targetDP Specification
Version 0.1.0

Alan Gray and Kevin Stratford

January 15, 2016

Contents

Chapter 1

Introduction

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 perspective, it is not only imperative to achieve performance, but also to retain maintainability, sustainability and portability. The use of a simplistic, well structured and clearly defined abstraction layer such as targetDP can allow the programmer to express the scientific problems in a way that will automatically achieve good performance across the range of leading hardware solutions.

The targetDP API (first introduced in [1]) 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.

The targetDP memory and execution models are described in Chapters 2 and 3 respectively, and the document goers on to specify the memory management functionality in Chapter 4 and data parallel execution functionality in Chapter 5. Implementation details are also provided for the existing C and CUDA versions of targetDP throughout these chapters. Finally, a simple example is given to demonstrate usage in Chapter 6.

Glossary

- `CPU`: Central Processing Unit. The main computer chip used in a system, suitable for a wide variety of computational tasks.
- `Accelerator`: A processing unit which is not used in isolation, but instead in tandem with the CPU, with the aim of improving the performance of key code sections.
- `GPU`: Graphics Processing Unit. A type of accelerator, originally evolved to render graphics content (particularly to satisfy demands of the gaming industry), but now widely used for general purpose computation.
- `Host`: Another term for the CPU that "hosts" the application.
- `Data Parallel`: The type of algorithmic parallelism involved where a single operation is performed to each element of a data set. The extent of parallelism is determined by the size of the data set.
- `Target`: The device targeted for execution of data parallel operations. Depending on the underlying hardware available, the target could simply be the a CPU, or it could be a separate device such as an accelerator.
- `CUDA`: Compute Unified Device Architecture. The parallel platform and model created by NVIDIA to allow general purpose programming of their GPU architectures.
- `TLP`: Thread Level Parallelism.
- `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” refers 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). Each thread in the team will have a unique index. For some architectures, it is important to expose not just thread-level parallelism (TLP), but also instruction-level parallelism (ILP). The targetDP model facilitates this by allowing striding at the thread level, such that each thread can operate on a chunk of data (e.g. multiple grid points) at the instruction level. The size of the chunk, which we call the “Virtual Vector Length” (VVL), can be tuned to the hardware.

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.

In applications, it is sometimes necessary to perform reductions, where multiple data values are combined in a certain way. For example, values existing on each grid point may be summed into a single total value. The targetDP model supports such operations in a simplistic way. It is the responsibility of the application to create the array of values (using standard targetDP functionality) to act as the input to the reduction operation. The application can then pass this array to the API function corresponding to the desired reduction operation (e.g. `targetDoubleSum` for the summation of an array of double-precision Values). If the required reduction operation does not yet exist, the user can simply extend 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 space accessed by the host and that accessed by the target. Even although there is a trend towards "unified" address spaces, on which this distinction is not strictly required for applications to run successfully, such visibility at the application level is often crucial to allow good performance when running on those target architectures that have associated high-bandwidth memory systems (such as GPU and Intel Xeon Phi architectures). In the targetDP model, it is assumed 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 data structures 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.

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

This chapter specifices the memory management functionality in targetDP.

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 6.3 in Section 6.

4.1.4 Implementation

C

`malloc`

CUDA

`cudaMalloc`

4.2 targetCalloc

4.2.1 Description

The `targetCalloc` function allocates, and initialises 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 6.3 in Section 6.

4.2.4 Implementation

C

`calloc`

CUDA

`cudaMalloc` followed by `cudaMemset`

4.3 targetMallocUnified

4.3.1 Description

The `targetMallocUnified` function allocates unified memory that can be accessed on the host or the target. This should be used with caution since it may result in poor performance relative to use of `targetMalloc`.

4.3.2 Syntax

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

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

4.3.3 Example

Analogous to Line 1 in Figure 6.3 in Section 6.

4.3.4 Implementation

C

```
malloc
```

CUDA

```
cudaMallocManaged
```

4.4 targetCallocUnified

4.4.1 Description

The `targetCallocUnified` function allocates, and initialises to zero, unified memory that can be accessed on the host or the target. This should be used with caution since it may result in poor performance relative to use of `targetCalloc`.

4.4.2 Syntax

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

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

4.4.3 Example

Analogous to Line 1 in Figure 6.3 in Section 6.

4.4.4 Implementation

C

`calloc`

CUDA

`cudaMallocManaged` followed by `cudaMemset`

4.5 targetFree

4.5.1 Description

The `targetFree` function deallocates memory on the target.

4.5.2 Syntax

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

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

4.5.3 Example

See Line 11 in Figure 6.3 in Section 6.

4.5.4 Implementation

C

`free`

CUDA

`cudaFree`

4.6 copyToTarget

4.6.1 Description

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

4.6.2 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.

4.6.3 Example

See Line 3 in Figure 6.3 in Section 6.

4.6.4 Implementation

C

`memcpy`

CUDA

`cudaMemcpy`

4.7 copyFromTarget

4.7.1 Description

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

4.7.2 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.

4.7.3 Example

See Line 9 in Figure 6.3 in Section 6.

4.7.4 Implementation

C

`memcpy`

CUDA

`cudaMemcpy`

4.8 copyDeepDoubleArrayToTarget

4.8.1 Description

The `copyDeepDoubleArrayToTarget` function copies an array of double precision values from the host to the target, where the array is contained within another object.

