A CUDA Kernel Scheduler Exploiting Static Data Dependencies

Eva Burrows
Bergen Language Design Laboratory
Department of Informatics, University of Bergen, Norway
http://bldl.ii.uib.no

***Abstract*—The CUDA execution model of Nvidia's GPUs is based on the asynchronous execution of thread blocks, where each thread executes the same kernel in a data-parallel fashion. When threads in different thread blocks need to synchronise and communicate, the whole computation launched onto the GPU needs to be stopped and re-invoked in order to facilitate inter-block synchronisations and communication.**

The need for synchronisation is tightly connected with the underlying data dependency pattern of the computation. For a good range of algorithms, the underlying data dependency pattern is static, scalable and shows some regularity. For instance, sorting networks, the Fast Fourier Transform, stencil computations of PDE solvers are such examples, but parallel design patterns like scan, reduce, and alike can also be considered. In such cases, much of the effort of devising and scheduling CUDA kernels for the computation can be automatized by exposing the dataflow representation of the computation in the program code using a dedicated API.

We present a methodology to build a generic kernel scheduler and related kernel parameterised by this API. A computation formalised in the terms of this API then serves as the entry point to these generic computational mechanisms, leading to direct CUDA implementations.

I. Introduction

Over the last decade, Graphics Processing Units (GPUs), originally developed to accelerate real-time graphics of game applications, have matured into very powerful computing devices. Non-graphics oriented communities soon realised the huge potential of GPU computing power for general purpose computations as well. This prompted graphics vendors to search for higher level programming models that would enable users to program GPUs without the need of wrapping their code in graphics specific terminologies like texture, vertex, framebuffer, shading, rendering, and alike. After several initiatives, for instance ATI's CloseToMetal [1], Stanford University's BrookGPU [2], or RapidMind's Sh [3], Nvidia's CUDA [4] programming model set a lasting example after its first appearance in late 2006 [5]. CUDA also influenced the first release of OpenCL in 2009 [6], which is now supported by many hardware vendors as an emerging language extension-based open standard to program heterogeneous systems.

In this paper, we focus on the CUDA execution model of Nvidia's GPUs which is based on a single-program multiple-data (SPMD) programming style. In CUDA terminology, a single-program is called a *kernel*, and a kernel is executed by *thread blocks*. Candidate computations to be executed on a GPU are in general data-parallel and compute intensive. The more computation-focused the kernel is, without the need to synchronise threads, the better. Threads, however, need to communicate and synchronise unless the computation is embarrassingly parallel. Given the execution model of CUDA based on the asynchronous execution of thread blocks, when threads in different thread blocks need to synchronise, the whole computation launched onto the GPU needs to be stopped and re-invoked in order to facilitate inter-block communication.

The need for synchronisation is tightly connected with the underlying data dependency pattern of the computation. After all, it is the flow of data and the presence or lack of dependencies between computational steps which determine any parallel execution. Looking at a computation as a whole, it is often difficult to handcraft the exact time-steps when synchronisation is needed. For instance, the bitonic sort is a divide and conquer algorithm, not readily portable to GPUs. When devising a data-parallel GPU execution for the bitonic sort, we need to rigorously consider the underlying sorting network determined by the algorithm. Based on that, we define the kernel and determine when intra-block and inter-block synchronisations are needed amongst the threads. The latter requires the kernel to be re-invoked. The number of kernel invocations also depends on the size of the thread blocks.

For a good range of algorithms much of this effort can be automatised by exploiting the underlying data dependency pattern of the computations. These are primarily computations based on static, scalable data dependencies where the patterns show some regularity. For instance, sorting networks, the Fast Fourier Transform (FFT), stencil computations of PDE solvers are such examples, but parallel design patterns like scan, reduce, and alike can also be considered.

The proposed methodology is based on a programming style in which data dependencies are defined in the source code as implementations of a specific Application Programming Interface (API), the *Data Dependency Algebra* (DDA) specification [7]. The data dependency pattern can be referenced trough its API allowing computations to be defined explicitly in terms of their local data dependencies. The DDA abstraction in the program code naturally entails code generation based on dataflow principles. This has been presented for various back-ends, from shared- and distributed-memory model computers, via GPUs to Field Programmable Gate Arrays (FPGAs) [8], [9], [10].

Further exploiting the use of DDA API, the contribution of this paper consists of the presentation of:

- a *kernel scheduler* which automatically schedules CUDA kernels, and
- a data-dependency driven *kernel body*.

