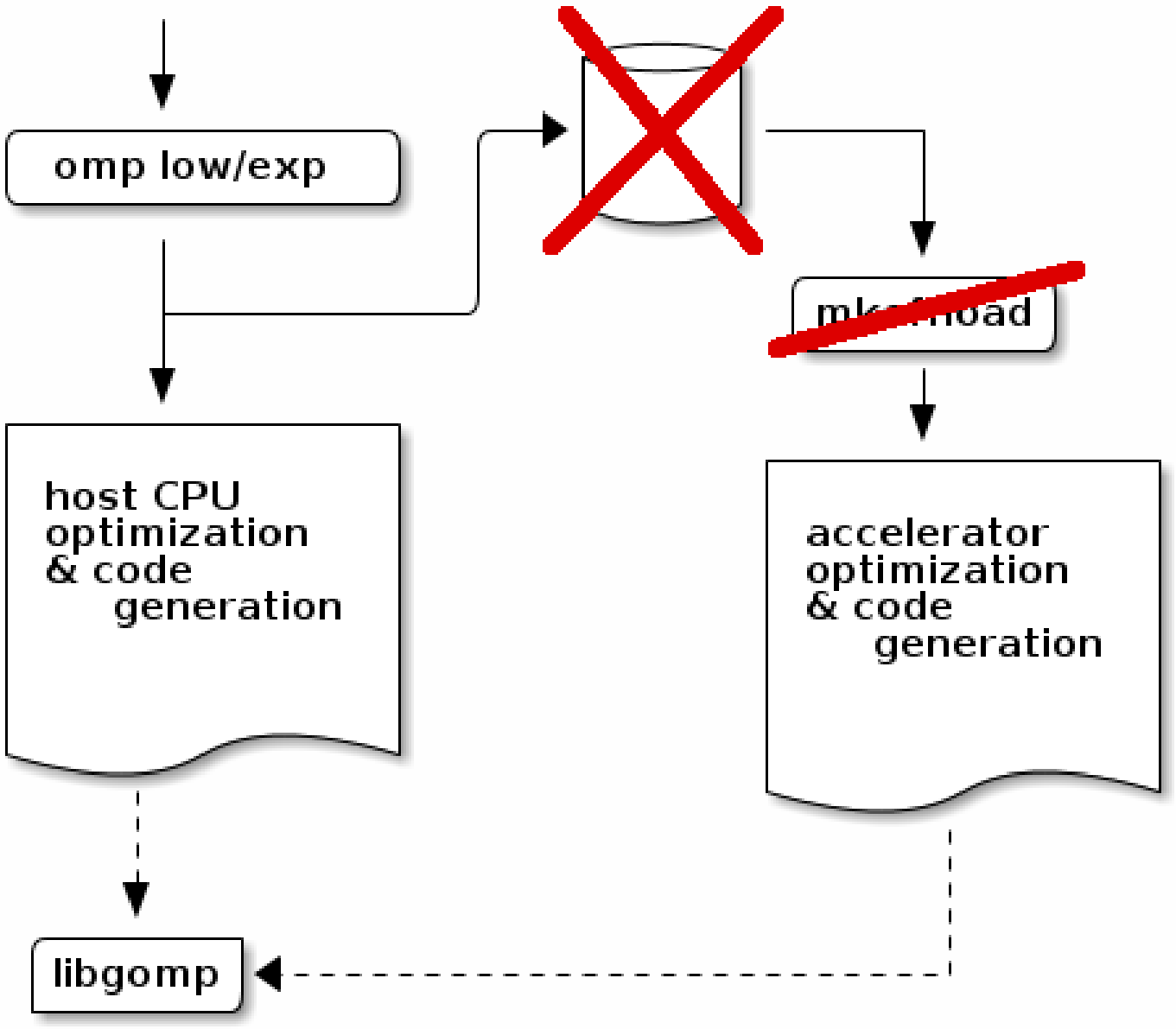
Acceleration via byte-code streaming (MIC, NvPTX)

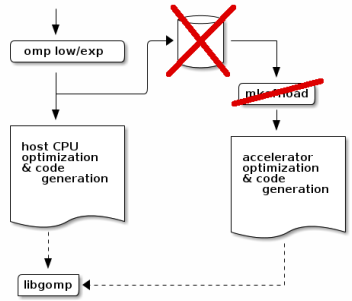




- Simplified scheme
- OMP lowering and OMP expansion use gimple statements corresponding to OpenMP and OpenACC statements to identify what code needs to be compiled also for accelerators.
- That code is then streamed out and a special utility mkoffload has it compiled by a different gcc back-end configured for a different target.
- This code is linked with the rest of the binary, registered with libgomp by compilation unit constructors and libgomp can then decide to run it.

