Dependency-driven Parallel Programming*

Eva Burrows † Magne Haveraaen ‡

Department of Informatics, University of Bergen, Norway

October 21, 2009

Abstract

With the appearance of low-cost, highly parallel hardware architectures, software portability between such architectures is in great demand. Software design lacks programming models to keep up with the continually increasing parallelism of today's hardware. This setting calls for alternative thinking in programming. When a computation has a static data-dependency pattern, extracting this pattern as a separate entity in a programming language, one can reformulate the computations. As a consequence, data-dependencies become active participants in the problem solving code. This allows us to deal with parallelism at a high-level. Data-dependency abstractions facilitate the mapping of computations to different hardware architecture without the need of rewriting the problem solving code. This in turn addresses portability and reusability issues.

1 Introduction

Computational devices are rapidly evolving into massively parallel systems. Multi-core processors are standard, and high performance processors such as the Cell processor [3] and graphics processing units (GPUs) featuring hundreds of on-chip processors, e.g. [8], are all developed to accumulate processing power. They make parallelism commonplace, not only the privilege of expensive high-end platforms. However, current parallel programming paradigms cannot readily exploit these highly parallel systems. In addition, each hardware architecture comes along with a new programming model and/or application programming interface (API). This makes the writing of portable, efficient parallel code difficult. As the number of processors per chip is expected to double every year over the next few years, entering parallel processing into the mass market, software needs to be parallelized and ported in an efficient way to massively parallel, possibly heterogeneous, architectures. Certain steps have been made towards standardization, e.g. OpenCL (Open Computing Language) has recently been announced as a framework for writing programs across heterogeneous platforms, and many hardware vendors seem to support it. However, the programming community is still in great need of high-level

*This paper recollects some of the theoretical results presented in an earlier work [2], by describing the underlying concepts at a higher level. In addition it also reports on the outcome of recent practical experiments.

†http://www.ii.uib.no/~eva/

‡http://www.ii.uib.no/~magne/

This paper was presented at the NIK-2009 conference; see http://www.nik.no/.

parallel programming models to adapt to the new era of commonly available parallel computing devices.

1.1 Dependency-driven Thinking

Miranker and Winkler [7] suggested that program data dependency graphs can abstract how parts of a computation depends on data supplied by other parts. This is a basis for parallelizing compilers, see e.g. [12]. In our formalism these graphs are captured by algebraic abstractions – Data Dependency Algebras (DDAs) – and turned into first-class citizens in program code. This allows us to formulate the computation as expressions over consecutive computational points of the dependency pattern, such that dependencies between computational steps (DDA points) become explicit entities in the expression itself.

A parallel hardware architecture's space-time communication layout, its API, can also be captured by special space-time dependency patterns – Space-Time Algebras (STAs). This is obtained by projecting the static spatial connectivity pattern of the hardware over time. Mapping a computation to an available hardware resource then becomes a task of finding an embedding of the computation's DDA into the STA of the hardware [2]. This can be defined and easily modified at a high-level using DDA-embeddings, and a DDA-based compiler then can generate the executable for the required target machine.

Though the background theory of Data Dependency Algebras and their use for parallel computing was first established more than a decade ago [6], we believe that the potential of dependency-driven thinking constitutes a valuable field for parallel computing research, in particular today, when computing is irreversibly going parallel.

1.2 Contribution and Overview of the Paper

This paper revisits the DDA-approach, and recollects some of our results from the past year [2]. Its main focus is to present the underlying concepts at a higher level. In addition, it emphasizes that DDAs can abstract spatial placements of computations by showing new results using our small visualization tool. It points out how this setting also can serve as an aid when adapting to highly parallel hardware architectures, e.g., a GPU. And it discusses the viability of dependency-driven programming by showing experimental results.

The next section contains a gentle introduction to DDAs. Section 3 illustrates their nature through some examples by extracting dependency patterns of well-known computations. It also argues why DDA abstractions can serve as a powerful means when computations need to be mapped to different hardware architectures. Section 4 reports on recent practical experiments based on DDA-based programming. Finally, Section 5 gives a quick overview on the status of the implementation, before the paper concludes in the last section.

2 Data Dependency Algebras

We will not give a detailed formal presentation of the mathematical background theory, and advice the reader that formal definitions of DDAs and of STAs can be found in [6, 2]. However, the notational conventions and program codes we use in this paper are similar to [2]. We only recall the necessary minimum which can serve as a basis for our discussions.

The concept of data dependency algebra is equivalent to that of a directed multigraph. In our formalism, we talk about points defined by `P`, branch indices defined by `B`, and requests and supplies. At each point (nodes of the graph), request branches (arcs) lead to

neighbours that the point requires data from, whereas supply branches (the opposite arcs) lead to neighbours that are supplied with data from the point.