The paper is organised as follows. The next section briefly describes the CUDA programming model. In Section III, we show how to define a computation to explicitly reference its local data dependencies in the code, illustrating this on the Fast Fourier Transform as a canonical example. In Sections IV and V, we present the kernel scheduler and the kernel body, respectively, based on the DDA API. Section VI discusses the performance model. Finally, in section VII, we present some related work, before we conclude in section VIII.

II. The CUDA Programming Model

In CUDA [4], the GPU operates as a Co-processor to the main CPU. The GPU is capable of handling a huge number of parallel threads each executing the same kernel. The total number of CUDA threads that execute a kernel is configured by the main program running on the host when the kernel is launched onto the device. The threads are organised in *grid of thread blocks*. A thread block contains a limited number of threads, but a grid of blocks may contain any (reasonable) number of blocks. The actual figures are device dependent. Newer GPUs are as a general rule capable of handling an increased number of threads due to the gradual developments in the device architecture.

Each thread executing the kernel is automatically assigned a unique thread and block ID accessible through built-in variables. Through these, threads can access/write specific segments of the GPU memory.

The cores on the GPU device are organised in multiprocessors (MP). Each thread block is run on one and only one MP. Threads within one block therefore can synchronize and share data through fast on-chip shared memory. The CUDA runtime is responsible for scheduling the thread blocks for parallel execution on the multiprocessors. There is no guarantee as to which blocks run physically in parallel at a time, or in which order blocks are sequenced. Therefore, threads in different blocks can only communicate asynchronously via the main GPU memory, and in practice are unable to exchange information within the same kernel. However, since the GPU memory is persistent across kernels, inter-block communications can always be attained by ending and re-invoking the kernel. The GPU memory is also a means for initialising computations and obtaining results.

We can abstract over this execution model by introducing a *CUDA Space-Time API*. We denote by `T` and `B` the number of threads per block, and the number of blocks that execute the kernel, respectively. The type `CudaT` presents us with a *space-time thread*, in which a `thread` is considered at a given `time`-step along the kernel execution.

```
struct CudaT {
 ThreadT thread;
 unsigned int time;
 };

struct ThreadT {
 unsigned int blockIdx;
 unsigned int threadIdx;
 };

bool DI (ThreadT t) {
 return ((t.blockIdx < B) && (t.threadIdx < T)); }
```

In order to abstract over when space-time threads can synchronise within a block or outside block boundaries, we can use conveniently the thread type `ThreadT` to identify ingoing and outgoing communication channels between space-time threads along consecutive time-steps:

```
CudaT in(CudaT t, ThreadT ch) {
 return {ch, t.time-1};
 }

CudaT out(CudaT t, ThreadT ch) {
 return {ch, t.time+1};
 }
```

Whenever

```
in(t, ch).thread.blockIdx == t.thread.blockIdx;
```

the ingoing communication channel at space-time thread `t` from space-time thread `ch` in the previous time-step is within block boundaries. Otherwise it crosses block boundaries, indicating the need for a new kernel invocation.

Likewise, whenever

```
out(t, ch).thread.blockIdx == t.thread.blockIdx;
```

the outgoing communication channel at space-time thread `t` to space-time thread `ch` in the next time-step is within block boundaries, otherwise crosses block boundaries.

III. Programming with Static Data Dependencies

In this section, we present the DDA API as a means of data dependency based programming [11]. The Fast Fourier Transform (FFT), being one of the most important numerical algorithms of many engineering applications, will serve as an illustration. Central to this approach is that the underlying data dependency graph of a computation is defined in terms of a specific API rather than a graph data structure stored in the memory.

The DDA API specification consists of two generic types: `Point` to define the graph points, and `Branch` to differentiate between all producer and all consumer arcs at each point using branch indices. In addition, the API specifies two sets of function declarations on these types. These are to define point-to-point dependencies in both the *request* (consumer) and *supply* (producer) directions along a dependency arc. The request function `Point rp(Point,Branch)` is to be called only when the guard `bool rg(Point,Branch)` holds. The guard `bool rg(Point,Branch)` determines which branch indices are in particular used at a point to differentiate amongst all the request directions at the point. The supply component of the API defines the opposite direction, being dual of the request.