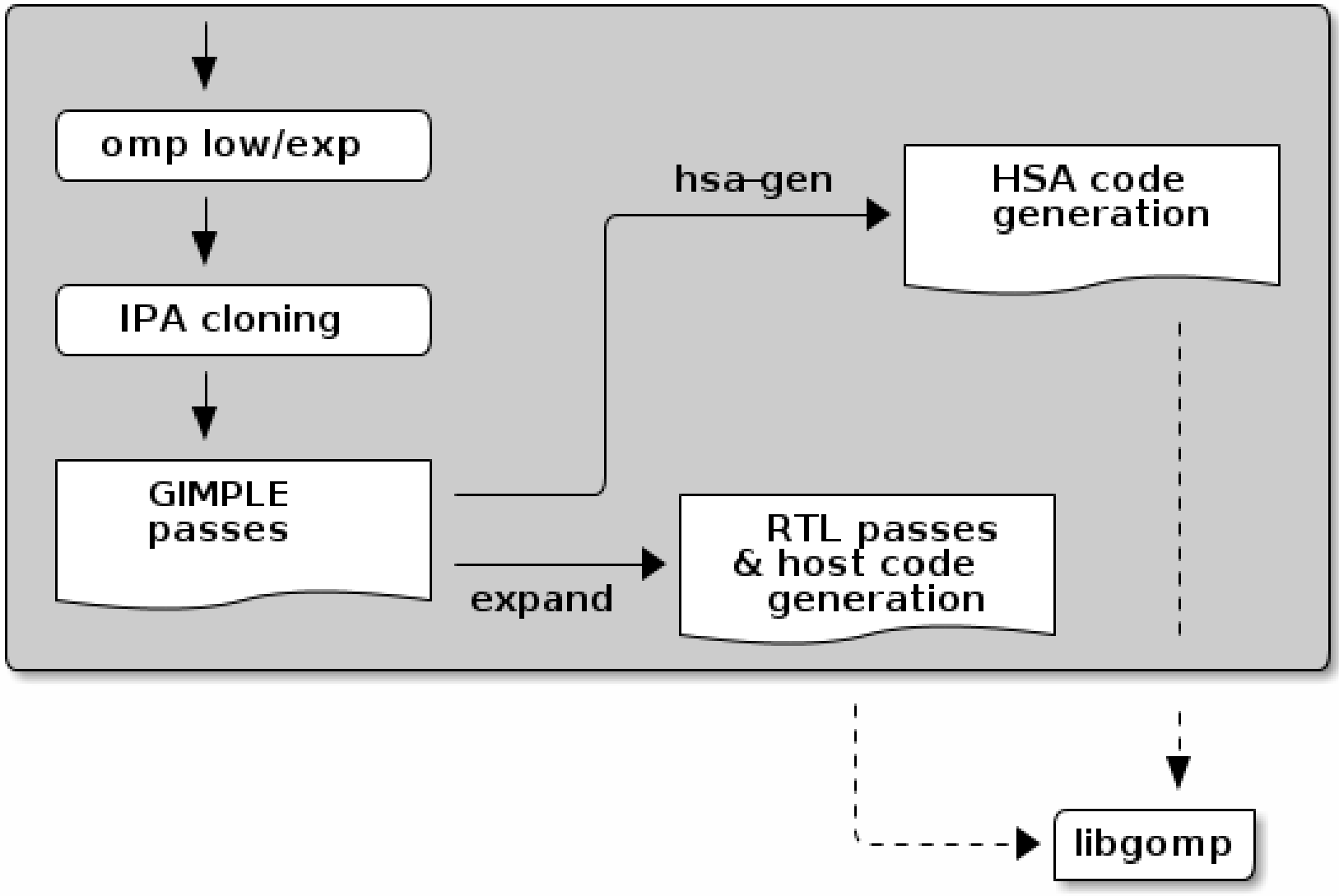
That's not how we do it

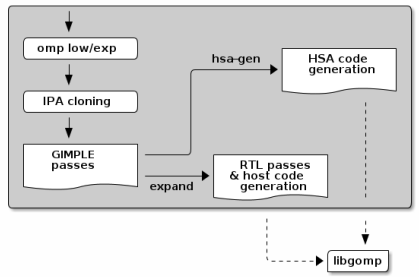




- We don't stream byte-code to disk.
- We don't have mkoffload either.
- We do compilation within one compiler (configured for the host).

HSAIL generation





- We introduce HSA generating pass just before expand
- Main advantage: no streaming
- Main disadvantage: gimple passes tuned for the target, not HSAIL. E.g. vectorizer needs to be switched off.
- Soon we want to have each function passing IPA or late gimple pass pipeline to be either for host o for HSA.
- Compilation unit constructor also registers generated BRIG modules with libgomp.
- It is also configured via the `--enable-offload` configure option.

HSA back-end

Currently three stages:

1. **hsa-gen.c**: Gimple $\rightarrow$ our internal HSAIL representation (which is in SSA form)
2. **hsa-regalloc.c**: Out-of-SSA and register allocation
3. **hsa-brig.c**: BRIG generation and output

Other components:

- **hsa.h**: Classes making up our internal HSAIL representation
- **hsa.c**: Common functionality
- **hsa-dump.c**: HSAIL dumping in textual form
- **hsa-brig-format.h**: HSA 1.0 BRIG structures

HSA back-end

Currently three stages:

1. **hsa-gen.c**: Gimple → our internal HSAIL representation (which is in SSA form)
2. **hsa-regalloc.c**: Out-of-SSA and register allocation
3. **hsa-brig.c**: BRIG generation and output

Other components:

- **hsa.h**: Classes making up our internal HSAIL representation
- **hsa.c**: Common functionality
- **hsa-dump.c**: HSAIL dumping in textual form
- **hsa-brig-format.h**: HSA 1.0 BRIG structures

- Our internal representation resembles HSAIL and BRIG specification
