

# targetDP Specification

Alan Gray and Kevin Stratford

January 14, 2016

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Abstract . . . . .	2
1.2	Motivation . . . . .	2
1.3	Glossary . . . . .	3
<b>2</b>	<b>Execution model</b>	<b>4</b>
<b>3</b>	<b>Memory model</b>	<b>6</b>
3.1	Model overview . . . . .	6
3.2	Implementation . . . . .	7
<b>4</b>	<b>Memory Management</b>	<b>8</b>
4.1	targetMalloc . . . . .	9
4.2	targetCalloc . . . . .	10
4.3	targetFree . . . . .	11
4.4	copyToTarget . . . . .	12
4.5	copyFromTarget . . . . .	13
4.6	targetConst . . . . .	14
4.7	copyConstToTarget . . . . .	15
4.8	copyConstFromTarget . . . . .	16
4.9	targetConstAddress . . . . .	17
4.10	targetInit3D . . . . .	18
4.11	targetFinalize3D . . . . .	19
4.12	copyToTargetBoundary3D . . . . .	20
4.13	copyFromTargetBoundary3D . . . . .	21

---

## CONTENTS

---

4.14	copyToTargetPointerMap3D . . . . .	22
4.15	copyFromTargetPointerMap3D . . . . .	23
<b>5</b>	<b>Data Parallel Execution</b>	<b>24</b>
5.1	targetEntry . . . . .	25
5.2	target . . . . .	26
5.3	targetHost . . . . .	27
5.4	targetLaunch . . . . .	28
5.5	targetSynchronize . . . . .	29
5.6	targetTLP . . . . .	30
5.7	targetILP . . . . .	31
5.8	targetCoords3D . . . . .	32
5.9	targetIndex3D . . . . .	33
<b>6</b>	<b>Example</b>	<b>34</b>

# Chapter 1

## Introduction

### 1.1 Abstract

The targetDP API provides an abstraction layer which allows applications to target Data Parallel hardware in a platform agnostic manner, by abstracting the memory spaces and hierarchy of hardware parallelism. Applications written using targetDP syntax are performance portable: the same source code can be compiled for different targets (where we currently support GPU accelerators and modern multicore or manycore SIMD CPUs), without performance overheads. The model is appropriate for abstracting the parallelism contained within each compute node, and can be combined with, e.g. MPI to allow use on systems containing multiple nodes. The targetDP API is primarily aimed at the types of parallelism found in grid-based applications, but may be applicable to a wider class of problems. This document introduces the targetDP memory and execution models, and specifies the syntax and functionality of the interface. Implementation details are also provided, and a simple example is given to demonstrate usage.

### 1.2 Motivation

It is becoming increasingly difficult for applications to exploit modern computers, which continue to increase in complexity and diversity with features including multicore/manycore CPUs, vector floating point units, accelerators such as GPUs and non-uniform distributed memory spaces. From a scientist's perspec-

1 tive, it is not only imperative to achieve performance, but also to retain main-  
2 tainability, sustainability and portability. The use of a simplistic, well structured  
3 and clearly defined abstraction layer such as targetDP can allow the program-  
4 mer to express the scientific problems in a way that will automatically achieve  
5 good performance across the range of leading hardware solutions.

## 6 **1.3 Glossary**

- 7     • CPU: Central Processing Unit. The main computer chip used in a system,  
8       suitable for a wide variety of computational tasks.
- 9     • Accelerator: A processing unit which is not used in isolation, but instead  
10       in tandem with the CPU, with the aim of improving the performance of  
11       key code sections.
- 12     • GPU: Graphics Processing Unit. A type of accelerator, originally evolved  
13       to render graphics content (particularly to satisfy demands of the gaming  
14       industry), but now widely used for general purpose computation.
- 15     • Host: Another term for the CPU that “hosts” the application.
- 16     • Data Parallel: The type of algorithmic parallelism involved where a  
17       single operation is performed to each element of a data set. The extent of  
18       parallelism is determined by the size of the data set.
- 19     • Target: The device targeted for execution of data parallel operations.  
20       Depending on the underlying hardware available, the target could simply  
21       be the a CPU, or it could be a separate device such as an accelerator.
- 22     • CUDA: Compute Unified Device Architecture. The parallel platform and  
23       model created by NVIDIA to allow general purpose programming of their  
24       GPU architectures.
- 25     • TLP: Thread Level Parallelism.
- 26     • ILP: Instruction Level Parallelism.