An arc between two points of the graph stands for a request-supply connection. Branch indices from $\mathtt{B}$ are local at each end of the branch: one identifies the request and one the supply direction. Obviously we want to keep $\mathtt{B}$ as small as possible, but different points may have a varying number of request and supply arcs, motivating the need for the following two relations $\mathtt{r_g} \subseteq \mathtt{P} \times \mathtt{B}$ and $\mathtt{s_g} \subseteq \mathtt{P} \times \mathtt{B}$, one for requests and one for supplies. We may use the shorthand $\mathtt{r_g}(\mathtt{p}, \mathtt{b})$ for $(\mathtt{p}, \mathtt{b}) \in \mathtt{r_g}$ and so forth.

Input points typically do not have any request branches, since the computation is supposed to start up from these. Likewise, output points typically do not have associated supply branches, since the computation is supposed to cease on these.

Both requests and supplies have three components. Requests are defined by the previously mentioned request-guard $\mathtt{r_g}$, the request-function $\mathtt{r_p} : \mathtt{r_g} \longrightarrow \mathtt{P}$ and the request-branchback $\mathtt{r_b} : \mathtt{r_g} \longrightarrow \mathtt{B}$. Likewise the supplies are defined by the supply-guard $\mathtt{s_g}$, the supply-function $\mathtt{s_p} : \mathtt{s_g} \longrightarrow \mathtt{P}$ and the supply-branchback $\mathtt{s_b} : \mathtt{s_g} \longrightarrow \mathtt{B}$.

The second component of requests, $\mathtt{r_p}$ identifies for a legal point and branch index the point the branch leads to (i.e. where data is requested from). Likewise, the second component of supplies, $\mathtt{s_p}$ identifies for a legal point and branch the point where the branch leads to (i.e. where data is supplied to).

The third component of both requests and supplies, the branchback functions ($\mathtt{r_b}$ and $\mathtt{s_b}$), identify the branch index of the opposite direction along the same branch. The request branch-back $\mathtt{r_b}$ gives the branch index at the supplying end of the arc. Similarly, the supply branch-back $\mathtt{s_b}$ gives the branch index at the requesting end of the arc.

In addition requests and supplies satisfy certain axioms that assure their duality and interchangeability.

Pictorially, consider a simple data-dependency pattern sketched in Fig. 1.a).

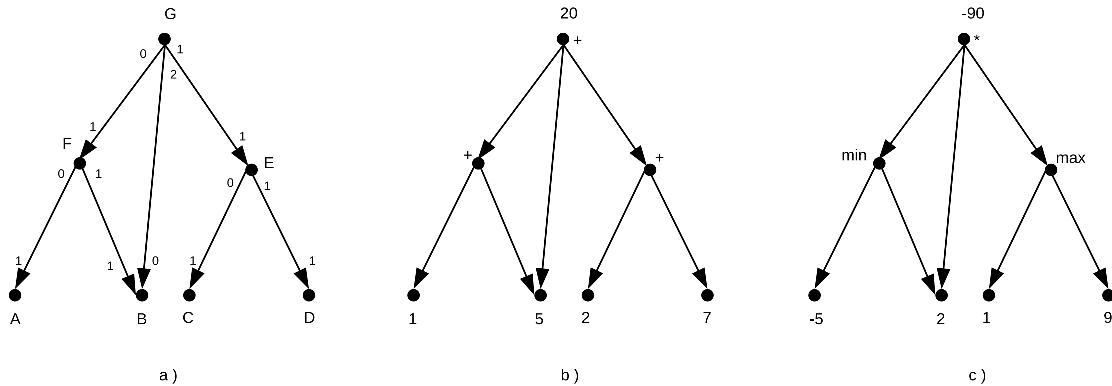

Figure 1: a) A simple data-dependency graph. b) Summing computations are being performed at each point of the DDA, for a given input. c) More complex computations are being performed on the points, for a different input set.

This pattern can be described in the DDA formalism as follows:

- The set of points $\mathtt{P} = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}, \mathtt{G}\}$
- The set of branch indices $\mathtt{P} = \{0, 1, 2\}$
- The requests are defined by:

– $\mathtt{r_g} = \{(\mathtt{F},0),(\mathtt{F},1),(\mathtt{E},0),(\mathtt{E},1),(\mathtt{G},0),(\mathtt{G},1),(\mathtt{G},2)\}$
– the request-function then will be:

$$\begin{array}{llll} \mathtt{r_p}(\mathtt{F},0)=\mathtt{A} & \mathtt{r_p}(\mathtt{F},1)=\mathtt{B} & \mathtt{r_p}(\mathtt{E},0)=\mathtt{C} & \mathtt{r_p}(\mathtt{E},1)=\mathtt{D} \\ \mathtt{r_p}(\mathtt{G},0)=\mathtt{F} & \mathtt{r_p}(\mathtt{G},1)=\mathtt{E} & \mathtt{r_p}(\mathtt{G},2)=\mathtt{B} & \end{array}$$