- We have to have our own register allocator because we do not use RTL stage at all but a simpler one seems sufficient.
- Our register allocator was written by Michael Matz
- We do not plan to grow a real optimization pipeline here.
- We rely on gimple passes
- Perhaps only value numbering to remove redundancies arising from address calculations or some very specific HSA transformations such as (possibly) pointer segment tracking.

Input

We target (primarily) OpenMP 4

...and that is the biggest headache.

We target (primarily) OpenMP 4

...and that is the biggest headache.

Three kinds of problems:

1. Things that perhaps can't even be reasonably implemented on HSA or a GPU in general, e.g. `#pragma omp critical` (critical section).
2. Libgomp usually necessary for OpenMP construct implementation but libgomp cannot be easily ported to HSA
   - It is based on mutexes
   - It uses indirect calls and function pointers extensively which are very slow and cumbersome

   $\Longrightarrow$ that a lot of things need to be implemented from scratch and often it is not clear if it is worth it.
3. The form to which we we expand OpenMP loops in particular is very inefficient for a GPU.

A simplest loop...

```
#pragma omp target
#pragma omp parallel firstprivate(n) private(i)
#pragma omp for
  for (i = 0; i < n; i++)
    a[i] = b[i] * b[i];
```

A simplest loop...

```
#pragma omp target
#pragma omp parallel firstprivate(n) private(i)
#pragma omp for
  for (i = 0; i < n; i++)
    a[i] = b[i] * b[i];
```

Lets have a look at how this simple parallel array multiplication is lowered and expanded by `omplower` and `ompexp`.