## 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 could be the same CPU as the host, or it could be a separate device such as an accelerator (depending on the hardware available).

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, in principle, 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 synchronisation 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.

---

1        In applications, it is sometimes necessary to perform reductions, where mul-  
2        tiple data values are combined in a certain way. For example, values existing on  
3        each grid point may be summed into a single total value. The targetDP model  
4        supports such operations in a simplistic way. It is the responsibility of the appli-  
5        cation to create the array of values (using standard targetDP functionality) to  
6        act as the input to the reduction operation. The application can then pass this  
7        array to the API function corresponding to the desired reduction operation (e.g.  
8        `targetDoubleSum` for the summation of an array of double-precision Values). If  
9        the required reduction operation does not yet exist, the user can simply extend  
10       the targetDP API using existing functionality as a template.

# 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



- 1        its unique index to access the portion of data for which it is responsible.
- 2        2. Those created using the targetDP constant data management functionality.
- 3        These are read-only and normally used for relatively small amounts of
- 4        constant data.
- 5        3. Those declared within the body of a target function. These are private to
- 6        each thread in the team and should be used for temporary scratch struc-
- 7        tures.
- 8        **COMMENT:** For 3, at the moment any data declared in the function but
- 9        above the targetTLP keyword will be private for GPU threads but shared for
- 10       CPU (since outside parallel region). So either need to make this clearer above,
- 11       or somehow move OpenMP parallel region to target launch.

## 12    **3.2 Implementation**

### 13    **3.2.1 C**

14    The target memory structures will exist on the same physical memory as the host

15    structures. The implementation may either use separate target copies (managed

16    using regular C/C++ memory management functionality), or use pointer alias-

17    ing for the target versions such that a reference to any part of a target structure

18    will correspond to exactly the same physical address as that of the correspond-

19    ing host structure.

### 20    **3.2.2 CUDA**

21    The target memory space will exist on the distinct GPU memory, i.e. in a sepa-

22    rate memory space from the host structures.

## <sup>1</sup> **Chapter 4**

# <sup>2</sup> **Memory Management**

## 1 **4.1 targetMalloc**

### 2 **4.1.1 Description**

3 The targetMalloc function allocates memory on the target.

### 4 **4.1.2 Syntax**

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

- 6     • targetPtr: A pointer to the allocated memory.
- 7     • n: The number of bytes to be allocated.

### 8 **4.1.3 Example**

9 See Line 1 in Figure 6.3 in Section 6.

### 10 **4.1.4 Implementation**

11 **C**

12 `malloc`

13 **CUDA**

14 `cudaMalloc`

## 1 **4.2 targetCalloc**

### 2 **4.2.1 Description**

3 The `targetCalloc` function allocates, and initialises to zero, memory on the  
4 target.

### 5 **4.2.2 Syntax**

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

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

### 9 **4.2.3 Example**

10 Analogous to Line 1 in Figure 6.3 in Section 6.

### 11 **4.2.4 Implementation**

#### 12 **C**

13 `calloc`

#### 14 **CUDA**

15 `cudaMalloc` followed by `cudaMemset`

## 1 **4.3 targetFree**

### 2 **4.3.1 Description**

3 The targetFree function deallocates memory on the target.

### 4 **4.3.2 Syntax**

5 `void targetFree(void* targetPtr);`

- 6
  - targetPtr: A pointer to the memory to be freed.

### 7 **4.3.3 Example**

8 See Line 11 in Figure 6.3 in Section 6.

### 9 **4.3.4 Implementation**

10 **C**

11 `free`

12 **CUDA**

13 `cudaFree`

## 1 **4.4 copyToTarget**

### 2 **4.4.1 Description**

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

### 4 **4.4.2 Syntax**

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

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

### 9 **4.4.3 Example**

10 See Line 3 in Figure 6.3 in Section 6.

### 11 **4.4.4 Implementation**

#### 12 **C**

13 `memcpy`

#### 14 **CUDA**

15 `cudaMemcpy`

## 1 **4.5 copyFromTarget**

### 2 **4.5.1 Description**

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

### 4 **4.5.2 Syntax**

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

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

### 9 **4.5.3 Example**

10 See Line 9 in Figure 6.3 in Section 6.

### 11 **4.5.4 Implementation**

#### 12 **C**

13 `memcpy`

#### 14 **CUDA**

15 `cudaMemcpy`

## 1 **4.6 targetConst**

### 2 **4.6.1 Description**

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

### 6 **4.6.2 Syntax**

7 `__targetConst__ type variableName`

- 8 • `variableName`: The name of the variable or array.
- 9 • `type`: The type of variable or array.