– and the request branch-back:

$$\begin{array}{llll} \mathtt{r_b}(\mathtt{F},0)=1 & \mathtt{r_b}(\mathtt{F},1)=1 & \mathtt{r_b}(\mathtt{E},0)=1 & \mathtt{r_b}(\mathtt{E},1)=1 \\ \mathtt{r_b}(\mathtt{G},0)=1 & \mathtt{r_b}(\mathtt{G},1)=1 & \mathtt{r_b}(\mathtt{G},2)=0 & \end{array}$$

• And the supplies are defined by:

– $\mathtt{s_g} = \{(\mathtt{A},1),(\mathtt{B},1),(\mathtt{B},0),(\mathtt{C},1),(\mathtt{D},1),(\mathtt{F},1),(\mathtt{E},1)\}$
– the supply-function then will be:

$$\begin{array}{llll} \mathtt{s_p}(\mathtt{A},1)=\mathtt{F} & \mathtt{s_p}(\mathtt{B},1)=\mathtt{F} & \mathtt{s_p}(\mathtt{B},0)=\mathtt{G} & \mathtt{s_p}(\mathtt{C},1)=\mathtt{E} \\ \mathtt{s_p}(\mathtt{D},1)=\mathtt{E} & \mathtt{s_p}(\mathtt{F},1)=\mathtt{G} & \mathtt{s_p}(\mathtt{E},1)=\mathtt{G} & \end{array}$$

– and the supply branch-back:

$$\begin{array}{llll} \mathtt{s_b}(\mathtt{A},1)=0 & \mathtt{s_b}(\mathtt{B},1)=1 & \mathtt{s_b}(\mathtt{B},0)=2 & \mathtt{s_b}(\mathtt{C},1)=0 \\ \mathtt{s_b}(\mathtt{D},1)=1 & \mathtt{s_b}(\mathtt{F},1)=0 & \mathtt{s_b}(\mathtt{E},1)=1 & \end{array}$$

$\mathtt{A},\mathtt{B},\mathtt{C},\mathtt{D}$ are considered input points and F the output. Note that the arrows go in the direction of the requests. This means that data will flow in the opposite direction of the arrows, i.e., along the supply directions.

This DDA may serve as a leading pattern for different computations. In Fig. 1.b) each point of the DDA is associated with a summing operation, i.e., it will add together the data values coming in along its request branches. Consider the array of integers `V:P → Int`, with index-sort P. Provided that the initial values have been assigned to `V[A]`, `V[B]`, `V[C]` and `V[D]`, the computation to be performed on the rest of the points can be defined by:

```
V[p] = if ((p=F)||(p=E)) V[rp(p,0)]+V[rp(p,1)]
       else V[rp(p,0)]+V[rp(p,1)]+V[rp(p,2)]
```

Note how the dependency pattern described by $\mathtt{r_p}$ becomes an explicit entity in the computation, so that computation and dependency become separated in a modular way. [4]

Fig. 1.c) illustrates another computation being performed on the points, for a different input set. We define this for an array of integers `V':P → Int`:

```
V'[p] = if (p=F) min(V'[rp(p,0)],V'[rp(p,1)])
        else if (p=E) max(V'[rp(p,0)],V'[rp(p,1)])
             else V'[rp(p,0)]*V'[rp(p,1)]*V'[rp(p,2)]
```

DDA graphs with associated computations and input values have a double reading. On one hand they specify in a concise way the computations that are to be performed on the points in order to achieve the final result. This may correspond to a top-bottom reading fashion. On the other hand, the supply directions facilitate a bottom-up reading. Starting up from the input values, it drives the computation along the dependencies. The latter reading yields various dependency-driven computational mechanisms depending on the target machine. Be that a sequential, a shared- or distributed-memory parallel machine, or some processing chip for parallel computing, this abstraction serves as a powerful tool to both map the computation to a specific hardware and to generate running code for it [2].

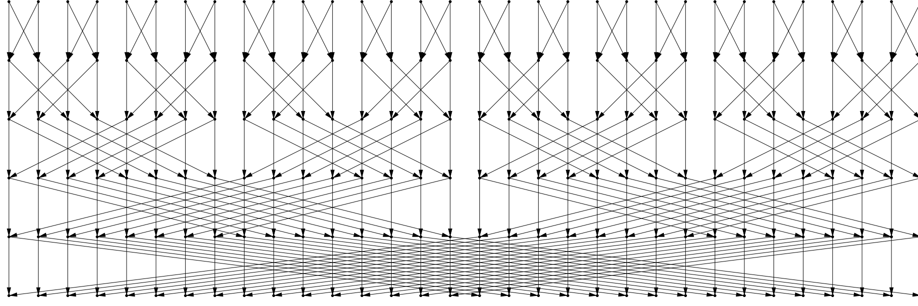

Figure 2: Butterfly dependency for 2^5 input elements.