...is currently expanded to

```
n = .omp_data_i->n;
q = n / nthreads
tt = n % nthreads
if (threadid < tt) {
  tt = 0;
  q++;
}
s0 = q * threadid + tt
e0 = s0 + q

for (i = s0; i < e0; i++)
{
  a = .omp_data_i->a;
  b = .omp_data_i->b
  a[i] = b[i] * b[i];
}
```

Compiling for HSA accelerators with GCC

Input: OpenMP 4

...is currently expanded to

...is currently expanded to

```
n = .omp_data_i->n;
q = n / nthreads
tt = n % nthreads
if (threadid < tt) {
  tt = 0;
  q++;
}
s0 = q * threadid + tt
e0 = s0 + q

for (i = s0; i < e0; i++)
{
  a = .omp_data_i->a;
  b = .omp_data_i->b
  a[i] = b[i] * b[i];
}
```

- Each thread computes its loop bounds
- And then loops over its portion of the iteration space

...but the idea of programming HSA GPUs is different

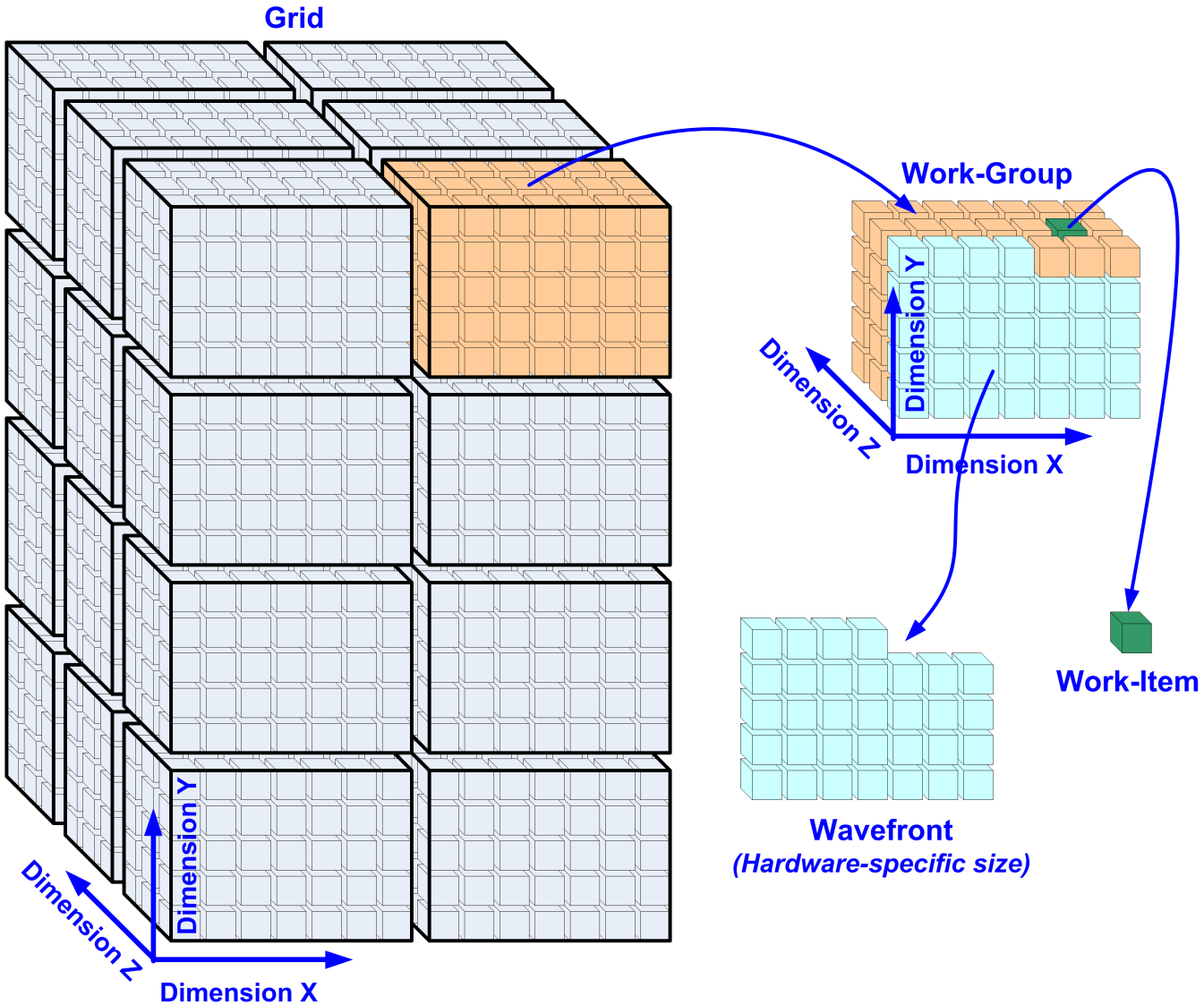


Image from www.hsafoundation.com

Compiling for HSA accelerators with GCC

Input: OpenMP 4

...but the idea of programming HSA GPUs is different

