Translator

This document highlights some of the implementation decisions when creating the translator class.

Parser and tree

The parser classifies statements via the `gpufort.util.parsing` package, and then uses either parsers from that pacakge or a `pyparsing` grammar to translate statements into a tree representation. While `pyparsing` subtrees are created via `<pyparsing_grammar_object>.parse_string(<statement_str>)`, tokens identified via `util.parsing` are translated explicitly into tree nodes via appropriate constructors. (We noticed that a `pyparsing`-based translation can be rather slow. Therefore, we began replace more and more pyparsing parse expressions by dedicated parse routines from the `gpufort.util.parsing` package.) While the OpenACC to OpenMP translation relies on a `pyparsing` grammar for all directives, the OpenACC to HIP conversion relies only on the directives associated with compute constructs and not with directives that are translated to OpenACC runtime calls.

All translator tree nodes are defined in the subpackage `translator.tree`. They all have the prefix the `TT` ("translator tree"); as opposed to the `ST` prefix, which indicates a scanner tree node).

Arithmetic and logical expressions, complex lvalues and rvalues

Complex lvalue and rvalue expressions such as a Fortran derived type member access are implemented via `pyparsing.Forward`, which can be used to forward declare certain expressions and, therefore, to realize a parser for recursive expressions:

Example 1 (`Forward`, recursive expressions):

```
from pyparsing import *
derived_type = Forward()
identifier = pyparsing_common.identifier
derived_type <<= identifier + Literal("%") + (identifier|derived_type)
```

The pyparsing grammar constructed in **Ex. 1** is able to parse arbitrary recursive expressions such as

```
a%b1
a%b1%c2_3%d
```

In order to parse arithmetic and logical expressions as they appear in assignments and conditional expressions as they, e.g., appear in `IF`, `ELSEIF` statements, we rely on another `pyparsing` shortcut, `infixNotation`, which take an rvalue-expresion and a list of operators and their respective number of operands and their associativity. The position of the operator in the list of operators indicates its preference.

Example 2 (`infixNotation`):

```
import pyparsing as pyp

number     = pyp.pyparsing_common.number.copy().setParseAction(lambda x: str(x))
identifier = pyp.pyparsing_common.identifier
terminal   = identifier | number

expr = pyp.infixNotation(terminal, [
    (pyp.Regex('[\-+]'), 1, pyp.opAssoc.RIGHT),
    (pyp.Regex('[*/+\-]'), 2, pyp.opAssoc.LEFT),
]) + pyp.stringEnd()

print(expr.parseString("-5 + 3"))
print(expr.parseString("-5"))
print(expr.parseString("-( a + (b-1))"))
```

Loop and compute construct directives

Fortran `DO` (and `DO_CONCURRENT`) loops are the main targets for directive-based offloading moels, which annotate the former with information on how to map that particular loop (or loopnest) to a device's compute units. In the GPUFORT translator tree, `DO`-loops have a loop annotation for storing subtrees associated with OpenACC (`acc loop`, `acc parallel loop`, `acc kernels loop`) and CUDA Fortran directives (`!$cuf kernel do`). These loop annotations implement the interface `translator.tree.directives.ILoopAnnotation` so that information can be obtained from them in an unified way. A similar interface, `translator.tree.directives.IComputeConstruct`, exists to represent the different OpenACC compute constructs (`acc kernels`, `acc kernels loop`, `acc parallel loop`, `acc parallel`, `acc serial`) and the CUDA Fortran construct `!$cuf kernel do` in a unified way.

HIP C++ code geneneration from OpenACC compute constructs

OpenACC parallelism levels

OpenACC considers three levels of parallelism, *gang*, *worker*, and *vector*. Gangs run independently from each other and in arbitrary order. Since we consider NVIDIA and AMD GPU devices, we can map gangs to CUDA/HIP thread blocks, and we can map workers to CUDA *warps*/HIP *wavefronts*.

While we refer to the OpenACC specification for more details, we explain a number of different example OpenACC compute constructs to show the implications of certain directives and attached parallelism clauses.

