DECADES Documentation and Specifications

A Introduction

Below we provide in-depth descriptions about the DECADES compiler and the DECADES simulator.

B Compiler

At a high level, DEC++ is a convenient wrapper around a collection of Clang visitor and LLVM transformation passes. DEC++ can be invoked directly on a set of `.cpp` source files and mirrors the behavior of a traditional C++ compiler, e.g. g++. Using various compiler *modes*, the source code can be compiled and transformed to target various features of the DECADES architecture. The output of DEC++ can be one of: (1) an x86 native executable, (2) LLVM computations graphs for the Pythia simulator (see Section C), or (3) a RISC-V binary to execute on our target chip architecture composed of OpenPiton and Ariane.

B.1 Annotations and Limitations

We begin by discussion annotations and limitations of the compiler for C++ programs. While the list may seem daunting, we have found that for many applications of interest these restrictions and annotations have not been an issue as they are straightforward to incorporate.

- **Kernels:** DEC++ targets a single kernel function. This function must be named `_kernel_` and have a return type of `void`.
- **Tile ID and Tile Count:** Tiles are the base computation model for DECADES. Therefore, the last two arguments to the `_kernel_` function must be two integers that represent a tile id for the kernel and the number of tiles. When invoking `_kernel_` (e.g. from `main`), the values for these arguments should be `DECADES_TILE_ID` and `DECADES_NUM_TILES`, both defined in the `DECADES/DECADES.h` header file that should be included. DEC++ will automatically populate these argument values based on the tile argument (discussed below).
- **Single Function:** DEC++ does not handle inter-function analysis. Thus, all functions called by the kernel are inlined in the first pass. The behavior of DEC++ is undefined if functions cannot be inlined. This includes code in external libraries. The compiler will warn, but not error, upon finding a function that it cannot inline.
- **I/O and Memory Allocation:** Kernel functions should not allocate memory or perform I/O operations. These should be done in the wrapper functions (e.g. main).
- **Recursion:** The source code cannot contain recursive functions, as these are not supported by DEC++.

B.2 DEC++ Modes

DEC++ has four different modes that it can run. These modes are invoked when DEC++ is run at the command line using the `--comp_model` (or `-m`) flag.

B.2.1 Native ("n"): The native mode simply compiles the source files with no LLVM transformations or passes. This mode is useful for verification of correct program output and debugging, as well as application development outside of any DECADES features.

B.2.2 DECADES Base ("db"): This is the default DEC++ mode. It identifies the `_kernel_` function, performs function inlining and wraps the function invocation in the tile launcher. This mode supports all three backend targets: x86 emulation, Pythia Simulator, and RISC-V binary.

B.2.3 Decoupled Implicit ("di"): This compilation mode slices the `_kernel_` function into supply and compute programs completely *automatically*. The supply and compute kernels communicate through a DECADES API that is included and linked in the compilation process. The compute slice does not access memory *at all*, and instead queries the supply unit for values.

The supply slice additionally goes through a second pass where *terminal* loads are identified and annotated so that can be handled in a special way by the architecture.

These API calls are *not* inlined, as the simulator can use them to explore architectural innovations. We provide a shared memory implementation, which allows for x86 emulation even without special hardware features.

B.2.4 Biscuit ("b"): This compilation mode assumes that the programmer has used our intelligent storage tile API (see Section B.5) in their program. The compiler then links to the library implementation and launches an intelligent storage tile to service the requests made by the API.

B.3 Tile Model and Single Program Multiple Thread (SPMT) Parallelism

The DEC++ programming model is tile based. Recall that the kernel function must take a tile id and tile count argument.

At compile time, the user can provide the tile count flag ("-t") to specify how many tiles to compile for. We denote this value as *N*. The user can then assume that: (1) *N* kernel functions will be run in parallel; (2) each kernel will be given a contiguous unique tile id, which is passed in the `DECADES_TILE_ID` argument; and (3) each kernel will receive the value N to fulfill the `DECADES_NUM_TILES` argument.