...but the idea of programming HSA GPUs is different

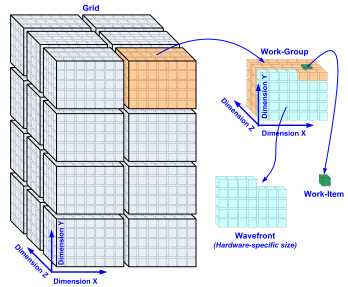


Image from www.hsafoundation.com

- That is contrary to the way HSA GPUs are meant to be programmed
- But GPGPUs hate control-flow
- So we have modified omp lowering and expansion of the loop to also create a special HSA version and to pass the iteration space to HSA run-time through libgomp.

Stream benchmark (1)

```
/* Copy: */
#pragma omp target parallel for private(j)
   for (j=0; j<STREAM_ARRAY_SIZE; j++)
      c[j] = a[j];

/* Scale: */
#pragma omp target parallel for private(j)
   for (j=0; j<STREAM_ARRAY_SIZE; j++)
      b[j] = scalar *c[j];

/* Add: */
#pragma omp target parallel for private(j)
   for (j=0; j<STREAM_ARRAY_SIZE; j++)
      c[j] = a[j]+b[j];

/* Triad: */
#pragma omp target parallel for private(j)
   for (j=0; j<STREAM_ARRAY_SIZE; j++)
      a[j] = b[j]+scalar*c[j];
```