3 Simple Abstractions, Large Gain

The DDAs are well suited to work with large graphs with regular patterns. Then we can describe the graphs, and the computations on the graphs, as simple program code. In this section we focus on, by examples, how to define DDAs, how we in simple ways can abstract over the placement of the points of a DDA, and how we can exploit this for parallel programming.

We use a notation inspired by ML for defining a point data type, projections of its components and its constructor. For instance, `T = p1 Nat * p2 Nat` would mean that we introduce a type `T` with a data structure consisting of two natural numbers. `p1,p2: Nat` $\rightarrow$ `T` are projections and `T: Nat,Nat` $\rightarrow$ `T` is the constructor.

3.1 Butterfly DDA

Consider first the butterfly dependency, which appears in many divide-and-conquer algorithms, such as the Fast Fourier Transform. In this section we only focus on the dependency pattern itself and its various layouts, and do not specify computations to be performed on the points of the DDA for any particular algorithm. Fig. 2 shows the most common way this dependency is laid out in a two dimensional spatial grid.

A *butterfly DDA of height* $\mathtt{h} \in \mathbb{N}$, $\mathtt{BF_h}$, can be defined with DDA points $\mathtt{BF_h}$ `= row Nat * col Nat` with a data invariant `DI(p) = (row(p)`$\leq$`h) && (col(p)<`$2^{\mathtt{h}}$`)`.

Let the branch indices $\mathtt{B} = \{0, 1\}$, such that all vertical arcs are indexed with 0 at both ends while those across with 1 at both ends. Define the request and supply functions by:

$$
\begin{aligned}
\mathtt{r_g(p,b)} &= \mathtt{0 \leq row(p) < h} \\
\mathtt{r_p(p,b)} &= \mathtt{if\ (b=0)\ then\ BF_h(row(p)+1,col(p))} \\
&\quad\ \mathtt{else\ BF_h(row(p)+1,flip(row(p),col(p)))}
\end{aligned}
$$

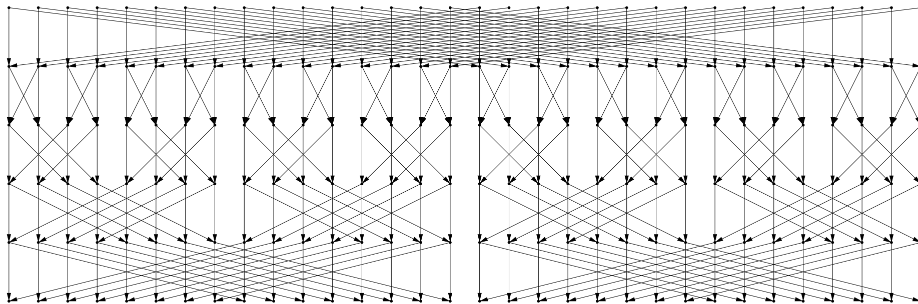

Figure 3: Butterfly dependency laid out using an alternativ DDA-projection.

```
s_g(p,b) = 0<row(p)≤h
s_p(p,b) = if (b=0) then BF_h(row(p)-1,col(p))
           else BF_h(row(p)-1,flip(row(p)-1,col(p)))
```

where `flip(i,n)` flips the $\mathtt{i}^{\mathtt{th}}$ bit (where bit 0 is the rightmost, least significant bit of `n`) in the binary representation of the integer `n`.

In general, the points can be placed in many different ways in a grid. The abstractions available in the DDA concept allow us to easily define different mappings of the same dependency by placing the points in different positions in the grid but preserving the dependency relation between the points. This endows us to control, at a high-level, the embedding of the computation into the available hardware.

Let us illustrate this by changing the layout of the butterfly DDA of height 5, in Fig. 2. First consider projections: `altRow,altCol:BF_h→Nat` with:

```
altRow(p)=row(p)
altCol(p)=ShR_h(col(p),1)
```

The shift right function `ShR_h:Nat,Nat→Nat` is defined such that `ShR_h(n,i)` returns the value of a cyclic shift to the right on the h-bit binary representation of `n` by `i` positions.

The resulting layout is seen Fig. 3. Fig. 4 illustrates the same butterfly dependency but now laid out using yet an other new set of projections for the `BF_h` points:

```
newaltRow(p)=row(p)
newaltCol(p)=ShR_h(col(p),3)
```

We can also define placements that will result in a repetitive network topology, see Fig. 5. Consider `shuffleRow,shuffleCol:BF_h→Nat` such that `shuffleCol` depends on both `col` and `row`:

```
shuffleRow(p)=row(p)
shuffleCol(p)=ShR_h(col(p),row(p))
```

Out of these various layouts, in particular the last one, which is a variation of the shuffle network [11], plays an important role in parallel processing.