Gang-redundant, worker-single, vector-single mode

If no `acc loop` directive have been encountered so far or the associated loops have been marked as non-parallelizable, a compute construct's instructions are run in gang-redundant (GR), worker-single (WS), vector-single (VS) mode. In the CUDA/HIP world, this implies that a number of thread blocks are scheduled that all execute the same instructions with a single worker and a single vector lane enabled per worker, i.e. in CUDA words, only a single thread (and thus single warp) is required to execute the statements.

GR-WS-VS parallelism mode can be used to perform non-parallelizable operations on the device before running other work in parallel on the availabel workers and their vector lanes.

Note that if more workers or wider vectors are required by the compute construct later on, which is typically the case, this means that the additional resources must be masked out.

Let's have a look at an example.

Example 3 (Non-parallelizable loop):

```
! initialization of x and m
! ...
!$acc parallel
!!$acc parallel num_gangs(4)
do i = 1,m
 if ( i == 0 ) then
   x(i) = 1
 else
   x(i) = 2*x(i-1)
 endif
end do
!$acc end parallel
```

where `x` may be an array of floats here.

The number of gangs depends on other work that appears later on in the compute construct, or defaults to 1. However, the outcome of this analysis can be overwritten via the `num_gangs` clause as demonstrated in the commented out directive–the one prefixed by `!!` instead of `!`).

The above compute construct can be mapped to the following HIP C++ code, which consists of a kernel definition (indicated by the `__global__` attribute) plus a kernel launch that is placed somewhere into the host code.

```
// includes
// ...

__global__ void mykernel(float* x,int m) {
  int gang_tid = threadIdx.x;
  if ( gang_tid == 0 ) { // masks out additional workers and vector lanes if available
    for (int i = 1; i <= m; i++) {
      if ( i == 0 ) {
        x[i] = 1;
      } else {
        x[i] = 2*x[i-1];
      }
    }
  }
}

int main(int argc, char** argv) {
  // initialization of x, m, NUM_GANGS, NUM_WORKERS, and VECTOR_LENGTH
  // ...
  int NUM_THREADS = NUM_WORKERS*VECTOR_LENGTH
  hipLaunchKernelGGL((mykernel),dim3(NUM_GANGS),dim(NUM_THREADS),x,m)
  //hipLaunchKernelGGL((mykernel),dim3(4),dim(NUM_THREADS),x,m);
}
```

(The commented kernel launch statement fixes the number of thread blocks to 4 as has been done in the OpenACC compute construct via the `num_gangs(4)` clause.)

Gang-partitioned, worker-single, vector-single mode

If the OpenACC compilers encounters a parallelizable loop (gang-loop iterates must be completetly independent of each other) marked with the `gang` clause, it switches to gang-parallel (GP), worker-single (WS), vector-single (VS) mode. In GP-WS-VS mode, the annotated loop is now distributed among all available gangs, but still only a single worker and a single vector lane, i.e. one CUDA / HIP thread are used.

Again, the number of gangs can be derived automatically or forced manually, as shown in the two examples below:

Example 4:

```
!$acc parallel
!$acc loop gang
!!$acc loop gang(4)
do i = 1,m
 x(i) = 1
end do
!$acc end parallel
```

The `acc parallel` and `acc loop` directives can be combined and the gang usage can be limited via the argument of the `gang` clause.

Example 5:

```
!$acc parallel loop gang(4)
do i = 1,m
 x(i) = 1
end do
!$acc end parallel
```

Again, additional available workers and vector lanes must be masked out in an equivalent CUDA/HIP C++ implementation, like the one shown below:

```
// includes
// ...

__host__ __device__ int div_round_up(int x, int y) {
  return x / y + (x % y > 0);
}

__global__ void mykernel(float* x, int m) {
  int max_num_gangs = gridDim.x;
  int gang_id = blockIdx.x;
  int gang_tid  = threadIdx.x;
  // loop specific
  int num_gangs = max_num_gangs;
  // int num_gangs = 4; // if gang(4) is specified
  int gang_tile_size = div_round_up(m,num_gangs);
  if ( gang_tid == 0 ) { // masks out additional workers
                         // and vector lanes if available
    for ( int i = 1+gang_id*gang_tile_size;
              i <= (gang_id+1)*gang_tile_size; i++) {
      // total number of i iterates across all active gangs:
      // num_gangs*gang_tile_size >= m, due to rounding up
      if ( i <= m) {
        x[i] = 1;
      }
    }
  }
}

int main(int argc, char** argv) {
  // initialization of x, m, NUM_WORKERS, and VECTOR_LENGTH
  // ...
  int NUM_THREADS = NUM_WORKERS*VECTOR_LENGTH
  hipLaunchKernelGGL((mykernel),dim3(m),dim(NUM_THREADS),x,m)
  //hipLaunchKernelGGL((mykernel),dim3(4),dim(NUM_THREADS),x,m);
}
```