The underlying data dependency graph of the Radix-2 FFT is the butterfly dependency pattern, see Fig. 1. Using the

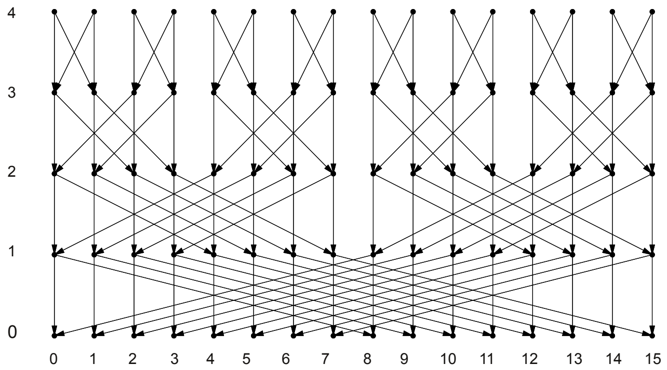

Fig. 1. FFT data dependency for h=5. The direction of the arrows corresponds to the request (consumer) directions. The FFT progresses bottom-up along this data dependency graph.

DDA API specification, we define the butterfly pattern by implementing all components of the API as follows:

```
1  unsigned int h;
2
3  typedef unsigned int Branch;
4
5  struct Point {
6    unsigned int row;
7    unsigned int col;
8    };
9
10 bool DI (Point p) {
11   return ((p.row<=h) && (p.col<2^h));
12   }
13
14 bool rg (Point p, Branch b) {
15  return ( ((b==0) || (b==1)) && (p.row>0) );
16  }
17
18 Point rp(Point p, Branch b) {
19   if (b==0) return {p.row-1, p.col};
20   else return {p.row-1, flip(p.row, p.col)};
21   }
22
23 bool sg (Point p, Branch b) {
24  return ( ((b==0) || (b==1)) && (p.row<h) );
25  }
26
27 Point sp(Point p, Branch b) {
28   if (b==0) return {p.row+1, p.col};
29   else return {p.row+1, flip(p.row+1, p.col)};
30   }
```

where `flip(i, m)` flips the `i`-th bit in the `h`-bit binary representation of `m`.

The variable `h` (line 1) defines the height of the graph which ultimately determines the size of the underlying grid. The shape of the grid is controlled by the data-invariant `DI` (line 10) which for a given `h` restricts the values of the type `Point` to the actual grid points. Branch directions are considered local at each point in the graph. In the FFT, `0` denotes all vertical directions, and `1` all directions across.

The computations performed by the FFT are defined in terms of the DDA API at each point. Let `v` be an array of complex numbers such that the elements of `v` are uniquely indexed by the type `Point`. First, `v` is initialised at the indices corresponding to the bottom row in the FFT. Explicitly utilising the local dependencies at each point given by `rp`, the rest of `v` can be computed by repeatedly calling the following function on points for which the local dependencies are met (we introduced the shortcuts `row = p.row; col = p.col`):

```
complex fft(Point p){
  return ((col < rp(p,1).col) ?
         v[rp(p,0)] + v[rp(p,1)]*w(rev(col>>h-row)):
         v[rp(p,1)] + v[rp(p,0)]*w(rev(col>>h-row)));
  }
```

where `rev(m)` reverses the bits-order in the `h`-bit binary representation of `m`, and `w(m)` denotes ω^m, where ω is the primitive 2^h-th root of unity.

The order in which `fft` is called to fill in the values of `v` is now a matter of how the dataflow information provided in the DDA is utilised.

In order to execute the FFT in CUDA, we need to embed the FFT DDA into the CUDA Space-Time API. This is facilitated by embedding FFT points to space-time CUDA threads, and data dependency arcs to communication channels between space-time threads.

```
CudaT em(Point p){
  ThreadT t = {idiv(p.col,T), mod(p.col,T)};
  return {t, p.row};
  }

ThreadT emR(Point p, Branch b){
  return em(rp(p,b)).thread;
  }

ThreadT emS(Point p, Branch b){
  return em(sp(p,b)).thread;
  }
```

We control the granularity of the embedding. We could assign the same space-time thread to a group of DDA points in a row. However, to keep our presentation simple, we assume that our embedding is injective, i.e., each DDA point is assigned uniquely a space-time thread. We discuss granularity issues in Section VI.

In general, the DDA concept yields dependency-driven computational mechanisms for various parallel hardware based on dataflow principles [9]. These are parameterised by:

- the *DDA API*, together with
- a *function* `ElementT f(Point)` which defines how the value at a DDA point of some type `ElementT` is to be computed utilising the DDA API (like the funciton `fft`), and
- an *embedding* of the DDA into the execution model of the hardware abstracted away as a Space-Time API.

For any algorithm expressed in this formalism, the computational mechanisms can be instantiated to yield the desired executable.

The embedding is only useful if it consistently maps request/supply directions of the DDA to incoming/outgoing communications channels in the hardware, across the whole DDA:

- `em(rp(p,b)) == in(em(p),emR(p,b))`
- `em(sp(p,b)) == out(em(p),emS(p,b))`

One can easily verify that the embeddings defined for the FFT above satisfy these requirements.