This provides an SPMT (single program, multiple thread) programming model by default. For programs that are not parallel, a value of 1 may be used for "-t", and the tile id and tile count arguments can safely be ignored (although integers must still be provided). The default value for "-t" is 1, so that users begin by programming a single tile.

B.3.1 Interaction with Modes:

- **Native:** The tile arguments are ignored for native mode.
- **DECADES Base:** The DECADES Nase mode can be used with tiles to provide N compute tiles, similar to a classic SPMT programming model (e.g. OpenMP).
- **Decoupled Implicit:** The DECADES Decoupled Implicit mode can also be used with tiles, which creates *N* compute tiles and *N* supply tiles that communicate pairwise.
- **Biscuit**: The tile arguments are currently ignored for biscuit mode.

B.3.2 Synchronization API: Communication across tiles is undefined except when using our synchronisation API. We currently support one instruction:

- `DECADES_BARRIER()`: An execution barrier that requires all tiles to reach the barrier and flush memory to a consistent view before continuing.

B.3.3 Implementation: The DECADES tile model is implemented using OpenMP under the hood. The first pass is a Clang visitor pass that duplicates kernel files for later slicing. It then wraps the kernel launch in an OpenMP environment and populates the tile id and tile count arguments.

B.4 Backends

DEC++ currently targets three backends. These backends are specified when DEC++ is run at the command line using the `--target` (or `-r`) flag.

B.4.1 x86 Emulation ("x86"): This backend creates an executable that can run natively on an x86 machine. Decoupled interactions, biscuit, and tiles are all emulated using shared memory. For decouling and biscuit, runtime information is gathered and printed at the end of the execution.

For the DECADES Decoupled Implicit mode, the displayed information consists of the number of interactions between the compute and supply slices, the types of data being transferred, and how many terminal loads were issued. For biscuit, the printed output includes the size of the communication buffer transferred between tiles and statistics about the vector widths of the computations passed to the biscuit tile. For the DECADES Base mode, the output is simply that of a shared memory SPMT program that can be used for performance benchmarking (e.g. against single-threaded application baselines).

All DEC++ modes and multiple tiles are supported for this backend.

B.4.2 Simulator ("simulator"): To target Pythia, the DECADES simulator, the user must provide the "-spp" (simulator preprocessing) flag with the path to the Pythia simulator preprocessing script (located in the simulator repository). This produces (1) instrumented x86 executables and (2) LLVM computation graphs. To run a DEC++ compiled application on the simulator, the instrumented x86 executables must be run to generate memory traces.

Once the memory traces have been created, the simulator can be run on the DEC++ output. For more details on the simulator, see Section C.

All modes except biscuit and native are supported for this backend. Multiple tiles are supported for this backend.

B.4.3 RISC-V ("riscv64"): This backend produces binaries to run on the multi-core RISC-V architecture composed of OpenPiton and Ariane, the hardware architecture of DECADES.

Currently only the DECADES Base mode is supported for this backend. Multiple tiles can be specified though, providing a SPMT programming target.

B.5 Intelligent Storage Tile API

Below are the functions a programmer can call to utilize an intelligent storage tile for parallelism and efficient memory accesses. Each intelligent storage tile contains a bit-serial processor ("biscuit"), which can quickly compute on large batches of parallel computations with SIMD instructions.