The `__global__` attribute flags the function `mykernel` as the device code entry point, i.e. as so-called “GPU kernel”. The structs gridDim, blockIdx, and threadIdx are built-in variables that are only available in the body of a CUDA/HIP kernel. These built-ins cannot neither be used in host code not in functions with __device__ attribute, i.e. device subroutines that can be called only from within a GPU kernel or other device subroutines.

Note that the construct

```
for ( int i = 1+gang_id*gang_tile_size;
          i <= (gang_id+1)*gang_tile_size; i++) {
  // total number of i iterates across all active gangs:
  // num_gangs*gang_tile_size >= m, due to rounding up
  if ( i <= m) {
    // ...
  }
}
```

masks out all gangs with gang_id > num_gangs as the loop is tiled with respect to num_gangs,

```
int gang_tile_size = div_round_up(m,num_gangs);
```

Worker-partioned, vector-single mode

In this section, we assume gang-partitioned mode and refer to the previous sections regarding the difference between gang-partitioned mode and gang-redundant mode.

If an OpenACC compiler, encounters a loop or workshare that is annotated with the worker clause, it switches from worker-single to worker-partitioned mode, i.e. more than a single CUDA warp / HIP wavefront may be used to process the loop. Again, the number of workers is determined automatically by the compiler or can be prescribed. The latter can be done either globally for all worker-partitioned loops within the compute construct by adding a num_workers(<int-arg>) clause to the directive that opens the compute construct or locally by adding the worker(<int-arg>) clause to the loop directive. If both are specified, the locally chosen number of workers is constrained to not be larger than the global value.

The amount of workers is constrained by the device’s hardware. Only a certain amount of workers can be allocated per gang. On current-gen CUDA devices, the limit is $1024/32 = 32$ workers per gang while it is \$$1024/64 = 16$‘ for current-gen AMD GPUs as both vendor’s device support up to 1024 threads per thread block and have warp/wavefront size of 32 and 64, respectively.

Example 6:

Fortran:

```
!$acc parallel
```

```
!$acc loop gang(4)
do j = 1,n
 !$acc loop worker(2)
 do i = 1,m
   x(i,j) = 1 ! column-major
 end do
end do
!$acc end parallel
```

Equivalent HIP C++ implementation:

```
// includes
// ...

__host__ __device__ int div_round_up(int x, int y) {
  return x / y + (x % y > 0);
}

__global__ void mykernel(float* x, int m, int n) {
  int max_num_gangs = gridDim.x;
  int gang_id = blockIdx.x;
  int gang_tid  = threadIdx.x;
  //
  int max_num_workers = div_round_up(blockDim.x / warpSize);
  int worker_id = gang_tid / warpSize;
  int worker_tid = gang_tid % warpSize;

  if ( worker_tid == 0 ) { // masks out all other threads/lanes of a worker
    // loop specific
    int num_gangs = max_num_gangs;
    // int num_gangs = 4; // if gang(4) is specified
    int num_workers = max_num_workers;
    // int num_workers = 2; // if worker(2) is specified
    int gang_tile_size = div_round_up(n,num_gangs);
    int worker_tile_size = div_round_up(m,num_workers);
    //
    for ( int j = 1+gang_id*gang_tile_size;
              j <= (gang_id+1)*gang_tile_size; j++) {
      // total number of j iterates across all gangs:
      // num_gangs*gang_tile_size >= n, due to rounding up
      if ( j <= n ) {
        for ( int i = 1+worker_id*worker_tile_size;
                  i <= (worker_id+1)*worker_tile_size; i++ ) {
          // total number of i iterates across all workers:
          // num_workers*worker_tile_size >= m, due to rounding up
          if ( i <= m) {
            x[i+n*j] = 1;
          }
        }
      }
    }
  }
}

int main(int argc, char** argv) {
  // initialization of x, m, n, NUM_WORKERS,
  // and MAX_VECTOR_LENGTH (32 or 64 depending on arch)
  // ...
  int NUM_THREADS = 2*MAX_VECTOR_LENGTH;
  hipLaunchKernelGGL((mykernel),dim3(4),dim(NUM_THREADS),x,m,n);
}
```

The `warpSize` variable in the HIP C++ snippet is another HIP built-in. Its value depends on the target architecture.