We are now ready to define the body of a kernel utilising the DDA API. Such a kernel, instantiated for the FFT, will be executed by 2^h number of threads gradually filling in the missing parts of a device version of `v`. Since in the general case threads require cross-block communication, we first build a kernel scheduler. This is directly related to the embedding.

IV. CUDA Kernel Scheduler

In general, the CUDA execution of a computation is based on repeated kernel invocations, unless the particular problem we are dealing with is embarrassingly parallel (no dependency exists between the parallel parts), or it is small enough to be executed by a single thread block. Kernels should only run as long as intra-block communication is guaranteed, otherwise they need to be ended and re-invoked. The exact time-steps for invoking and ending a kernel is an inherent property of the embedding of the DDA into the CUDA Space-Time and is in fact independent from the actual computations performed at a point. This property is made concrete next.

We denote by `tmax` the maximum time-step of the embeddings, i.e., `em(p).time<=tmax` for all DDA points. (In case of the FFT embedding, `tmax = h`.) The kernel scheduler will use a support array `K`, the *schedule-array*, which contains all time-steps when a kernel has to be ended. Algorithm 1 presents the pseudo-code for computing `K`.

```
1  i=0;
2  t=0;
3  while t<tmax do
4      onlyWithinBlock=True;
5      forall the DDA points s.t. em(p).time=t do
6          if for some b
7          (em(p).thread.blockIdx !=
           em(sp(p,b)).thread.blockIdx) then
8              onlyWithinBlock=False;
9          end
10     end
11     if onlyWithinBlock==False then
12         K[i]=t;
13         i++
14     end
15     t++;
16 end
17 K[i] = tmax;
```

Algorithm 1: Computing the schedule-array.

The algorithm computing `K` is parsing all DDA point embeddings. For all points embedded with the same time component (line 5), it is checked whether any of these have a supply direction leading to a DDA point embedded into a different thread block (line 7). If so, then we record this time-step in `K`, indicating that the kernel has to be ended here.

The *CUDA Kernel Scheduler* (Algorithm 2) is run on the host. It will systematically invoke CUDA kernels of dimension `B*T`, one at a time, based on the schedule-array `K`. The start and end time-steps of the kernel are specified int the kernel argument-list, fetched from `K`. The other parameters in the argument-list, `d` and `ig` are specified when discussing the kernel body.

The first kernel is invoked at time-step `0`, and runs until time-step `K[0]`. When a kernel is ended at some time-step `K[i]`, it is re-invoked at the next time-step `K[i]+1`, unless `K[i]=tmax`, and runs until time-step `K[i+1]`. If the length of `K` is `tmax+1`, the kernel is ended at every time-step and re-invoked at the next time-step. When the length of `K` is `1`, i.e., when `K[0]=tmax`, the kernel is invoked once.

```
1 kernel<<B,T>>(d,ig,0,K[0]);
2 if K[0]<tmax then
3     i=1;
4     while K[i]<=tmax do
5         kernel<<B,T>>(d,ig,K[i-1]+1,K[i]);
6         i++;
7     end
8 end
```

Algorithm 2: CUDA Kernel Scheduler.

Fig. 2 illustrates the conceptual overview of the kernel scheduler for FFT embedded into the CUDA Space-Time. The length of `K` depends on the embedding, and ultimately on `B` and `T`. Larger the `T`, less inter-block communication is involved.

A more elaborate kernel scheduler instead of pre-computing the entire array `K` would compute every next entry in the schedule array on-the-fly right after a kernel has just been invoked. This could speed up the accumulated total execution time, since CUDA allows concurrent execution between host and GPU device, i.e., control is returned to the host before the device has completed the requested task. On the other hand, building the schedule array can always be considered fixed and shipped with the embedding. It is independent of the computations performed at the DDA points or any input data.

V. Kernel Body

The kernel is invoked by the scheduler in lines 1 and 5. In previous work [9], we described a kernel that explicitly utilises the supply component of the DDA API, following dataflow principles [9]. Here, we present an alternative kernel which utilises the request component instead.

CUDA differentiates between host and device memory. The GPU can only write/access data stored on the GPU. The host manages the memory spaces visible to kernels (e.g. its arguments) through calls to the CUDA run-time. This includes device memory allocation and deallocation as well as data transfer between host and device memory. We will, therefore, skip presenting the fine details of this memory management, they are being specific to CUDA and straightforward in the context of a given kernel.