- `void` **`decades_ist_init()`** - initialize all intelligent storage tile memories
- `void` **`decades_ist_flag_set_char`**`(int` *`bid`*`, IS_compute_context*` *`local_ist_context`*`, char*` *`base`*`, unsigned long` *`offset`*`)` - set the value stored in the local scratchpad at *base[offset]* to be 1
  - *`bid`* = biscuit ID
  - *`local_ist_context`* = the scratchpad information (scratchpad and scratchpad size) local to the computation kernel
  - *`base`* = the array to store graph information
  - *`offset`* = the location in the array whose value is to be set to 1
- `void` **`decades_ist_incr_uint`**`(int` *`bid`*`, IS_compute_context*` *`local_ist_context`*`, unsigned int*` *`base`*`, unsigned long` *`offset`*`)` - increment the value stored in the local scratchpad at *base[offset]*
  - *`bid`* = biscuit ID
  - *`local_ist_context`* = the scratchpad information (scratchpad and scratchpad size) local to the computation kernel
  - *`base`* = the array to store graph information
  - *`offset`* = the location in the array whose value is to be incremented
- `void` **`decades_ist_flush`**`(int` *`bid`*`, IS_compute_context*` *`local_ist_context`*`)` - send batch of kernel computations to a specific biscuit in asynchronous biscuit mode
  - *`bid`* = biscuit ID
  - *`local_ist_context`* = the scratchpad information (scratchpad and scratchpad size) local to the computation kernel
- `void` **`decades_ist_load_uint_values`**`(int` *`bid`*`, unsigned int*` *`values`*`)` - load unsigned integer *values* into the local memory of biscuit with biscuit ID *bid*
  - *`bid`* = biscuit ID
  - *`values`* = the unsigned integer values to be stored in local memory
- `void` **`decades_ist_store_uint_values`**`(int` *`bid`*`, unsigned int*` *`values`*`)` - store unsigned integer *values* at the scratchpad addresses of biscuit with biscuit ID *bid*
  - *`bid`* = biscuit ID
  - *`values`* = the unsigned integer values to be stored in the scratchpad addresses
- `void` **`decades_ist_load_char_values`**`(int` *`bid`*`, char*` *`values`*`)` - load character *values* into the local memory of biscuit with biscuit ID *bid*
  - *`bid`* = biscuit ID

  - *values* = the character values to be stored in local memory
- `void` **`decades_ist_store_char_values`**`(int` *`bid`*`, char*` *`values`*`)` - store character *values* at the scratchpad addresses of biscuit with biscuit ID *bid*
  - *bid* = biscuit ID
  - *values* = the character values to be stored in the scratchpad addresses
- `void` **`decades_ist_finalize`**`(int` *`bid`*`)` - notify biscuit with biscuit ID *bid* that all computation has been completed so biscuit can shut down
  - *bid* = biscuit ID
- `void` **`print_ist_info()`** - print statistics collected (batch information, time spent waiting on compute and memory copies)

C Simulator

This section describes the API for the DECADES Simulator. Specifically, it provides guidelines that allow one to efficiently design accelerators that can be integrated together into the pre-existing framework. For information on building and running the simulator, see: `https://github.com/PrincetonUniversity/pythia` for the README.

C.1 Simulator Overview

The DECADES simulator is designed to be a timing simulator that provides performance estimates for the heterogeneous systems that appear in the DECADES architecture. Specifically, we hope to simulate applications leveraging multiple tiles (cores, accelerators, intelligent storage) running simultaneously.

C.2 The Digestor

At the center of the simulator is the *digestor*, which is responsible for instantiating all tiles, orchestrating their forward progress, and coordinating communication among them. It can also be thought of as a Network on Chip that receives packets (Transactions in this case) and sends them to their destination tile at the right cycle time for that tile. Hence, the goal of this API is to describe how every tile model must be designed to interface properly with it. The digestor is defined as the class Simulator and is found in pythia/sim/sim.h and pythia/sim/sim.cc. We show the interface functions a tile designer should be aware of below, but we discuss them in more depth in later sections.

```
//code snippet from pythia/sim/sim.h
void Simulator::InsertTransaction(Transaction* t, uint64_t cycles); //called by a tile to send
    transaction t to another tile (source and destination are contained in t).
void registerTile(Tile* tile, int tid); //this adds a tile to a simulator, explicitly assigning
    tile id tid to it
void registerTile(Tile* tile); //this adds a tile to the digestor, which auto assigns the tile an
    id
```

C.3 Tiles

Tiles run alongside each other, each being called upon by the digestor to take a one cycle step and increment its cycle count. Note that the tiles may run at different clock speeds; the digestor is aware of this and calls upon the tiles to step based on their provided clockspeeds. Tiles communicate by creating transactions and enqueuing them within the digestor, via a function call. The digestor is responsible for sending transactions to the destination tile at the right clock cycle for the destination tile, which it does by explicitly calling upon Tiles to receive messages.