> TODO below must be adjusted so that local `num_gangs` != `max_num_gangs`
>
> TODO below must be adjusted so that local `num_workers` != `max_num_workers`
>
> TODO below must be adjusted so that local `vector_length` != `max_vector_length`

Vector-partitioned mode

In this section, we assume gang-partitioned and worker-partitioned mode. We refer to the previous sections regarding the difference between gang-partitioned mode and gang-redundant mode and between worker-single and worker-partitioned mode.

If an OpenACC compiler encounters a loop or workshare that is annotated with the `vector` clause, it switches from vector-single to vector-partitioned mode, i.e. more than a single CUDA thread/HIP SIMD lane may be used to process the loop. As stated before, the number of vector lanes per worker generally depends on the GPU architecture. In HIP C++, it is available via the `warpSize` builtin variable.

Example 7:

Fortran:

```
!$acc parallel
!$acc loop gang(4)
do k = 1,p
  !$acc loop worker(2)
  do j = 1,n
```

```
   !$acc loop vector(8)
   do i = 1,m
     x(i,j,k) = 1 ! column-major
   end do
  end do
end do
!$acc end parallel
```

Equivalent HIP C++ implementation:

```
// includes
// ...

__host__ __device__ int div_round_up(int x, int y) {
  return x / y + (x % y > 0);
}

__global__ void mykernel(float* x, int m, int n, int k) {
  // generic preamble
  int max_num_gangs = gridDim.x;
  int gang_id = blockIdx.x;
  int gang_tid = threadIdx.x;
  //
  int max_num_workers = div_round_up(blockDim.x / warpSize);
  int worker_id = threadIdx.x / warpSize;
  //
  int max_vector_length = warpSize;
  int vector_lane_id = threadIdx.x % warpSize; // renamed from 'worker_tid'
  //
  {
    // loop specific
    // int num_gangs = max_num_gangs; // if gang is specified
    // int num_workers = max_num_workers; // if worker is specified
    // int vector_length = max_vector_length; // if vector is specified
    int num_gangs = 4; // gang(4) is specified
    int num_workers = 2; // worker(2) is specified
    int vector_length = 8; // vector(8) is specified
    //
    int gang_tile_size = div_round_up(p,num_gangs);
    int worker_tile_size = div_round_up(n,num_workers);
    int vector_tile_size = div_round_up(m,vector_length);
    //
    if ( vector_lane_id < vector_length ) { // masks out all other threads/lanes
                                        // of a worker
      for ( int k = 1+gang_id*gang_tile_size;
                k <= (gang_id+1)*gang_tile_size; k++) {
        if ( k <= p ) {
          for ( int j = 1+worker_id*worker_tile_size;
                    j <= (worker_id+1)*worker_tile_size; j++ ) {
            if ( j <= n) {
              for ( int i = vector_lane_id; i <= m; i+=vector_tile_size ) {
                x[i+m*j+m*n*k] = 1;
              }
            }
          }
        }
      }
    }
  }
}

int main(int argc, char** argv) {
  // initialization of x, m, n, NUM_WORKERS,
  // and MAX_VECTOR_LENGTH (32 or 64 depending on arch)
  // ...
  int NUM_THREADS = 2*MAX_VECTOR_LENGTH;
  hipLaunchKernelGGL((mykernel),dim3(4),dim(NUM_THREADS),x,m,n);
}
```

A HIP or CUDA kernel expresses the work that a single HIP SIMD lane or CUDA thread performs. The memory controller of the compute units of AMD and NVIDIA GPUs can group the memory loads and writes of multiple such entities into larger requests if their collective data access pattern allows so. Therefore, we have used the `vector_tile_size` as increment in the above loop and not `1` as for the gangs and workers. This ensures thiat `vector_lane_id l < vector_length` writes to datum `x[l+...]` while `vector_lane_id l+1 < vector_length)` writes to datum `x[(l+1)+...]`, i.e. we have contiguous memory access across the SIMD lanes of a worker.