These projections can be used as a means of placing computations (at each point) on processors in a network, or choosing placements in silicon for Field Programmable Gate Arrays (FPGAs). Projections make DDAs very flexible and easy to map to different network topologies. As shown, this can be handled on a high-level by defining new projections from the DDA sort.

These figures were generated with our small visualization tool based on DDAs that allows us to experiment with different placements of the DDA points.

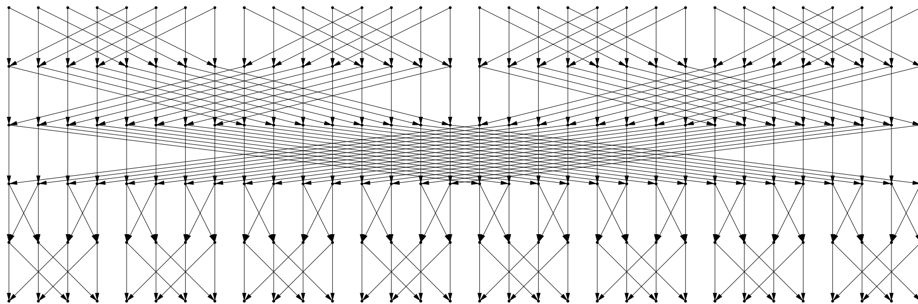

Figure 4: Butterfly dependency with yet another pair of projections.

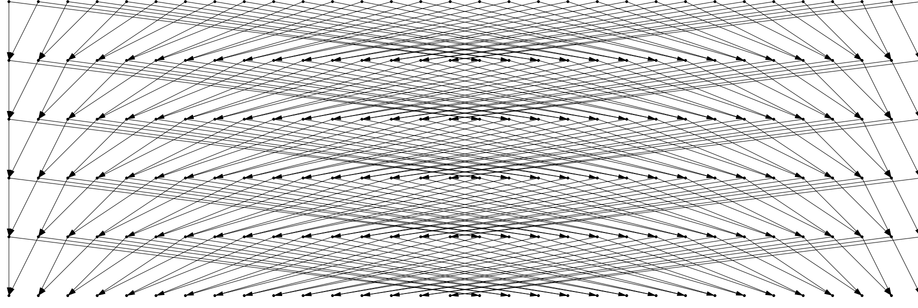

Figure 5: Butterfly dependency laid out as a shuffle network, as controlled by DDA-projections.

3.2 Bitonic Sort DDA

Fig. 6 shows the dependency pattern of the Bitonic sorter. This can be seen as a combination of several butterfly dependencies of different height, each sub-butterfly corresponding to a bitonic merge. This also illustrates how data dependency abstractions entail code-reusability and modularity.

Formally, the *bitonic sort DDA for* 2^{h} *inputs*, h $\in \mathbf{N}$, is defined by DDA points $\mathrm{BS_h}$ = sbf Nat * row Nat * col Nat, and with data invariant:

```
DI(p) = (0<sbf(p)≤h) &&
        ((row(p)<sbf(p)) || (row(p)≤sbf(p) && sbf(p)=1)) &&
        (col(p)<2^h)
```

The first projection identifies the sub-butterfly's height, the row projection is the local row number in the sub-butterfly, while col gives the global column number. For the points which belong to two sub-butterflies, we use the projections corresponding to the lower butterfly. The request and supply functions for branch indices $\mathrm{B} = \{0, 1\}$ become:

```
rg(p,b) = !(row(p)=1 && sbf(p)=0)
rp(p,b) =
    if (((sbf(p)>1) and (0<=row(p)<=sbf(p)-2)) or
       ((sbf(p)=1) and (row(p)=0))) then
              if (b=0) then BS(sbf(p),row(p)+1,col(p))
              else BS(sbf(p),row(p)+1,flip(row(p),col(p)))
    else if ((sbf(p)>1) and (row(p)=sbf(p)-1)) then
              if (b=0) then BS(sbf(p)-1,0,col(p))
              else BS(sbf(p)-1,0,flip(row(p),col(p)))


sg(p,b) = !(row(p)=0 && sbf(p)=h)
sp(p,b) =
    if (!(sbf(p)=h) and (row(p)=0)) then
              if (b=0) then BS(sbf(p)+1,sbf(p),col(p))
              else  BS(sbf(p)+1,sbf(p),flip(sbf(p),col(p)))
    else if (((sbf(p)=1) and (row(p)=1)) or
            (sbf(p)>1) and (1<=row(p)<=sbf(p)-1)) then
              if (b=0) then BS(sbf(p),row(p)-1,col(p))
              else BS(sbf(p),row(p)-1,flip(row(p)-1,col(p)))
```

The input values to be sorted are being assigned to the points of the bottom row.