C.4 Tile Abstract Class

To enforce the above behaviors, each tile must inherit the Tile abstract class in pythia/sim/tile/Tile.h, with the code shown below. It includes some virtual functions that must be implemented in any Tile subclass. See an example derived tile in pythia/sim/tile/tile_example/ExampleTile.cc. To run it and see the key highlights on

how it was designed and the shell script to correctly set up the example files, follow the README at pythia/sim/tile/tile_example/README.

```
//code snippet from pythia/sim/tile/Tile.h
class Tile {
 public:
  int id; //tile id
  Simulator* sim; //the digestor
  uint64_t clockspeed; //clockspeed in MHertz
  uint64_t cycles; //cycle count

Tile(Simulator* sim, int clockspeed): sim(sim),clockspeed(clockspeed) {
    cycles=0;
  }
  virtual bool process()=0; //function to take a one-cycle simulation step, called by digestor
  virtual bool ReceiveTransaction(Transaction* t)=0; //function to receive a transaction, called
     by digestor
};
```

C.4.1 bool Tile::process(): This function is called by the digestor to force a tile to take a one-cycle step. Any implementation of it is required to do a cycle's worth of work, increase its cycle count, and return a boolean indicating if the tile could have remaining work (e.g., if it still could be invoked by other tiles or still has queued transactions it's processing). Note that more than a cycle's worth of work could be be done as long as the tile tracks when the results should be ready and acts on it when its cycle count has reached that future time. This is shown in pythia/sim/tile/tile_example/ExampleTile.cc.

C.4.2 bool Tile::ReceiveTransaction(Transaction* t): This function is called by the digestor to force a tile to receive a message. A boolean is returned to indicate if the message was successfully received or couldn't be received, perhaps due to resource constraints such as full message queues. The tile may chose to act immediately on a message pushed to it or enqueue it and process all its enqueued messages when the digestor calls process(). We will discuss the contents of *Transaction*, which represent message packets in Section C.5.1 below.

C.5 Inter-tile Communication

As discussed above, tiles receive messages through ReceiveTransaction() calls. However, if a tile wants to send a message to another tile, it simply calls sim->InsertTransaction(t, cycle), shown below. *t* is the transaction to be sent, and cycles is the clock cycle the tile intends the digestor to assume it was sent (cycle is typically the cycle count of the tile *cycles*).

Figure 1 illustrates inter-tile communication in which Tile0 wishes to communicate to Tile2, using Tile1 as an intermediary buffer. Tile0 sends the transaction t1 to the digestor at Tile0's cycle C1. The digestor waits until Tile1 reaches cycle C2 (the real-time equivalent of C1+some queue delay) and sends the transaction to Tile1. Tile 1 enqueues the transaction, modifies it to t2 and, after an appropriate number of cycles representing some queuing delay model internal to Tile1, sends the transaction to the digestor. Finally, the digestor relays the transaction to Tile2.

```
//code snippet from pythia/sim/tile/Tile.h
class Transaction {
public:
  Transaction(int id, int src_id, int dst_id) : id(id), src_id(src_id), dst_id(dst_id) {}; //
     constructor for inter-tile communication
  int id;
  int src_id;
  int dst_id;
  bool complete=false;
  TransType type=DEFAULT;
};
```

C.5.1 Transactions: All communication between tiles is done through Transactions. We provide a generic transaction base class. Tile designers can then inherit that class to add special fields required for the communication and synchronization needs of their tiles. This could also include details like data width and special transaction types that a tile uses to determine the required actions (e.g., forward a message to an intermediary tile, send a direct