### 10 **4.6.3 Example**

11 `**TO DO**`.

### 12 **4.6.4 Implementation**

#### 13 **C**

14 Holds no value

#### 15 **CUDA**

16 `__constant__`



## 1 **4.7 copyConstToTarget**

### 2 **4.7.1 Description**

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

### 6 **4.7.2 Syntax**

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

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

### 12 **4.7.3 Example**

13 See Line 4 in Figure 6.3 in Section 6.

### 14 **4.7.4 Implementation**

#### 15 **C**

16 `memcpy`

#### 17 **CUDA**

18 `cudaMemcpyToSymbol`

## 1 **4.8 copyConstFromTarget**

### 2 **4.8.1 Description**

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

### 5 **4.8.2 Syntax**

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

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

### 11 **4.8.3 Example**

12 Analogous to Line 4 in Figure 6.3 in Section 6.

### 13 **4.8.4 Implementation**

#### 14 **C**

15 `memcpy`

#### 16 **CUDA**

17 `cudaMemcpyFromSymbol`

## 1 **4.9 targetConstAddress**

### 2 **4.9.1 Description**

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

### 5 **4.9.2 Syntax**

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

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

### 11 **4.9.3 Example**

```
12 **TO DO**.
```

### 13 **4.9.4 Implementation**

#### 14 **C**

15 Explicit copying of address.

#### 16 **CUDA**

```
17 cudaGetSymbolAddress
```

## 1 **4.10 targetInit3D**

### 2 **4.10.1 Description**

3 The targetInit3D initialises the environment required to perform any of the  
4 “3D” operations described in the rest of this chapter.

### 5 **4.10.2 Syntax**

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

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

### 11 **4.10.3 Example**

12 `**TO DO**.`

### 13 **4.10.4 Implementation**

#### 14 **C**

15 `**TO DO**.`

#### 16 **CUDA**

17 `**TO DO**.`

1 **4.11 targetFinalize3D**

2 **4.11.1 Description**

3 The targetFinalize3D finalises the targetDP 3D environment.

4 **4.11.2 Syntax**

5 void targetFinalize3D();

6 **4.11.3 Example**

7 **\*\*TO DO\*\*.**

8 **4.11.4 Implementation**

9 **C**

10 **\*\*TO DO\*\*.**

11 **CUDA**

12 **\*\*TO DO\*\*.**

## 1 **4.12 copyToTargetBoundary3D**

### 2 **4.12.1 Description**

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

### 5 **4.12.2 Syntax**

```
6 void copyToTargetBoundary3D(void* targetData, const void* data,  
7                             size_t extent3D[3], size_t nField,  
8                             size_t offset, size_t depth);
```

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

### 17 **4.12.3 Example**

18 **\*\*TO DO\*\*.**

### 19 **4.12.4 Implementation**

#### 20 **C**

21 **\*\*TO DO\*\*.**

#### 22 **CUDA**

23 **\*\*TO DO\*\*.**

## 1 **4.13 copyFromTargetBoundary3D**

### 2 **4.13.1 Description**

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

### 5 **4.13.2 Syntax**

```
6 void copyFromTargetBoundary3D(void* data, const void* targetData,  
7                               size_t extent3D[3], size_t nField,  
8                               size_t offset, size_t depth);
```

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

### 17 **4.13.3 Example**

18 **\*\*TO DO\*\*.**

### 19 **4.13.4 Implementation**

#### 20 **C**

21 **\*\*TO DO\*\*.**

#### 22 **CUDA**

23 **\*\*TO DO\*\*.**

## 1 **4.14 copyToTargetPointerMap3D**

### 2 **4.14.1 Description**

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

### 6 **4.14.2 Syntax**

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

- 10 • `targetData`: A pointer to the destination array on the target.
- 11 • `data`: A pointer to the source array on the host.
- 12 • `extent3D`: An array of 3 integers corresponding to the 3D dimensions of  
13 the lattice.
- 14 • `nField`: The number of fields per lattice site.
- 15 • `includeNeighbours`: A Boolean switch to specify whether each included  
16 site should also have its neighbours included (in the 19-point 3D stencil).
- 17 • `pointerArray`: An array of `nSite` pointers, where `nSite` is the total num-  
18 ber of lattice sites. Each lattice site should be included unless the pointer  
19 corresponding to that site is `NULL`.