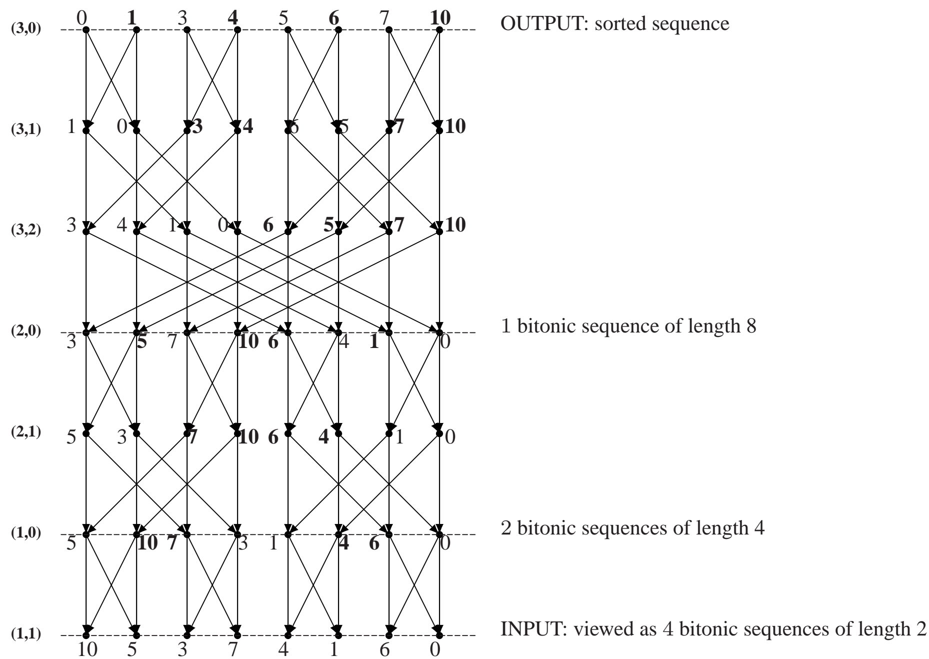

Figure 6: Bitonic sort DDA for sorting 2^3 inputs. On the bottom nodes reside the actual initial data values to be sorted. The rest of the nodes show the values `V[p]` computed at each time-step. On the top row the initial data values have become sorted.

We can now define the expression `V[p]` to be computed on the rest of the points of the bitonic sort DDA.

```
V[p] = if (bit(sbf(p),col(p)) == bit(row(p),col(p))
                  min(V[rp(p,0)],V[rp(p,1)])
          else max(V[rp(p,0)],V[rp(p,1)])
```

The result of the computation are the values of `V[p]` for the points `p` on the top row. Depending on the target machine they may remain in memory for being used in future computations, or offloaded in some way or another.

Fig. 6 also shows the result of the computations of the values `V[p]` on points `p:BS`$_3$ for a concrete initial data set.

How the computation actually proceeds through the DDA (row parallel, a point at a time, or skewed in some strange way), can be controlled by the embedding of the bitonic sort DDA into the space-time communication layout of a chosen hardware architecture.

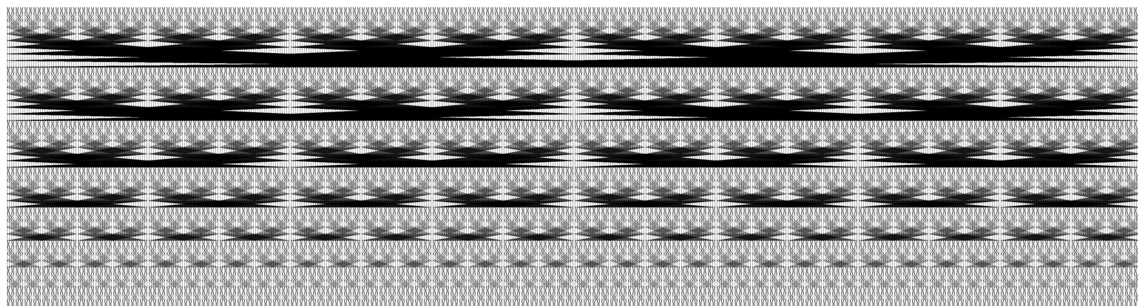

Figure 7: Bitonic sort dependency for 512 inputs.