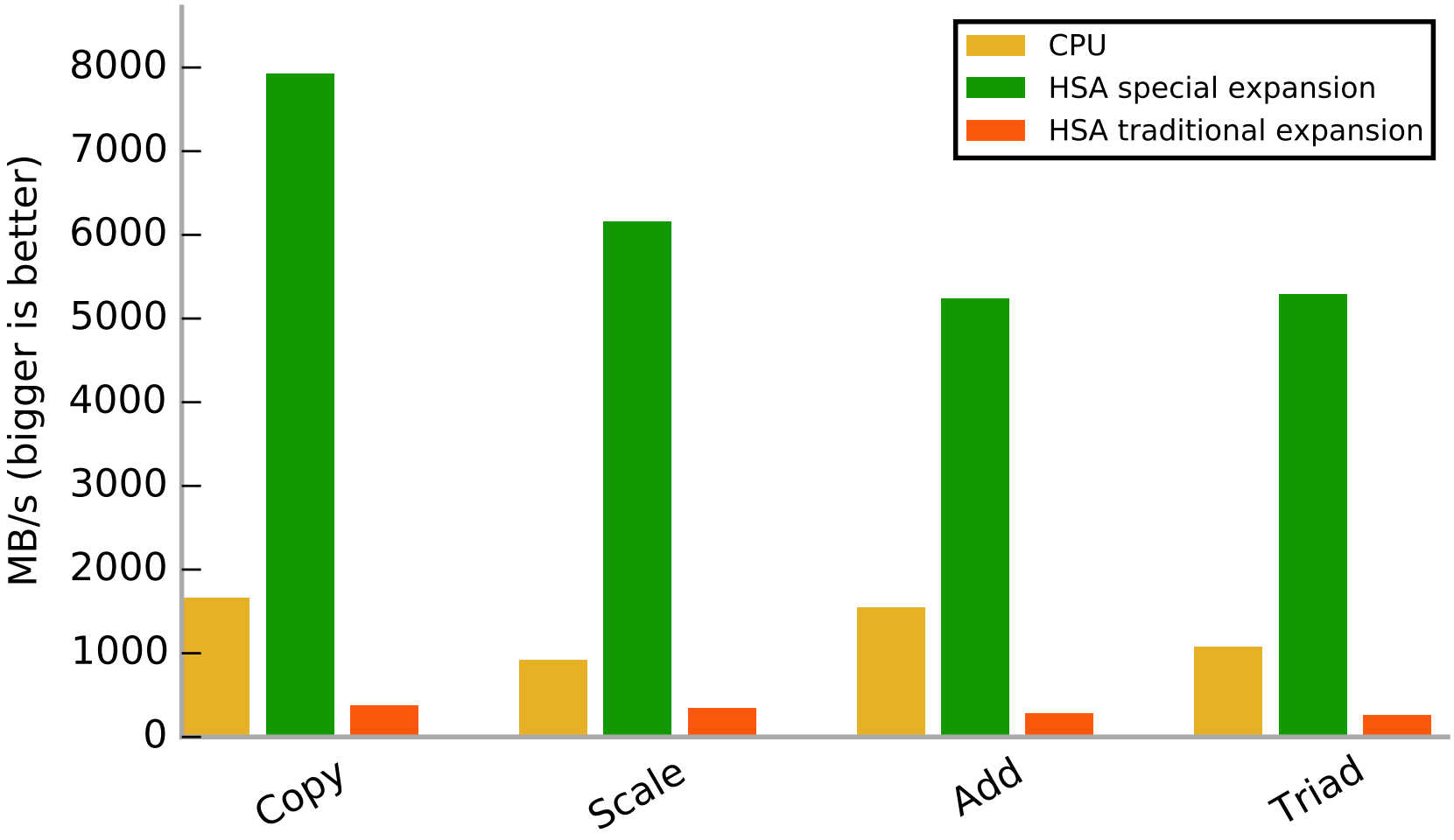
Stream benchmark (1)

```
/* Copy:*/
#pragma omp target parallel for private(j)
   for (j=0; j<STREAM_ARRAY_SIZE; j++)
      c[j] = a[j];

/* Scale: */
#pragma omp target parallel for private(j)
   for (j=0; j<STREAM_ARRAY_SIZE; j++)
      b[j] = scalar *c[j];

/* Add: */
#pragma omp target parallel for private(j)
   for (j=0; j<STREAM_ARRAY_SIZE; j++)
      c[j] = a[j]+b[j];

/* Triad: */
#pragma omp target parallel for private(j)
   for (j=0; j<STREAM_ARRAY_SIZE; j++)
      a[j] = b[j]+scalar*c[j];
```

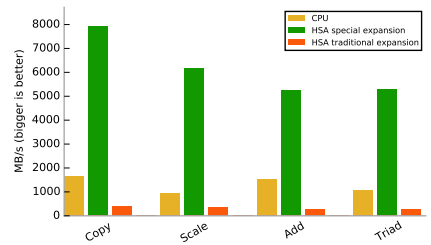
- Stream benchmark measures the throughput of these four very simple loops.

Stream benchmark (2)

Stream benchmark results for 64kB arrays (16k of floats) on a Carrizo APU:



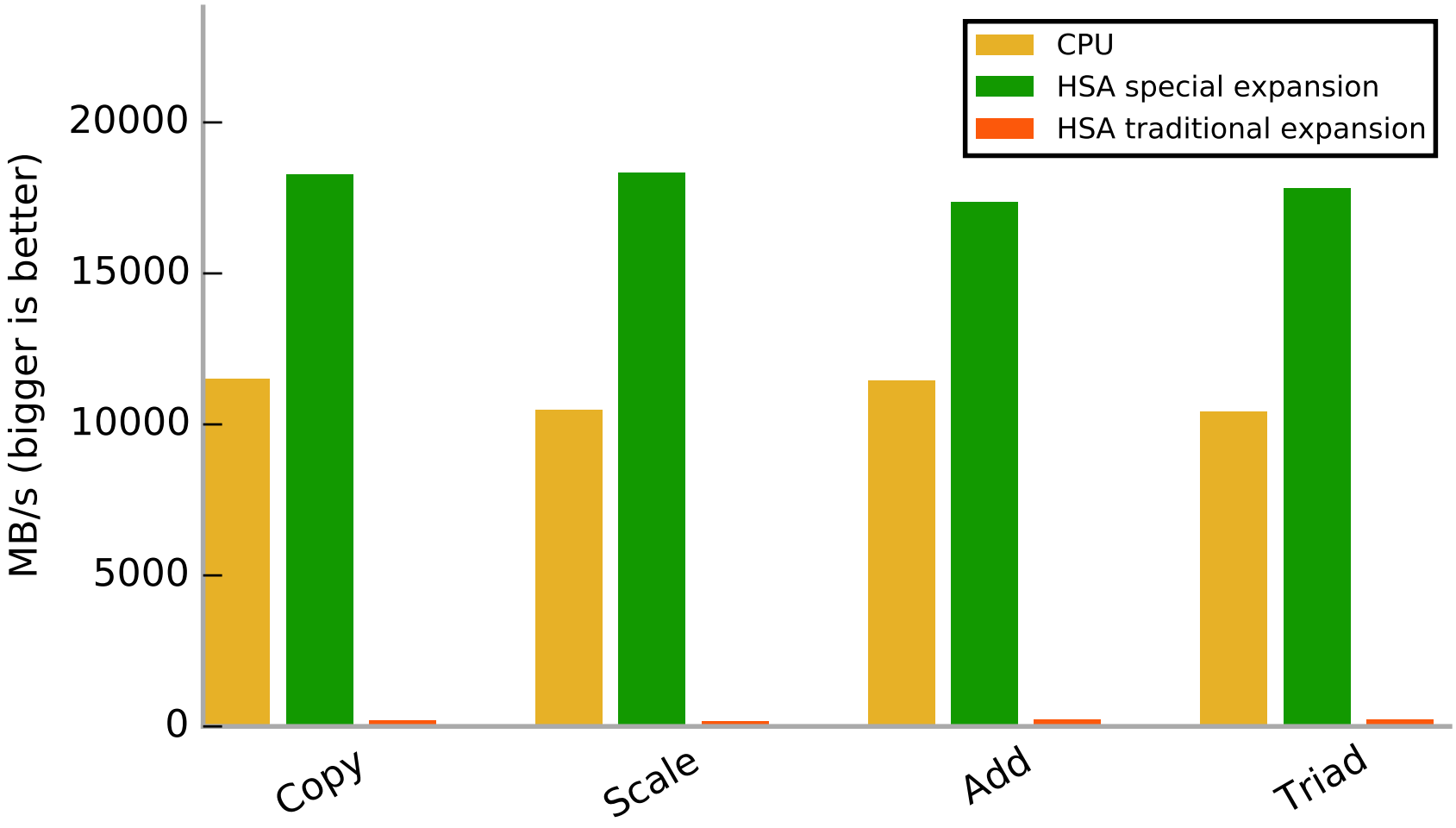
Stream benchmark (2)



- CPU speed to provide a reference. The particular Carrizo I used is not a very quick processor (still my very nice one year old notebook performs worse on the benchmark).
- If we compile the loops specially for GPUs, we get a nice speedup.
- If we do not, the performance is horrible.
- It is even slightly worse if dynamic parallelism as to be used to implement the parallel construct.

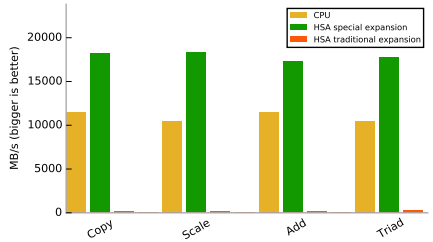
Stream benchmark (3)

Stream benchmark results for 128MB arrays (32M of floats) on a Carrizo APU:



Stream benchmark (3)

Stream benchmark results for 128MB arrays (32M of floats) on a Carrizo APU:



- The stream benchmark is supposed to be run on big arrays, so this data point is probably more descriptive.
- **The bottom line is that now you hopefully agree that the loop kernels need to be expanded differently.** But that is going to be ugly.