The kernel has 4 arguments: `(d, ig, start, end)`, where `d` is a device array which prior to the first kernel invocation is a copy of the host array `v`. Some elements of `v` (and hence of `d`) will be defined, these representing the *initial values* from which the computation start. (In the FFT, these will be the elements corresponding to the bottom row, but in order to maintain generality, we assume that the computation can be initialised anywhere in the DDA graph.) Since, from a computational point of view, `d` (and `v` also) are partially defined (not all values are computed yet), we assume a guard will keep track of that: `ig(i)=true` whenever `d[i]` is *defined*. Both `d` and its guard `ig` are device arrays and are updated on the GPU memory by the threads across all kernels. We consider the result of the whole computation to be the device array `d` together with its guard. The last two arguments, `start` and `end` will mark the time-steps for starting and terminating the kernel, respectively.

Let `f` be the function describing how the values of `v` are to be defined. The CUDA run-time requires any function callable on the GPU to be explicitly declared as a device function, preceded by the `__device` keyword. Therefore, we assume a device version of `f`, and a device version of the DDA API implementation, in our further discussion.

The *request-based kernel execution* will gradually fill the elements of the device array `d` as follows.

The device array is distributed in the GPU memory such that thread `em(p).thread` will compute `d[p]` at time-step `em(p).time`, using `f` instantiated for `d`.

All `thread` across `ThreadT` will execute in parallel the body of the kernel described in Algorithm 3.

```
1  t = thread.blockIdx * T + thread.threadIdx;
2  i = start;
3  while i<=end do
4      Point p={t, i};
5      if DI(p) then    /* p is a valid DDA point */
6          if !ig[p] && (ig[rp(p,b)] whenever rg(p,b))
           then  /* d[p] is ready to be computed */
7              d[p]=f(p);
8              ig[p]=True;
9          end
10     end
11     __syncthreads();
12     i++;
13 end
```

Algorithm 3: Kernel body utilising the request component of the DDA.

Proposition. *When the kernel scheduler of Algorithm 2 using the kernel body of Algorithm 3 finishes, the device array* `d` *will be filled with all values that can possibly be computed from its initial values.*

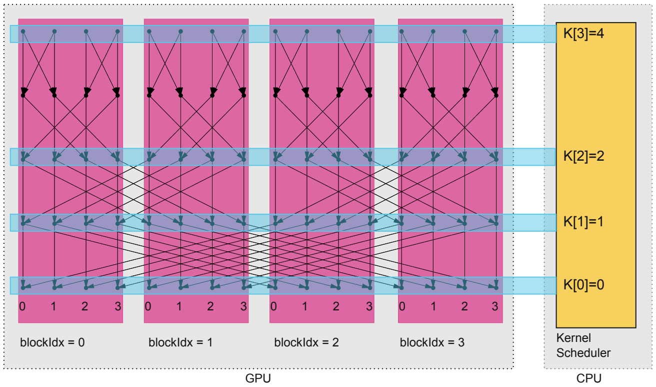

Fig. 2. FFT embedded into the CUDA Space-Time API with B=4 and T=4.

Proof. We assumed that in our embedding each DDA point is assigned to exactly one space-time thread. Therefore line 4 in Algorithm 3 ensures that (1) all DDA points will be served by a thread at some point in time, and (2) we can uniquely identify which DDA point the current space-time thread should deal with. As the embedding may not be surjective, we need the data invariant test in Line 5 to keep only those space-time threads busy that has in fact been assigned a DDA point.

At every time-step, each thread executing the kernel computes a new value in `d`, provided that the dependencies are met at the point (line 6). The explicit synchronization in line 11 works across all the threads of a block, and for all blocks individually (as allowed by the CUDA run-time). This and the way the kernel scheduler is built ensure race-condition free execution between threads of the same and of different blocks. So the test whether all relevant dependencies of `d[p]` have been computed, in line 6, is guaranteed to be correct. Even if one of the dependencies is across block boundaries and has not been computed by the time-step of the test, then it will not be computed at all. (A very plausible scenario for this is when `d` is "under"-initialised. Such undefined behaviour is then further propagated along the dependencies.)

When the last kernel ends, the result of the computation will be the entire device array `d`, with as many defined values, indicated by `ig`, as possible computed from the initial values.

□

The FFT DDA is very regular, with initial values at the bottom row, and with a computation proceeding in a data parallel fashion from one row to the next. Other computations may have less regular DDAs, and the computation might be initialised at arbitrary DDA points. Nonetheless, the presented computational mechanism, even for such less regular DDAs, will compute all values that can be possibly computed given the DDA and the initial values, in a controlled manner.