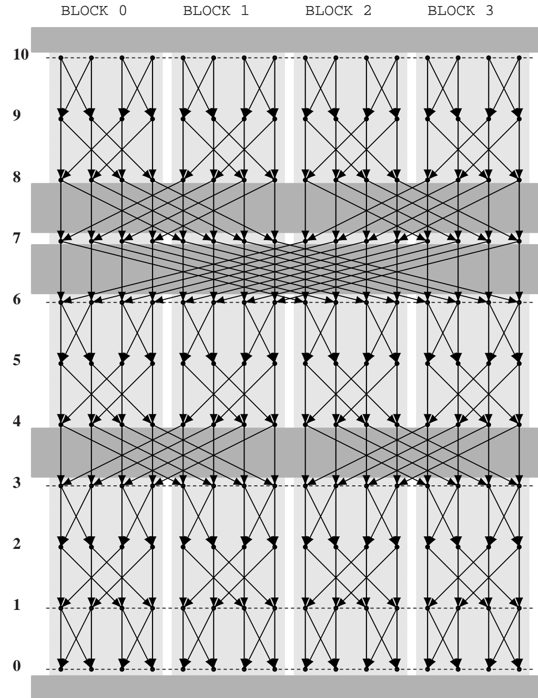

Figure 8: Bitonic sort DDA for 2^4 inputs, embedded into NVIDIA's CUDA programming model, executed by 4 kernel invocations, each consisting of 4 blocks of threads.

Fig. 7 is a generated illustration of this dependency for 512 input elements. Such visualization can be of good help when mapping the computation to a highly parallel chip, e.g., a GPU. [8] It may guide the programmer in choosing the right size parameters for the GPU kernel so that the number of kernel invocations may be optimal. We shall look into this in the next section.

3.3 Embedding into NVIDIA's CUDA

In Nvidia's CUDA [8] the GPU device operates as a Co-processor to the main CPU. The GPU is capable of handling a huge number of threads. The threads are downloaded by the host onto the device in the form of a kernel. Threads within a kernel are organised as a grid of blocks. A block contains a limited number of threads, but a grid of blocks may contain any (reasonable) number of blocks. Threads within one block can synchronize and share data through fast on-chip shared memory. Threads in different blocks within the same kernel can only communicate asynchronously via the main GPU memory. There is no guarantee as to which blocks run in parallel, or in which order blocks are sequenced, when the grid has more blocks than can be executed in parallel. So threads in different blocks are in practice unable to exchange information within the same kernel. The GPU memory is persistent across kernels, and is also a means for initialising computations and outputting results.

The CUDA space-time communication layout is determined by thread-communication over time. Embedding the bitonic sort DDA into this is achieved by mapping $\mathtt{BF}_\mathtt{h}$ points to threads at a given time step, and request-supply arcs to communication channels between threads of consecutive time-steps.

Fig. 8 illustrates the embedding when the number of threads per block is chosen to be 4. We see that the bitonic sort will be split across several kernels in order to communicate

between blocks for the wider branches. The process in each thread will receive data by reading from the memory location corresponding to the communication channel pointed out by the bitonic sort DDA's request component. If this channel is across block numbers, then the read is from the global GPU memory. Otherwise it is local block memory. Then the appropriate min/max values are computed, and data is stored in the appropriate local memory location. Inter-block storage means storing in the global GPU memory. Storage to global GPU memory also takes place at the end of the kernel. With this strategy, the initial data sets are stored in the relevant locations on the global GPU memory, and the result of the bitonic sort algorithm likewise ends in the global GPU memory.

Data shared within one block is through fast shared memory. When the edges of the bitonic sort DDA graph intersects block borders, threads of different blocks need to share data via the global GPU memory. The darker grey frames across all threads highlight this: threads write to the GPU memory, the kernel terminates, and control is handed over to the host, which invokes a new kernel with threads first fetching data from the GPU memory.

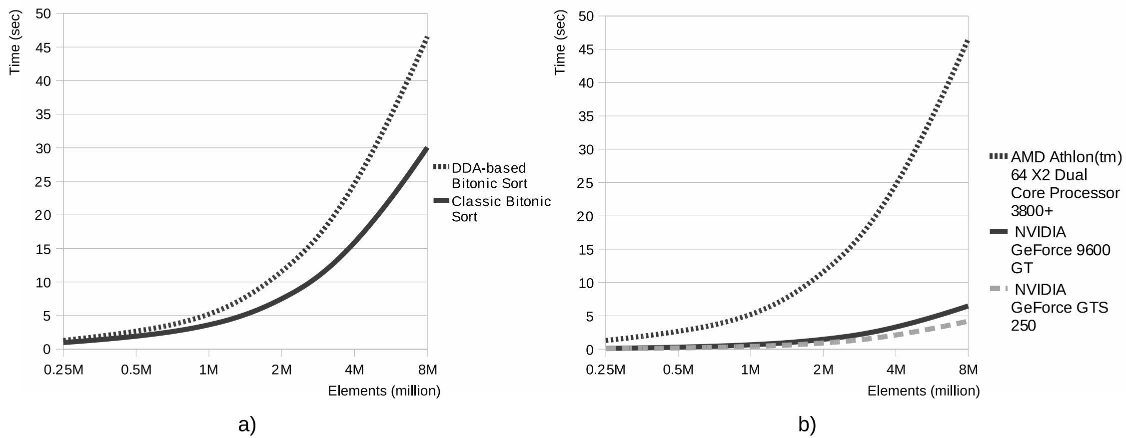

Figure 9: a) DDA-based bitonic sort vs. classical bitonic sort running times on the same CPU. b) DDA-based bitonic sort running times on various platforms.