Interim conclusions 1

At this stage, we conclude the following:

- We can use a generic preamble in HIP C++ kernels to query the maximum number of gangs, workers, and vector lanes. as well as the ids of the current gang, worker, and vector lane.

```
// generic preamble
int max_num_gangs = gridDim.x;
int gang_id = blockIdx.x;
int gang_tid = threadIdx.x;
//
int max_vector_length = warpSize;
int max_num_workers = div_round_up(blockDim.x / max_vector_length);
```

```
//
int worker_id = threadIdx.x / max_vector_length;
int vector_lane_id = threadIdx.x % max_vector_length;
//
```

Note that this preamble will look different if a `tile` clause is specified as more grid and block dimensions are used.

- The `vector length` must be provided explicitly either by hardcoding it into the kernel or by passing it as kernel argument
- The checks `if ( vector_lane_id == 0)` associated with vector-single mode and `if (vector_lane_id < vector_length )` associated with vector-partitioned mode can be combined to the latter expression if`vector_length` is set to `1` in the former case.

Loops with both gang and worker parallelism

In this section, we investigate loops with both `gang` and `worker` clause. Depending on what resource, gang or worker, the compiler can choose freely, different HIP C++ eqivalents are possible.

Example 8 (Gang-worker parallelism):

Fortran:

```
!$acc parallel
!$acc loop gang worker
! other variants
!!$acc loop gang worker(2)
!!$acc loop gang(4) worker
!!$acc loop gang(4) worker(2)
do i = 1,m
  x(i) = 1
end do
!$acc end parallel
```

A possible HIP C++ implementation:

```
// includes and definitions
// ...

__global__ void mykernel(float* x, int m) {
  // generic preamble
  int max_num_gangs = gridDim.x;
  int gang_id = blockIdx.x;
  int gang_tid = threadIdx.x;
  //
  int max_vector_length = warpSize;
  int max_num_workers = div_round_up(blockDim.x / max_vector_length);
```

```
 int worker_id = threadIdx.x / max_vector_length;
 int vector_lane_id = threadIdx.x % max_vector_length;
 //
 if ( vector_lane_id == 0 ) { // masks out all other vector lanes
  // loop specific
  int num_gangs = max_num_gangs;
  // int num_gangs = 4; // if gang(4) is specified
  int num_workers = max_num_workers;
  // int num_workers = 2; // if worker(2) is specified
  int vector_length = 1;
  //
  if ( gang_id < num_gangs && worker_id < num_workers ) {
   int gang_worker_id = gang_id*num_workers + worker_id;
   int gang_worker_tile_size = div_round_up(m,num_gangs*num_workers);
   for ( int i = 1+gang_worker_id*gang_worker_tile_size;
        i <= (gang_worker_id+1)*gang_worker_tile_size; i++ ) {
    if ( i <= m) {
     x[i] = 1;
    }
   }
  }
 }
}

int main(int argc, char** argv) {
 // initialization of x, m, NUM_WORKERS,
 // and MAX_VECTOR_LENGTH (32 or 64 depending on arch)
 // ...
 int NUM_THREADS = NUM_WORKERS*MAX_VECTOR_LENGTH;
 hipLaunchKernelGGL((mykernel),dim3(m/NUM_WORKERS),dim(NUM_THREADS),x,m);
}
```

Mixed gang-vector parallelism

In this section, we discuss scenarios where a gang clause appears together with a vector clause on a loop directive. In this case, the compiler switches to gang-partitioned, worker-single, vector-partitioned mode, i.e. the loop is partitioned across all available gangs and all available vector lanes of a single worker.