The kernel scheduler, suitably modified, may suspend the kernel invocations prior to reaching the kernel schedule entry `tmax`. To discuss this, we introduce `tspan`, the maximum number of time-steps spanned by a request: `em(p).time-em(rp(p,b)).time<=tspan` for any DDA point. This is a property of the embedding. (In case of the FFT embedding `tspan=1`.)

Using `tspan`, the scheduler can keep track of whether there has been computed any new value in `d` over the last `tspan` consecutive time-steps. If not, it checks whether all initial values have already been consumed. This can be determined by checking for all point `p` where `d[p]`, prior to the first kernel invocation, was initialised whether `em(p).time<K[i]-tspan` for the current kernel schedule entry `K[i]`. If so, then the kernel scheduler can safely suspend the computation, as nothing more can be computed from the initial values either.

The mechanism can deal with any computational DDA embedded into the CUDA Space-Time API. Depending on structural properties of the DDA, the mechanism may be fine-tuned for better performance. For instance, if `tspan=1`, and all input points are embedded to the time-step marking the start of the computation, a more efficient kernel could be devised. Some of these issues will be discussed in the next section.

VI. Performance Model

Based on our experiments, the presented dependency-driven computational mechanism raises several issues related to performance and resource utilisation that need to be addressed. We consider several alternatives in order to improve on these. A thorough profiling of these to actually gain insight into which technique suits best a certain setting is on the go.

A. Memory Usage

The computational model has control over storage space utilisation on the GPU. The size of the arrays `d` and `ig` is the same as the number of DDA points. Having this size makes sense in a computation where all values computed along the DDA are useful. In other computations, however, we might be interested only in the final result, and not so much in the intermediate values.

Considering the FFT, the *target points* are the ones corresponding to the top row of the graph, where we read the result of the computation:

```
bool target(Point p){
  return (p.row == h);
  }
```

The intermediate values computed up to row `h-1` can be casted once they have been used up as the computation proceeds. This can be controlled by `tspan`.

We first modify the kernel's argument list:

- introduce a new device array `t` dedicated to record the computed target values,
- instead of `d` we use a `window` array and its guard of size `T*B*(tspan+1)`, meant to act as a sliding window over DDA points, where `tspan+1` represents the height of the window.

Accordingly, the body of the kernel is modified to let a thread compute the value at point `p` (line 4) in terms of the data recorded in the `window` array. A DDA point `p={t,i}` corresponds to the point `pp={t, mod(i,tspan+1)}` in the `window` array. If `p` is a target value, then the computed value in `window` is recorded in `t` as well.

In CUDA, the host manages the main GPU memory, such as allocation, deallocation, data movement between host and device, and device and device. Hence the kernel scheduler is responsible to "slide" the window over the DDA, thereby casting the intermediate values that have been used. This comes at the additional cost of device memory management.

In addition, a new schedule-array will be required, since a kernel shouldn't run longer than `tspan` consecutive time-steps. In case of FFT, this would require the kernel to be re-invoked at every time-step in order to let the scheduler to advance the window. In other cases, several kernel invocations may take place across large `tspan` time-steps. Therefore we may choose to work with a sliding window of at least `tspan+1` high, but increase the height to balance out the overhead caused by the additional kernel invocations.

In summary, device memory usage can be decreased at the cost of increased workload on the kernel scheduler, increased device memory management, and by possibly increasing the number of kernel invocations.

B. Global vs. Shared Memory on the GPU

The CUDA memory model differentiate between high-latency device memory (managed by the host) and low-latency, on-chip shared memory available for threads within a thread block.

The threads executing our kernel continuously access slow GPU memory locations instead of fast on-chip memory.

When a kernel spans over 1 time-step, this is reasonable, since the results need to be recorded in the device memory to be ready for the next kernel. However, when the kernel spans over several time-steps, and we are not interested in keeping intermediate values, threads could do most of their computations using local shared memory within the thread block.

There is an efficient solution using this technique for DDAs with `tspan=1`, where initial values are provided at time-step 0, and target points are in the last time-step. The FFT DDA is such, but most algorithms that we are targeting here have this property: sorting networks, stencil computations, binary trees, etc. In this case, instead of employing a sliding window, we use two device arrays of size `T*B`: `In` and `Out`. The initial/intermediate values are provided in the array `In` at the beginning of the kernel, which in turn updates the array `Out` at the end of the kernel with the new intermediate/final values. When the kernel runs over several time-steps, i.e, `end-start > 1`, threads first read the elements of the array `In` into the shared memory, and continue with the computation described in the kernel body (line 3-12) instantiated on shared-memory. After the last time-step, threads update the array `Out`. The scheduler then swaps the addresses of the arrays `In` and `Out` so that the array `In` is updated with the new intermediate values in the next kernel, but without the need of expensive device-to-device copies. Due to fast memory accesses, this alternative gives better execution time while device memory usage is reduced to the minimum.

