targetDP Specification

Alan Gray and Kevin Stratford

Contents

Chapter 1

Introduction

COMMENT: TO DO.

1.1 Abstract

**

1.2 Motivation

**

1.3 Glossary

- Accelerator: ***
- Host: ***
- Target: ***
- CUDA: ***

Chapter 2

Execution model

The terminology "host" is used to refer to the CPU that is hosting the execution of the application, and "target" to refer to the device targeted for execution of data parallel operations. The target can be the same CPU as the host, or it can be a separate device such as an accelerator.

The targetDP API follows the fork-join model of parallel execution. When the application initiates, a single thread executes sequentially on the host, until it encounters a function to be launched on the target. This function will be executed by a team of threads on the target cooperating in a data-parallel manner (e.g. for a structured grid problem each thread is responsible for a subset of the grid).

To ensure that the target function has completed, a `targetSynchronize` statement should follow the code location where target function is launched. When the initial thread encounters this statement, it will wait until the target region has completed. It is possible for the initial thread to execute other instructions (which do not depend on the results of the target function), after the function launch but before the syncronisation call. This may result in overlapping of host and target execution, and hence optimisation, in some implementations. Once the target function has completed, the initial thread will continue sequentially until another target function launch is encountered.

Within each target function, each thread is given a unique index which it uses to work in a data-parallel manner. Each thread works independently from all others, but usually operating on a shared data structure where the index is used to determine the portion of data to process.

COMMENT: Possibly add targetSyncThreads() to allow communication within target functions?]

Chapter 3

Memory model

3.1 Model overview

The targetDP model draws a distinction between the memory spaces accessed by the host and target, such that code executed on the host always accesses the host memory space, and code executed on the target (i.e. within target functions) always accesses the target memory space. The host memory space can be initialised using regular `C/C++` functionality, and the targetDP API provides the functionality necessary to manage the target memory space and transfer data between the host and target. For each data-parallel data structure, the programmer should create both host and target copies, and should update these from each other as and when required.

3.1.1 Host memory model

The sequential thread executing host code should always access host data structures. The memory model is the same as one would expect from a regular sequential application.

3.1.2 Target memory model

The team of threads performing the execution of target functions should always act on target data structures. These can take one of 3 forms:

1. Those created using the targetDP memory allocation API functions. These are shared between all threads in the team, where each thread should use

its unique index to access the portion of data for which it is responsible.

2. Those created using the targetDP constant data management functionality. These are read-only and normally used for relatively small amounts of constant data.

3. Those declared within the body of a target function. These are private to each thread in the team and should be used for temporary scratch structures.

COMMENT: For 3, at the moment any data declared in the function but above the targetTLP keyword will be private for GPU threads but shared for CPU (since outside parallel region). So either need to make this clearer above, or somehow move OpenMP parallel region to target launch.

3.2 Implementation

3.2.1 C

The target memory structures will exist on the same physical memory as the host structures. The implementation may either use separate target copies (managed using regular C/C++ memory management functionality), or use pointer aliasing for the target versions such that a reference to any part of a target structure will correspond to exactly the same physical address as that of the corresponding host structure.

3.2.2 CUDA

The target memory space will exist on the distinct GPU memory, i.e. in a separate memory space from the host structures.

Chapter 4

Memory Management

4.1 targetMalloc

4.1.1 Description

The `targetMalloc` function allocates memory on the target.

4.1.2 Syntax

```
void targetMalloc(void **targetPtr, size_t n);
```

- `targetptr`: A pointer to the allocated memory.
- `n`: The number of bytes to be allocated.

4.1.3 Example

See Line 1 in Figure 6.2 in Section 6.

4.1.4 Implementation

C

`malloc`

CUDA

`cudaMalloc`

4.2 targetCalloc

4.2.1 Description

The `targetCalloc` function allocates, and initializes to zero, memory on the target.

4.2.2 Syntax

```
void targetCalloc(void **targetPtr, size_t n);
```

- `targetptr`: A pointer to the allocated memory.
- `n`: The number of bytes to be allocated.

4.2.3 Example

Analogous to Line 1 in Figure 6.2 in Section 6.

4.2.4 Implementation

C

`calloc`

CUDA

`cudaMalloc` followed by `cudaMemset`

4.3 targetFree

4.3.1 Description

The `targetFree` function deallocates memory on the target.

4.3.2 Syntax

```
void targetFree(void *targetPtr);
```

- `targetPtr`: A pointer to the memory to be freed.

4.3.3 Example

See Line 10 in Figure 6.2 in Section 6.

4.3.4 Implementation

C

`free`

CUDA

`cudaFree`

4.4 copyToTarget

4.4.1 Description

The `copyToTarget` function copies data from the host to the target.