OMP lowering

```
...data packaging...
#pragma omp target map(...)
  {
    ...declarations, data re-packaging...
    #pragma omp parallel private(j) shared(c) shared(a)
      {
        ...declarations, data un-packaging...
        #pragma omp for nowait
        for (j = 0; j <= 33554431; j = j + 1)
          {
             ...loop body...
             #pragma omp continue
             #pragma omp return
          }
        #pragma omp return
      }

    #pragma omp return
  }
```

OMP lowering

```
...data packaging...
#pragma omp target map(...)
  {
    ...declarations, data re-packaging...
    #pragma omp parallel private(j) shared(c) shared(a)
      {
        ...declarations, data un-packaging...
        #pragma omp for nowait
        for (j = 0; j <= 33554431; j = j + 1)
          {
            ...loop body...
            #pragma omp continue
            #pragma omp return
          }
        #pragma omp return
      }

    #pragma omp return
  }
```

- OpenMP handling in middle-end divided into lowering and expansion.
- Lowering does variable re-mapping and creation of sender-receiver parameters.
- Expansion does actual outlining into separate function and final expansion of special omp gimple statements.
- CFG is built in between the two.
- I have tried creating a special version of the loop for HSA only in omp-expand but I ended up with a most horrible hack I have ever written (all because of undoing various mappings done by omp-lower).
- We need to modify lowering to avoid variable re-mapping for the parallel construct and expansion for outlining.

The current solution/hack(?)

```
...data packaging...
#pragma omp target map(...)
  {
    ...declarations, data re-packaging...
    #pragma omp parallel private(j) shared(c) shared(a)
      {
        ...declarations, data un-packaging...
        #pragma omp for nowait
        for (j = 0; j <= 33554431; j = j + 1)
          {
             ...loop body...
             #pragma omp continue
             #pragma omp return
          }
        #pragma omp return
      }
    #pragma omp kernel_for_body { ...unpackaging, loop body... }
    #pragma omp return
  }
```

The current solution/hack(?)

```
...data packaging...
#pragma omp target map(...)
  {
    ...declarations, data re-packaging...
    #pragma omp parallel private(j) shared(c) shared(a)
      {
        ...declarations, data un-packaging...
        #pragma omp for nowait
        for (j = 0; j <= 33554431; j = j + 1)
          {
            ...loop body...
            #pragma omp continue
            #pragma omp return
          }
        #pragma omp return
      }
    #pragma omp kernel_for_body { ...unpackaging, loop body... }
    #pragma omp return
  }
```

- So we create a special copy of the loop within target (and outside parallel), have only the target mappings performed on it.
- This has many potential downsides, including:
  - So far I only do this if there is perfect nesting, two consecutive parallel or for constructs might pose insurmountable obstacle.
  - Parallel clauses get basically ignored.
  - Collapse, reductions etc. may force a re-think once again.
  - We might declare all GPUs "`omp simd` only" targets?

The bottom line is that we will need these kinds of changes, even though they do not fit the current scheme of things very well.

Summary

- OpenMP construct expansion will have to be different.
- A lot of things that are done in libgomp now might need to be handled by the compiler (to eliminate control flow, calls and so forth).
- A lot of functionality will not be implemented on GPGPU soon and some perhaps never (critical sections).
- What to do if only a target construct cannot be compiled for a particular accelerator? Can we detect it well enough to handle it gracefully?
- There are benefits to be gained for things HSA can do.
- So even given the current problems we plan to merge the branch to gcc 6.

...any questions?

Summary

- OpenMP construct expansion will have to be different.
- A lot of things that are done in libgomp now might need to be handled by the compiler (to eliminate control flow, calls and so forth).
- A lot of functionality will not be implemented on GPGPU soon and some perhaps never (critical sections).
- What to do if only a target construct cannot be compiled for a particular accelerator? Can we detect it well enough to handle it gracefully?
- There are benefits to be gained for things HSA can do.
- So even given the current problems we plan to merge the branch to gcc 6.

...any questions?

- OpenMP construct expansion will have to be different.
- A lot of things that are done in libgomp now might need to be handled by the compiler (to eliminate control flow, calls and so forth).
- A lot of functionality will not be implemented on GPGPU soon and some perhaps never (critical sections).
- What to do if only a target construct cannot be compiled for a particular accelerator? Can we detect it well enough to handle it gracefully?
- There are benefits to be gained for things HSA can do.
- So even given the current problems we plan to merge the branch to gcc 6.