C. Granularity

We have seen that embeddings are programmable through the presented APIs. Here we discuss some basic principles of the embeddings which ultimately determine the granularity of the computation, affecting overall execution time.

1. The proposed Space-Time API that abstracts over CUDA, can be extended to abstract over a multi-GPU system. Assuming we have a system with `N` CUDA GPUs, we allow kernels to run on the GPUs with a non-uniform number of threads and blocks across the GPUs, specified by the arrays `T` and `B` of size `N`:

```
struct CudaT {
  GpuT thread;
  unsigned int time;
  };

struct GpuT {
  unsigned int gpuIdx;
  unsigned int blockIdx;
  unsigned int threadIdx;
  };

bool DI (GpuT t) {
 return ((t.gpuIdx<N) && (t.blockIdx < B[t.gpuIdx]) &&
         (t.threadIdx < T[t.gpuIdx])); }
```

Embedding a DDA into this execution model follows the same principle as the one presented earlier on which both the kernel scheduler and kernel were built. Here we are able to determine the exact time-step for inter-GPU communication as well as inter-block communication within a GPU.

Whenever:

```
em(p).thread.gpuIdx == em(sp(p,b)).thread.gpuIdx
```

`p` and `sp(p,b)` are embedded onto the same GPU, otherwise the scheduling need to facilitate a safe communication between them. This abstraction obviously calls for a new kernel scheduler which is now able to orchestrate kernels and threads over a multi-GPU system in a controlled manner. However, the principles are the same.

2. We can easily define different embeddings for the same DDA into the same hardware in a flexible way. According to the results of [12], well-chosen (injective) embeddings lead to less kernel invocations which is considered desirable in CUDA programming, and it is therefore worth considering in this context.

3. By coarsening the embedding, the computational throughput of a thread increases. This is achieved by embedding DDA points into the CUDA space-time API such that a space-time thread computes `N` values at a given time-step, instead of just one value.

```
CudaT em(Point p){
  unsigned int n = idiv(p.col, N);
  ThreadT t = {idiv(n, T), mod(n, T)};
  return {t, p.row};
  }
```

Such embedding obviously changes the schedule-array, leads to less kernel invocations, and requires the kernel body to be adapted to allow a space-time thread to compute multi-values of `d`.

VII. Related Work

Our focus on data dependency graphs makes as closely related to approaches exploiting dataflow principles. This framework calls for a micro dataflow representation of the computation, separating computations from their local data dependencies, and requires an abstract representation of the hardware. In this paper, we focused on CUDA, as an intermediate programming layer between the user and GPU, but other layers such as OpenCL should also suit the approach. In principle, any hardware that can be abstracted as a Space-Time API, e.g., distributed-, shared-memory systems, can serve as target platforms, offering a deterministic and data-race free computational mechanism for them.

The programming model is restricted to computations that can be represented as a static dataflow graph. This makes the framework less general than more advanced dataflow-focused approaches which can deal with dynamic graph representations and are more general programming models for heterogeneous platforms, such as CnC [13], [14], the Codelet Model [15], Liquid metal [16] and Lime [17], Flexstream [18], OmpSs [19], and STAPL [20], just to mention a few representatives.

CnC's separation of concerns based on graphs describing the dependencies between serial pieces of code, in particular, shows familiarity to our framework. CnC is a general macro dataflow model, wheres the DDA-approach is based on a fine-grained dataflow representation. STAPL's skeleton framework allows dataflow graphs to be built using parametric dependencies propagated into a global dependency pattern. Our focus on a dedicated API to represent data dependency graphs also allows the building of larger graphs by the means of combinators applied on DDA implementations. Currently, we are investigating this technique to enhance the programming model, enabling it to deal with dynamic dataflow graphs as well.

VIII. Conclusion

This paper presents a general methodology to automatically port computations with static scalable data dependencies to the CUDA programming model of NVIDIA GPUs. This is formalised around a fine-grained dataflow-based representation of the computation using a dedicated API. The methodology describes how to build a kernel scheduler and a related kernel parameterised by this API.

A thorough profiling embracing a large set of examples for the proposed computational mechanism and its alternatives is on the go. The initial results show that there is a distinct overhead caused by the additional machinery that drives the computation via calls to the API and related embeddings when compared to fine-tuned hand-coded CUDA versions. However, the early experiments also show that the computational mechanism scales with the number of threads, making the framework a promising approach to generate for a good range of computations scalable CUDA code.