4.4.2 Syntax

```
void copyToTarget(void *targetData, const void *data, size_t n);
```

- `targetData`: A pointer to the destination array on the target.
- `data`: A pointer to the source array on the host.
- `n`: The number of bytes to be copied.

4.4.3 Example

See Line 3 in Figure 6.2 in Section 6.

4.4.4 Implementation

C

`memcpy`

CUDA

`cudaMemcpy`

4.5 copyFromTarget

4.5.1 Description

The `copyFromTarget` function copies data from the target to the host.

4.5.2 Syntax

```
void copyFromTarget(void *data, const void *targetData, size_t n);
```

- `data`: A pointer to the destination array on the host.
- `targetData`: A pointer to the source array on the target.
- `n`: The number of bytes to be copied.

4.5.3 Example

See Line 9 in Figure 6.2 in Section 6.

4.5.4 Implementation

C

`memcpy`

CUDA

`cudaMemcpy`

4.6 targetConst

4.6.1 Description

The `__targetConst__` keyword is used in a variable or array declaration to specify that the corresponding data can be treated as constant (read-only) on the target.

4.6.2 Syntax

```
__targetConst__ type variableName
```

- `variableName`: The name of the variable or array.
- `type`: The type of variabe or array.

4.6.3 Example

TO DO.

4.6.4 Implementation

C

Holds no value

CUDA

`__constant__`

4.7 copyConstToTarget

4.7.1 Description

The `copyConstToTarget` function copies data from the host to the target, where the data will remain constant (read-only) during the execution of functions on the target.

4.7.2 Syntax

```
void copyConstToTarget(void *targetData, const void *data, size_t n);
```

- `targetData`: A pointer to the destination array on the target. This must have been declared using the `__targetConst__` keyword.
- `data`: A pointer to the source array on the host.
- `n`: The number of bytes to be copied.

4.7.3 Example

See Line 4 in Figure 6.2 in Section 6.

4.7.4 Implementation

C

```
memcpy
```

CUDA

```
cudaMemcpyToSymbol
```

4.8 copyConstFromTarget

4.8.1 Description

The `copyConstFromTarget` function copies data from a constant data location on the target to the host.

4.8.2 Syntax

```
void copyConstToTarget(void *targetData, const void *data, size_t n);
```

- `data`: A pointer to the destination array on the host.
- `targetData`: A pointer to the source array on the target. This must have been declared using the `__targetConst__` keyword.
- `n`: The number of bytes to be copied.

4.8.3 Example

Analogous to Line 4 in Figure 6.2 in Section 6.

4.8.4 Implementation

C

```
memcpy
```

CUDA

```
cudaMemcpyFromSymbol
```

4.9 targetConstAddress

4.9.1 Description

The `targetConstAddress` function provides the target address for a constant object.

4.9.2 Syntax

```
void targetConstAddress(void **address, objectType object);
```

- `address` (output): The pointer to the constant object on the target.
- `objectType`: The type of the object.
- `object` (input): The constant object on the target. This should have been declared using the `__targetConst__` keyword.

4.9.3 Example

TO DO.

4.9.4 Implementation

C

Explicit copying of address.

CUDA

`cudaGetSymbolAddress`

4.10 targetInit3D

4.10.1 Description

The `targetInit3D` initialises the environment required to perform any of following the targetDP 3D lattice operations.

4.10.2 Syntax

```
void targetInit3D(size_t extent, size_t nFields);
```

- `extent`: The total extent of data parallelism (e.g. the number of lattice sites).
- `nFields`: The extent of data resident within each parallel partition (e.g. the number of fields per lattice site).

4.10.3 Example

TO DO.

4.10.4 Implementation

C

TO DO.

CUDA

TO DO.

4.11 targetFinalize3D

4.11.1 Description

The `targetFinalize3D` finalizes the targetDP 3D environment.

4.11.2 Syntax

```
void targetFinalize3D();
```

4.11.3 Example

TO DO.

4.11.4 Implementation

C

TO DO.

CUDA

TO DO.

4.12 copyToTargetBoundary3D

4.12.1 Description

The `copyToTargetBoundary3D` function copies the data corresponding to the boundaries of a 3D lattice from the host to the target.

4.12.2 Syntax

```
void copyToTargetBoundary3D(void *targetData, const void *data, size_t extent3D[3], size_t r
```

- `targetData`: A pointer to the destination array on the target.
- `data`: A pointer to the source array on the host.
- `extent3D`: An array of 3 integers corresponding to the 3D dimensions of the lattice.
- `nFields`: The number of fields per lattice site.
- `offset`: The number of sites from the lattice edge at which each boundary face should start.
- `depth`: The depth of each boundary face.

4.12.3 Example

TO DO.

4.12.4 Implementation

C

TO DO.