### 20 **4.14.3 Example**

21 `**TO DO**`.

### 22 **4.14.4 Implementation**

#### 23 **C**

24 `**TO DO**`.

#### 25 **CUDA**

26 `**TO DO**`.



## 1 **4.15 copyFromTargetPointerMap3D**

### 2 **4.15.1 Description**

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

### 6 **4.15.2 Syntax**

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

- 10 • `data`: A pointer to the destination array on the host.
- 11 • `targetData`: A pointer to the source array on the target.
- 12 • `extent3D`: An array of 3 integers corresponding to the 3D dimensions of  
13 the lattice.
- 14 • `nField`: The number of fields per lattice site.
- 15 • `includeNeighbours`: A Boolean switch to specify whether each included  
16 site should also have its neighbours included (in the 19-point 3D stencil).
- 17 • `pointerArray`: An array of `nSite` pointers, where `nSite` is the total num-  
18 ber of lattice sites. Each lattice site should be included unless the pointer  
19 corresponding to that site is `NULL`.

### 20 **4.15.3 Example**

21 **\*\*TO DO\*\*.**

### 22 **4.15.4 Implementation**

#### 23 **C**

24 **\*\*TO DO\*\*.**

#### 25 **CUDA**

26 **\*\*TO DO\*\*.**

## <sup>1</sup> **Chapter 5**

## <sup>2</sup> **Data Parallel Execution**

## 1 **5.1 targetEntry**

### 2 **5.1.1 Description**

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

### 6 **5.1.2 Syntax**

7 `__targetEntry__` functionReturnType functionName(...

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

### 11 **5.1.3 Example**

12 See Line 1 in Figure 6.2 in Section 6.

### 13 **5.1.4 Implementation**

#### 14 **C**

15 Holds no value.

#### 16 **CUDA**

17 `__global__`

## 1 **5.2 target**

### 2 **5.2.1 Description**

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

### 6 **5.2.2 Syntax**

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

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

### 11 **5.2.3 Example**

12 Analogous to Line 1 in Figure 6.2 in Section 6.

### 13 **5.2.4 Implementation**

#### 14 **C**

15 Holds no value.

#### 16 **CUDA**

17 `__device__`

## 1 **5.3 targetHost**

### 2 **5.3.1 Description**

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

### 5 **5.3.2 Syntax**

6 `__targetHost__` functionReturnType functionName(...

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

### 10 **5.3.3 Example**

11 Analogous to Line 1 in Figure 6.2 in Section 6.

### 12 **5.3.4 Implementation**

#### 13 **C**

14 Holds no value.

#### 15 **CUDA**

16 `extern 'C' __host__`

## 1 **5.4 targetLaunch**

### 2 **5.4.1 Description**

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

### 5 **5.4.2 Syntax**

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

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

### 12 **5.4.3 Example**

13 See Line 6 in Figure 6.3 in Section 6.

### 14 **5.4.4 Implementation**

#### 15 **C**

16 Holds no value.

#### 17 **CUDA**

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

## 1 **5.5 targetSynchronize**

### 2 **5.5.1 Description**

3 The targetSynchronize function is used to block until the preceding \_\_targetLaunch\_\_  
4 has completed.

### 5 **5.5.2 Syntax**

6 `void targetSynchronize();`

### 7 **5.5.3 Example**

8 See Line 7 in Figure 6.3 in Section 6.

### 9 **5.5.4 Implementation**

#### 10 **C**

11 Dummy function.

#### 12 **CUDA**

13 `cudaThreadSynchronize`

## 1 5.6 targetTLP

### 2 5.6.1 Description

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

### 8 5.6.2 Syntax

```
9 __targetTLP__(int baseIndex, size_t extent)  
10 {  
11 //code to be executed in parallel  
12 }
```