Acknowledgment

The author would like to thank Magne Haveraaen for his valuable feedback on this work.

This is a research supported by The Research Council of Norway through the project DMPL – Design of a Mouldable Programming Language.

References

[1] AMD, "ATI CTM (close to metal) guide," AMP Press Release, Technical Reference Manual, 2006.

[2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, Aug. 2004. [Online]. Available: http://doi.acm.org/10.1145/1015706.1015800

[3] Sh: A high-level metaprogramming language for modern GPUs. [Online]. Available: http://libsh.org/

[4] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[5] NVIDIA Unveils CUDA. the gpu computing revolution begins. [Online]. Available: http://www.nvidia.com/object/IO_37226.html

[6] "The OpenCL Specification Ver. 1.0," Khronos OpenCL Working Group, Manual, 2009. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf

[7] V. Čyras and M. Haveraaen, "Modular programming of recurrencies: a comparison of two approaches," *Informatica*, vol. 6, no. 4, pp. 397–444, 1995. [Online]. Available: http://www.ii.uib.no/saga/papers/CyrasHaveraaen1995informatica-scanjob.pdf

[8] S. Søreide, "Compiling sapphire into sequential and parallel code using assertions," Master thesis, Department of Informatics, University of Bergen, Norway, P.O. Box 7800, N-5020 Bergen, Norway, 1998.

[9] E. Burrows and M. Haveraaen, "A hardware independent parallel programming model," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 7, pp. 519 – 538, 2009, the 19th Nordic Workshop on Programming Theory (NWPT 2007). [Online]. Available: http://dx.doi.org/10.1016/j.jlap.2009.06.002

[10] E. Burrows, "Compiling a dataflow-based language abstraction onto an FPGA," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Proceedings of the International Conference on Parallel Computing, ParCo 2013, 10-13 September 2013, Garching (near Munich), Germany*, 2014, pp. 507–514. [Online]. Available: http://dx.doi.org/10.3233/978-1-61499-381-0-507

[11] A. Anderlik and M. Haveraaen, "On the category of data dependency algebras and embeddings," *Proceedings of the Estonian Academy of Sciences, Physics, Mathematics*, vol. 52, no. 4, pp. 337–355, 2003. [Online]. Available: https://books.google.no/books?id=w01eEXQ94WMC&lpg=PA331&pg=PA337

[12] E. Burrows and M. Haveraaen, "Programmable data dependencies and placements," in *Proceedings of the 7th workshop on Declarative aspects and applications of multicore programming*, ser. DAMP '12. New York, NY, USA: ACM, 2012, pp. 31–40. [Online]. Available: http://doi.acm.org/10.1145/2103736.2103741

[13] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar, "Mapping a data-flow programming model onto heterogeneous platforms," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, ser. LCTES '12. New York, NY, USA: ACM, 2012, pp. 61–70. [Online]. Available: http://doi.acm.org/10.1145/2248418.2248428

[14] M. Grossman, A. S. Sbîrlea, Z. Budimlić, and V. Sarkar, "CnC-CUDA: Declarative programming for GPUs," in *Proceedings of the 23rd international conference on Languages and compilers for parallel computing*, ser. LCPC'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 230–245. [Online]. Available: http://portal.acm.org/citation.cfm?id=1964536.1964552

[15] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a ”codelet” program execution model for exascale machines: Position paper," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, ser. EXADAPT '11. New York, NY, USA: ACM, 2011, pp. 64–69. [Online]. Available: http://doi.acm.org/10.1145/2000417.2000424

[16] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah, "Liquid metal: Object-oriented programming across the hardware/software boundary," in *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ser. ECOOP '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 76–103. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-70592-5_5

[17] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: A java-compatible and synthesizable language for heterogeneous architectures," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 89–108. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869469

[18] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke, "Flextream: Adaptive compilation of streaming applications for heterogeneous architectures," in *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 214–223. [Online]. Available: http://dx.doi.org/10.1109/PACT.2009.39

[19] V. K. Elangovan, R. M. Badia, and E. Ayguadé, "Scalability and parallel execution of ompss-opencl tasks on heterogeneous cpu-gpu environment," in *Proceedings of the 29th International Conference on Supercomputing - Volume 8488*, ser. ISC 2014. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 141–155. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-07518-1_9

[20] M. Zandifar, M. Abdul Jabbar, A. Majidi, D. Keyes, N. M. Amato, and L. Rauchwerger, "Composing algorithmic skeletons to express high-performance scientific applications," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 415–424. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751241