CUDA

TO DO.

4.13 copyFromTargetBoundary3D

4.13.1 Description

The `copyFromTargetBoundary3D` function copies the data corresponding to the boundaries of a 3D lattice from the target to the host.

4.13.2 Syntax

```
void copyFromTargetBoundary3D(void *data, const void *targetData, size_t extent3D[3], size_t
```

- `data`: A pointer to the destination array on the host.
- `targetData`: A pointer to the source array on the target.
- `extent3D`: An array of 3 integers corresponding to the 3D dimensions of the lattice.
- `nFields`: The number of fields per lattice site.
- `offset`: The number of sites from the lattice edge at which each boundary face should start.
- `depth`: The depth of each boundary face.

4.13.3 Example

TO DO.

4.13.4 Implementation

C

TO DO.

CUDA

TO DO.

4.14 copyToTargetPointerMap3D

4.14.1 Description

The copyToTargetPointerMap3D function copies a subset of lattice data from the host to the target. The sites to be included are defined using an array of pointers passed as input.

4.14.2 Syntax

```
void copyToTargetPointerMap3D(void *targetData, const void *data,
           size_t extent3D[3], size_t nField,
           int includeNeighbours, void** pointerArray);
```

- targetData: A pointer to the destination array on the target.
- data: A pointer to the source array on the host.
- extent3D: An array of 3 integers corresponding to the 3D dimensions of the lattice.
- nField: The number of fields per lattice site.
- includeNeighbours: A boolean switch to specify whether each included site should also have it's neighbours included (in the 19-point 3D stencil).
- pointerArray: An array of nSite pointers, where nSite is the total number of lattice sites. Each lattice site should be included unless the pointer corresponding to that site is NULL.

4.14.3 Example

TO DO.

4.14.4 Implementation

C

TO DO.

CUDA

TO DO.

4.15 copyFromTargetPointerMap3D

4.15.1 Description

The `copyFromTargetPointerMap3D` function copies a subset of lattice data from the target to the host. The sites to be included are defined using an array of pointers passed as input.

4.15.2 Syntax

```
void copyFromTargetPointerMap3D(void *data, const void *targetData,
           size_t extent3D[3], size_t nField,
           int includeNeighbours, void** pointerArray);
```

- `data`: A pointer to the destination array on the host.
- `targetData`: A pointer to the source array on the target.
- `extent3D`: An array of 3 integers corresponding to the 3D dimensions of the lattice.
- `nField`: The number of fields per lattice site.
- `includeNeighbours`: A boolean switch to specify whether each included site should also have it's neighbours included (in the 19-point 3D stencil).
- `pointerArray`: An array of `nSite` pointers, where `nSite` is the total number of lattice sites. Each lattice site should be included unless the pointer corresponding to that site is `NULL`.

4.15.3 Example

TO DO.

4.15.4 Implementation

C

TO DO.

CUDA

TO DO.

Chapter 5

Data Parallel Execution

5.1 targetEntry

5.1.1 Description

The `__targetEntry__` keyword is used in a function declaration or definition to specify that the function should be compiled for the target, and that it will be called directly from host code.

5.1.2 Syntax

```
__targetEntry__ functionReturnType functionName(...
```

- `functionName`: The name of the function to be compiled for the target.
- `functionReturnType`: The return type of the function.
- ... the remainder of the function declaration or definition.

5.1.3 Example

See Line 1 in Figure 6.1 in Section 6.

5.1.4 Implementation

C

Holds no value.

CUDA

5.2 target

5.2.1 Description

The `__target__` keyword is used in a function declaration or definition to specify that the function should be compiled for the target, and that it will be called from a `targetEntry` or another `target` function.

5.2.2 Syntax

```
__target__ functionReturnType functionName(...
```

- `functionName`: The name of the function to be compiled for the target.
- `functionReturnType`: The return type of the function.
- ... the remainder of the function declaration or definition.

5.2.3 Example

Analogous to Line 1 in Figure 6.1 in Section 6.

5.2.4 Implementation

C

Holds no value.

CUDA

5.3 targetHost

5.3.1 Description

The `__targetHost__` keyword is used in a function declaration or definition to specify that the function should be compiled for the host.

5.3.2 Syntax

```
__targetHost__ functionReturnType functionName(...
```

- `functionName`: The name of the function to be compiled for the host.
- `functionReturnType`: The return type of the function.
- . . . the remainder of the function declaration or definition.

5.3.3 Example

Analogous to Line 1 in Figure 6.1 in Section 6.

5.3.4 Implementation

C

Holds no value.

CUDA

```
extern ‘‘C’’ __host__
```

5.4 targetLaunch

5.4.1 Description

The `__targetLaunch__` syntax is used to launch a function across a data parallel target architecture.