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

### 17 5.6.3 Example

18 See Line 4 in Figure 6.2 in Section 6.

### 19 5.6.4 Implementation

#### 20 C

21 OpenMP parallel loop.

#### 22 CUDA

23 CUDA thread lookup.



## 1 **5.7 targetILP**

### 2 **5.7.1 Description**

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

### 8 **5.7.2 Syntax**

```
9 __targetILP__(int vecIndex)
10 {
11 //code to be executed in parallel
12 }
```

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

### 16 **5.7.3 Example**

17 See Line 9 in Figure 6.2 in Section 6.

### 18 **5.7.4 Implementation**

#### 19 **C**

20 Short vectorizable loop.

#### 21 **CUDA**

22 Short vectorizable loop.

## 1 **5.8 targetCoords3D**

### 2 **5.8.1 Description**

3 The targetCoords3D function provides the 3D lattice coordinates correspond-  
4 ing to a specified linear index.

### 5 **5.8.2 Syntax**

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

- 7 • coords3D (output): an array of 3 integers to be populated with the 3D  
8 coordinates.
- 9 • extent3D (input): An array of 3 integers corresponding to the 3D dimen-  
10 sions of the lattice.
- 11 • index (input): the linear index.

### 12 **5.8.3 Example**

13 **\*\*TO DO\*\***

### 14 **5.8.4 Implementation**

15 **C**

16 **\*\*TO DO\*\***

17 **CUDA**

18 **\*\*TO DO\*\***

## 1 **5.9 targetIndex3D**

### 2 **5.9.1 Description**

3 The targetIndex3D function returns the linear index corresponding to a speci-  
4 fied set of 3D lattice coordinates.

### 5 **5.9.2 Syntax**

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

- 7     • Xcoord (input): the specified coordinate in the X direction.
- 8     • Ycoord (input): the specified coordinate in the Y direction.
- 9     • Zcoord (input): the specified coordinate in the Z direction.
- 10    • extent3D (input): an array of 3 integers corresponding to the 3D dimen-  
11      sions of the lattice.

### 12 **5.9.3 Example**

13 `**TO DO**`

### 14 **5.9.4 Implementation**

#### 15 **C**

16 `**TO DO**`

#### 17 **CUDA**

18 `**TO DO**`

## Chapter 6

### Example

Consider a simple example: the scaling of a 3-vector field by a constant, as implemented in a sequential programming style in Figure 6.1. On each lattice site exists a 3-vector (a collection of three values corresponding to the 3 spatial dimensions). The outer loop corresponds to lattice sites, and the inner loop to the 3 components within each lattice site. This is a simple example of operations on “multi-valued” data, a very common situation in scientific simulations.

The lattice-based parallelism corresponding to the outer loop can be mapped to data parallel hardware using targetDP. We introduce targetDP by replacing the sequential code with the function given in Figure 6.2. The `t_` syntax is used to identify target data structures. The `__targetEntry__` syntax is used to specify that this function is to be executed on the target, and it will be called from host code. We expose the lattice-based parallelism to each of the TLP and ILP levels of hardware parallelism through use of the combination `__targetTLP__(baseIndex,N)` and `__targetILP__(vecIndex)` (See Sections 5.6 and 5.7). The former specifies that lattice-based parallelism should be mapped to TLP, where each thread operates on a chunk of lattice sites. The latter specifies that the sites within each chunk should be mapped to ILP. It can be seen that the `t_field` array is accessed by combining these indexes. The size of the chunk can be set within the targetDP implementation, to give the best performance for a particular architecture.

The `scale` function is called from host code as shown in Figure 6.3. The memory management facilities are used to allocate and transfer data to and from the target, as described in Chapter 4.

---

```

for (idx = 0; idx < N; idx++) { //loop over lattice sites
    int iDim;
    for (iDim = 0; iDim < 3; iDim++)
        field[iDim*N+idx] = a*field[iDim*N+idx];
}

```

Figure 6.1: A sequential implementation of the scalar multiplication of each element of a lattice data structure.

```

__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 6.2: The targetDP implementation of the scalar multiplication kernel.

```

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 6.3: The host code used to invoke the targetDP scalar multiplication kernel.