4 Experiments

We implemented two dependency-driven bitonic sorters based on the bitonic sort dependency and the corresponding expression described in section 3.2. The first implementation was a sequential C program, and the second a parallel CUDA program targeting Nvidia GPUs. The bitonic sort dependency was implemented in a separate module (C header file), so that both implementations could use the same module[1]. The order in which DDA points are chosen to compute the expression is architecture specific, hence the computation mechanisms differed somewhat from one implementation to the other.

The sequential DDA-based bitonic sorter was tested on an AMD Athlon 64 X2 Dual Core processor against the classic implementation of bitonic sort (with recursive calls to itself), see Fig. 9.a). Experiments show that the dependency-driven computation is 1.4 times slower. This should not come as a surprise, if we think about the machinery that

[1]In case of the CUDA-implementation CUDA-specific keywords needed to be inserted in front of the function declarations, but the rest of the module remained intact.

drives the computation (supply function calls, projection-, constructor-calculations, etc. at every step). A rigorous profiling could probably help to optimise the DDA-module, but we did not look into this closely.

On the other hand, the DDA-abstraction made it pretty easy to port the dependency-driven bitonic sorter to CUDA. We compared the sequential DDA-based bitonic sorter against the CUDA implementation. The latter was tested on the above mentioned CPU together with Nvidia GeForce 9600 GT and Nvidia GeForce GTS 250, respectively. Fig. 9.b) summarises the results over these platforms. The speedup when going parallel is significant: the parallel version runs 7-11 times faster than the sequential version.

While our dependency-driven CUDA-implementation compared to fine-tuned, sophisticated, and in most cases hybrid GPU sorting algorithms ([5, 10, 9]) is less efficient, the experiments underpin that DDA-based programming is highly portable, making the approach promising for parallel computing.

5 Implementing a DDA-enabled Compiler

In the above implementations we hand coded the computation mechanisms specific to each hardware architecture. To be able to generate these codes a DDA-enabled compiler is needed. Then in a corresponding base language we can define DDAs, computations over DDA-points, and target architectures as STAs. Currently, these efforts are being carried out in the framework of the Magnolia programming language [1], which itself is under development. Magnolia allows the definition of concepts to specify the interface and behaviour of abstract data types which are useful to express our DDA-formalism based on axioms.

6 Conclusion

This paper revisits the concept of Data Dependency Algebras pointing out how computations and dependencies can be separated in a modular way. We show, by examples, how to define DDAs, how we can abstract over the placement of the points of a DDA, and how we can exploit this for parallel programming. We illustrate the viability of this setting by using our small visualization tool to generate drawings of various spatial placements of dependency patterns, and by presenting benchmark results of DDA-based computations. Practical experiments show that once the dependency pattern of a computation is defined in a separate module, this can be reused over various platforms. This results in high portability, but at the cost of less optimal running times compared to fine-tuned platform specific program codes. This problem may be hopefully alleviated by a DDA-enabled compiler with a proper optimizing mechanism in place.

References

[1] A. H. Bagge. *Constructs & Concepts, Language Design for Flexibility and Reliability*. PhD thesis, Department of Informatics, University of Bergen, Norway, October 2009.

[2] E. Burrows and M. Haveraaen. A hardware independent parallel programming model. *Journal of Logic and Algebraic Programming*, 78:519–538, 2009.

[3] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation – A performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[4] V. Cyras and M. Haveraaen. Programming with data dependencies: a comparison of two approaches. In U. H. Engberg, K. G. Larsen, and P. D. Mosses, editors, *Proceedings of the 5th Nordic Workshop on program correctness, 1994.*, number 94-6 in BRICS notes series, pages 112–126, 1994.

[5] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, New York, NY, USA, 2006. ACM.

[6] M. Haveraaen. Data dependencies and space time algebras in parallel programming. Technical Report 45, Department of Informatics, University of Bergen, Norway, June 1990.

[7] W. L. Miranker and A. Winkler. Spacetime representations of computational structures. *Computing*, 32(2):93–114, 1984.

[8] NVIDIA. CUDA Programming Guide. Technical report, Nvidia, 2009.

[9] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10, 2009.

[10] E. Sintorn and U. Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *J. Parallel Distrib. Comput.*, 68(10):1381–1388, 2008.

[11] H. Stone. Parallel processing with the perfect shuffle. *Computers, IEEE Transactions on*, C-20(2):153–161, Feb. 1971.

[12] M. Wolfe, editor. *High Performance Compilers for Parallel Computing*. Addison Wesley; Reading, Mass., 1996.