5.4.2 Syntax

```
functionName __targetLaunch__(size_t extent) \
                (functionArgument1,functionArgument2,...);
```

- `functionName`: The name of the function to be launched. This function must be declared as `__targetEntry__` .
- `functionArguments`: The arguments to the function `functionName`
- `extent`: The total extent of data parallelism.

5.4.3 Example

See Line 6 in Figure 6.2 in Section 6.

5.4.4 Implementation

C

Holds no value.

CUDA

CUDA `<<<...>>>` syntax.

5.5 targetSynchronize

5.5.1 Description

The `targetSynchronize` function is used to block until the preceeding `__targetLaunch__` has completed.

5.5.2 Syntax

```
void targetSynchronize();
```

5.5.3 Example

See Line 7 in Figure 6.2 in Section 6.

5.5.4 Implementation

C

Dummy function.

CUDA

```
cudaThreadSynchronize
```

5.6 targetTLP

5.6.1 Description

The `__targetTLP__` syntax is used, within a `__targetEntry__` function, to specify that the proceeding block of code should be executed in parallel and mapped to thread level parallelism (TLP). Note that he behaviour of this operation depends on the defined virtual vector length (VVL), which controls the lower-level Instruction Level Parallelsim (ILP) (see following section).

5.6.2 Syntax

```
__targetTLP__(int baseIndex, size_t extent)
{
//code to be executed in parallel
}
```

- `extent`: The total extent of data parallelism, including both TLP and ILP
- `baseIndex`: the TLP index. This will vary from 0 to `extent-VVL` with stride VVL. This index should be combined with the ILP index to access shared arrays within the code block (see following section).

5.6.3 Example

See Line 4 in Figure 6.1 in Section 6.

5.6.4 Implementation

C

OpenMP parallel loop.

CUDA

CUDA thread lookup.

5.7 targetILP

5.7.1 Description

The `__targetILP__` syntax is used, within a `__targetTLP__` region, to specify that the proceeding block of code should be executed in parallel and mapped to instruction level parallelism (ILP), where the extent of te ILP is defined by the virtual vector length (VVL) in the targetDP implementation. COMMENT: need to document VVL in more detail somewhere and link.

5.7.2 Syntax

```
__targetILP__(int vecIndex)
{
//code to be executed in parallel
}
```

- `baseIndex`: the ILP index. This will vary from 0 to `VVL-1`. This index should be combined with the TLP index to access shared arrays within the code block (see previous section).

5.7.3 Example

See Line 9 in Figure 6.1 in Section 6.

5.7.4 Implementation

C

Short vectorizable loop.

CUDA

Short vectorizable loop.

5.8 targetCoords3D

5.8.1 Description

The `targetCoords3D` function provides the 3D lattice coodinates corresponding to a specified linear index.

5.8.2 Syntax

```
void targetCoords3D(int coords3D[3], int extent3D[3], int index);
```

- `coords3D` (output): an array of 3 integers to be populated with ther 3D coordinates.
- `extent3D` (input): An array of 3 integers corresponding to the 3D dimensions of the lattice.
- `index` (input): the linear index.

5.8.3 Example

TO DO

5.8.4 Implementation

C

TO DO

CUDA

TO DO

5.9 targetIndex3D

5.9.1 Description

The `targetIndex3D` function returns the linear index corresponding to a specified set of 3D lattice coodinates.

5.9.2 Syntax

```
int targetIndex3D(int Xcoord,int Ycoord,int Zcoord,int extent3D[3]);
```

- `Xcoord` (input): the specified coordinate in the X direction.
- `Ycoord` (input): the specified coordinate in the Y direction.
- `Zcoord` (input): the specified coordinate in the Z direction.
- `extent3D` (input): an array of 3 integers corresponding to the 3D dimensions of the lattice.

5.9.3 Example

TO DO

5.9.4 Implementation

C

TO DO

CUDA

TO DO

Chapter 6

Examples

COMMENT: TO DO: describe this example.

```
__targetEntry__ void scale(double* t_field) {

  int baseIndex;
  __targetTLP__(baseIndex, N) {

    int iDim, vecIndex;
    for (iDim = 0; iDim < 3; iDim++) {

      __targetILP__(vecIndex)  \
         t_field[iDim*N + baseIndex + vecIndex] =  \
         t_a*t_field[iDim*N + baseIndex + vecIndex];
    }
  }
  return;
}
```

Figure 6.1: ***

```
targetMalloc((void **) &t_field, datasize);

copyToTarget(t_field, field, datasize);
copyConstToTarget(&t_a, &a, sizeof(double));

scale __targetLaunch__(N) (t_field);
targetSynchronize();

copyFromTarget(field, t_field, datasize);
targetFree(t_field);
```

Figure 6.2: ***