4.8.2 Syntax

```
void copyDeepDoubleArrayToTarget(void* targetObjectAddress,
    void* hostObjectAddress,void* hostComponentAddress,size_t n);
```

- `targetObjectAddress`: A pointer to the target copy of the object that contains the data array.
- `hostObjectAddress`: A pointer to the host copy of the object that contains the data array.
- `hostComponentAddress`: A pointer to the host copy of the start of the array contained within the object.
- `n`: The number of elements to be copied.

4.8.3 Implementation

C

Pointer arithmetic to determine memory locations, followed by `memcpy`

CUDA

Pointer arithmetic to determine memory locations, followed by `cudaMemcpy`

4.9 copyDeepDoubleArrayFromTarget

4.9.1 Description

The `copyDeepDoubleArrayTFromTarget` function copies an array of double precision values from the target to the host, where the array is contained within another object.

4.9.2 Syntax

```
void copyDeepDoubleArrayFromTarget(void* hostObjectAddress,
   void* targetObjectAddress,void* hostComponentAddress,size_t n);
```

- `hostObjectAddress`: A pointer to the host copy of the object that contains the data array.
- `targetObjectAddress`: A pointer to the target copy of the object that contains the data array.
- `hostComponentAddress`: A pointer to the host copy of the start of the array contained within the object.
- `n`: The number of elements to be copied.

4.9.3 Implementation

C

Pointer arithmetic to determine memory locations, followed by `memcpy`

CUDA

Pointer arithmetic to determine memory locations, followed by `cudaMemcpy`

4.10 targetZero

4.10.1 Description

The `targetZero` function sets a (double precision) array on the target to zero.

4.10.2 Syntax

```
void targetZero(double* targetData, size_t n);
```

- `targetData`: A pointer to the array on the target.
- `n`: The number of elements in the array.

4.10.3 Implementation

C

A loop to zero each element.

CUDA

A kernel to zero each element.

4.11 targetSetConstant

4.11.1 Description

The `targetSetConstant` function sets each element of a (double precision) array on the target to the specified constant value.

4.11.2 Syntax

```
void targetSetConstant(double* targetData, double value, size_t n);
```

- `targetData`: A pointer to the array on the target.
- `value`: The value.
- `n`: The number of elements in the array.

4.11.3 Implementation

C

A loop to set each element.

CUDA

A kernel to set each element.

4.12 targetConst

4.12.1 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.

4.12.2 Syntax

```
__targetConst__ type variableName
```

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

4.12.3 Example

See Line 3 in Figure 6.3 in Section 6.

4.12.4 Implementation

C

Holds no value

CUDA

```
__constant__
```

4.13 copyConstToTarget

4.13.1 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.

4.13.2 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.

4.13.3 Example

See Line 4 in Figure 6.3 in Section 6.

4.13.4 Implementation

C

`memcpy`

CUDA

`cudaMemcpyToSymbol`

4.14 copyConstFromTarget

4.14.1 Description

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

4.14.2 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.

4.14.3 Example

Analogous to Line 4 in Figure 6.3 in Section 6.

4.14.4 Implementation

C

`memcpy`

CUDA

`cudaMemcpyFromSymbol`

4.15 targetConstAddress

4.15.1 Description

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

4.15.2 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.

4.15.3 Implementation

C

Explicit copying of address.

CUDA

`cudaGetSymbolAddress`

4.16 targetInit3D

4.16.1 Description

The `targetInit3D` initialises the environment required to perform any of the "3D" operations described in the rest of this chapter.

4.16.2 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).

4.17 targetFinalize3D

4.17.1 Description

The `targetFinalize3D` finalises the targetDP 3D environment.

4.17.2 Syntax

```
void targetFinalize3D();
```

4.18 copyToTargetPointerMap3D

4.18.1 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.

4.18.2 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`.

4.19 copyFromTargetPointerMap3D

4.19.1 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.

4.19.2 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`.

Chapter 5

Data Parallel Execution

This chapter specifices the data parallel execution functionality in targetDP.

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 5 in Figure 6.2 in Section 6.

5.1.4 Implementation

C

Holds no value.

CUDA

`__global__`

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 5 in Figure 6.2 in Section 6.

5.2.4 Implementation

C

Holds no value.

CUDA

`__device__`

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 5 in Figure 6.2 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 6.3 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 preceding `__targetLaunch__` has completed.

5.5.2 Syntax

```
void targetSynchronize();
```

5.5.3 Example

See Line 7 in Figure 6.3 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 Parallelism (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 8 in Figure 6.2 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 the ILP is defined by the virtual vector length (VVL) in the targetDP implementation (see 2).

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 13 in Figure 6.2 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 coordinates 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 the 3D coordinates.
- `extent3D` (input): An array of 3 integers corresponding to the 3D dimensions of the lattice.
- `index` (input): the linear index.

5.9 targetIndex3D

5.9.1 Description

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

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.

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.

```

//declare constant variable
__targetConst__ double t_a;

__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.

Bibliography

[1] Alan Gray and Kevin Stratford, *targetDP: an Abstraction of Lattice Based Parallelism with Portable Performance*, Proceedings of 2014 IEEE International Conference on High Performance Computing and Communications (HPCC), 312-315 (2014), arxiv.org/abs/1405.6162