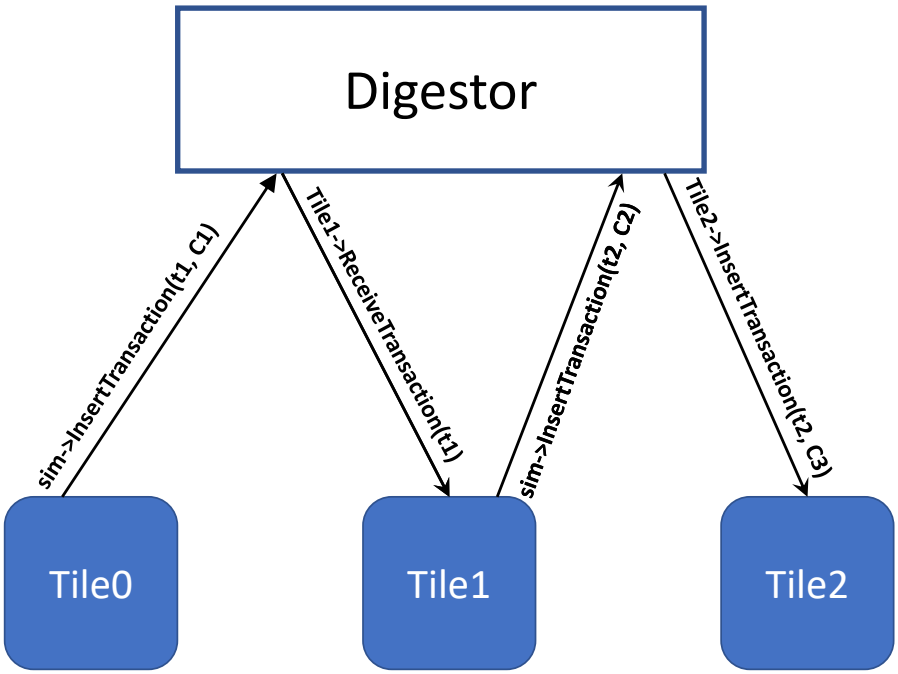

Figure 1: Inter-tile Communcation.

response, etc.) or number of cycles it would take to process a request. Above is the code for the the Transaction base class, from pythia/sim/tile/Tile.h.

C.6 Putting it Together

C.6.1 Instantiating a Tile:

1. Design a tile subclass that inherits the tile class in pythia/sim/tile/Tile.h and place the subclass source file in a new pythia/sim/tile/chosen_name directory.
2. In pythia/sim/tile/Tile.h, create new Transaction types and subclasses relevant for the communication of the tiles.
3. Code your main file and place it in pythia/sim/tile/chosen_name/main.cc, matching the code snippet below:

```
#include "../../sim.h" //include the header file for the simulator
#include "tile/chosen_name/ExampleTile.h" //include header file for your tile

int main(int argc, char const *argv[]) {
    Simulator* simulator=new Simulator();
    Tile* tile0 = new ExampleTile(simulator,2000);
    Tile* tile1 = new ExampleTile(simulator,333);
    simulator->registerTile(tile0,0); //assign tile id 0 to tile0
    simulator->registerTile(tile1,1); //assign tile id 1 to tile1

    simulator->run(); //run the simulator, which steps through each of the tiles until they're
        done working
    return 0;
}
```

4. In pythia/sim/CMakeLists.txt, remove "main.cc" and add "tile/chosen_name/main.cc" and "tile/chosen_name/ExampleTile.cc" to the list of source files in "ADD_EXECUTABLE".
5. Compile the simulator as normal.
6. Cd to pythia/bin and run the simulator with ./sim (add any command line arguments as required by your main file)

C.6.2 Integrating Tiles with Default Pythia Cores: Pythia comes with default core tile models, each with configurable hardware structures, a private L1 cache, and a shared L2 cache. We anticipate that it would be useful to integrate new tiles with the existing core tiles. Below are the steps to design and instantiate a tile with the default core tiles.

1. Follow steps 1-2 in Section C.6.1 above.
2. In pythia/sim/main.cc, find the commented region (shown in the code snippet below) and, just above it, create a new tile and register it, as shown below.

```
//snippet from pythia/sim/main.cc

/********
register the other tiles here
e.g.
Tile* tile = new ExampleTile(simulator,2000);
simulator->registerTile(tile,tileid); //manual tile id assignment
or
simulator->registerTile(tile); //auto tile id assignment
********/
```

3. In pythia/sim/CMakeLists.txt, add "tile/ExampleTile.cc" to the list of source files in "ADD_EXECUTABLE".
4. Compile and run the simulator as normal!