Example 11 (Gang-vector parallelism):

```
!$acc parallel
!$acc loop gang vector
! other variants:
!$acc loop gang vector(8)
!$acc loop gang(4) vector
!$acc loop gang(4) vector(8)
```

```
do i = 1,m
  x(i) = 1
end do
!$acc end parallel
```

An equivalent HIP C++ implementation may look as follows:

```
// includes and definitions
// ...

__global__ void mykernel(float* x, int m) {
  // generic preamble
  int max_num_gangs = gridDim.x;
  int gang_id = blockIdx.x;
  int gang_tid = threadIdx.x;
  //
  int max_vector_length = warpSize;
  int max_num_workers = div_round_up(blockDim.x / max_vector_length);
  int worker_id = threadIdx.x / max_vector_length;
  int vector_lane_id = threadIdx.x % max_vector_length;
  //
  if ( true ) { // all vector lanes are active
    // loop specific
    int num_gangs = max_num_gangs;
    // int num_gangs = 4; // if gang(4) is specified
    int num_workers = 1;
    int vector_length = max_vector_length;
    // int vector_length = 8; // if vector(8) is specified
    //
    if ( gang_id < num_gangs && worker_id < num_workers && vector_lane_id < vector_length )
      int gang_vector_id = gang_id*vector_length + worker_id;
      int gang_vector_tile_size = div_round_up(m,num_gangs*vector_length);
      for ( int i = 1+gang_vector_id; i <= m; i+=gang_vector_tile_size ) {
        x[i] = 1;
      }
    }
  }
}

int main(int argc, char** argv) {
  // initialization of x, m, NUM_WORKERS,
  // and MAX_VECTOR_LENGTH (32 or 64 depending on arch)
  // ...
  int NUM_THREADS = NUM_WORKERS*MAX_VECTOR_LENGTH;
  hipLaunchKernelGGL((mykernel),dim3(m/NUM_WORKERS),dim(NUM_THREADS),x,m);
}
```

“‘

Determining a statement’s parallelism level

In this section, how we can assign parallelism levels to statements in the code. The main challenge is that the parallelism level of loops and program flow statements depends on the parallelism-level of the statements in their body.

For example, an if-statement in an gang-redundant or gang-partitioned code section must be evaluated also by all workers if one of the statements is a worker-partitioned loop, and if there is a vector-partitioned loop somewhere in the body, this if-statement has to be evaluated by all vector lanes in the worker. However, other statements in that if-statement’s body that do not result in worker- or vector-partitioned loop sections must still only be enabled for resources associated with the original gang-redundant or -partitioned parallelism level.

Let’s walk through the following example:

Example XYZ (Determining parallelism levels)

```
!$acc parallel
gr_stmt1; // gang-redundant statement
if ( gr_to_wp_if1 ) then ! initially gang-redundant,
                         ! eventually worker-partioned if statement
  !$acc parallel loop gang
  do i = 1,m
    gp_stmt1;  ! statement in gang-partitioned loop
    !$acc parallel loop worker
    do j = 1,n
      wp_stmt1;  ! statement in worker-partitioned loop
    end do
  end do
  gr_stmt2; // gang-redundant statement
else ! initially gang-redundant,
     ! eventually worker-partioned else branch
     ! of if statement
  gr_stmt3
endif
gr_stmt4;
!$acc end parallel
```

TBA

GPUFORT’s solution

GPUFORT identifies the parallelism level of all statements with one top-down and bottom-up sweep through the translator tree. During the top-down sweep, the currently active parallelism level is forwarded to the statements in the body of program flow control or loop statement. During the backtracing, i.e. the bottom-up sweep, the parallelism level of the program flow control and loop statements is determined by determining the maximum parallelism level of the statements in the body.

Depending on the parallelism level, more or less resources are masked out when executing a statement:

Parallelism level	Activation mask	Remark
GR-WS-VS	`linearize(hipThreadIdx,blockDim) == 0`	
GP-WS-VS	`linearize(hipThreadIdx,blockDim) == 0`	Extraneous gangs masked out via loop tiling
G[RP]-WP-VS	`linearize(hipThreadIdx,blockDim) % warpSize == 0`	Extraneous gangs & workers masked out via loop tiling or if statements.
G[RP]-W[RP]-VP	`true`	Extraneous gangs, workers, and vector lanes masked out via loop tiling or if statements.

In the above table, the function `linearize` computes a linear index from the `threadIdx` `dim3` struct, where `blockDim` contains the strides: In the python-like pseudocode, the computation looks as follows:

```
def linearize(hipThreadIdx,blockDim):
    return (hipThreadIdx.x
            + blockDim.x*hipThreadIdx.y +
            blockDim.x*blockDim.y*hipThreadIdx.z)
```