Algebraic Combinators for Data Dependencies and Their Applications

Eva Burrows
Bergen Language Design Laboratory
Department of Informatics, University of Bergen

27th Nordic Workshop on Programming Theory
21-23 October 2015
Reykjavik, Island

Outline

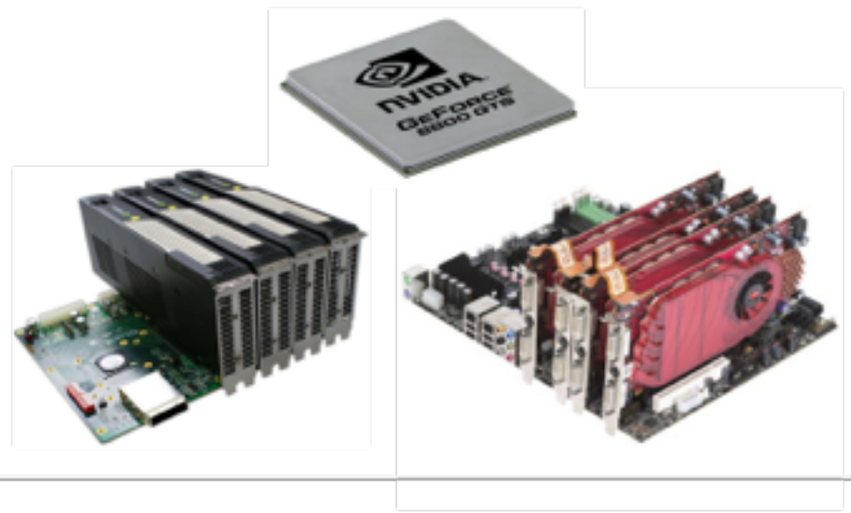
- Motivation
- Overall Approach
- Algebraic Combinators
- Illustration
- Summary

Programmability & Portability

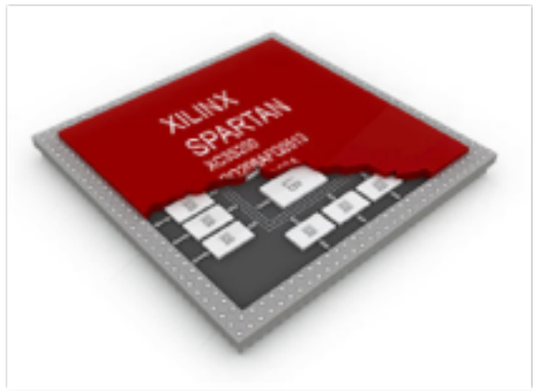
- Overwhelming parallelism: from chips to supercomputers
- Numerous programming models: Thread-based, Intel TBB, Intel Cilk Plus, OpenMP, MPI, Cuda, OpenCL, OpenAcc, etc, etc...

Multi-cores

GPUs



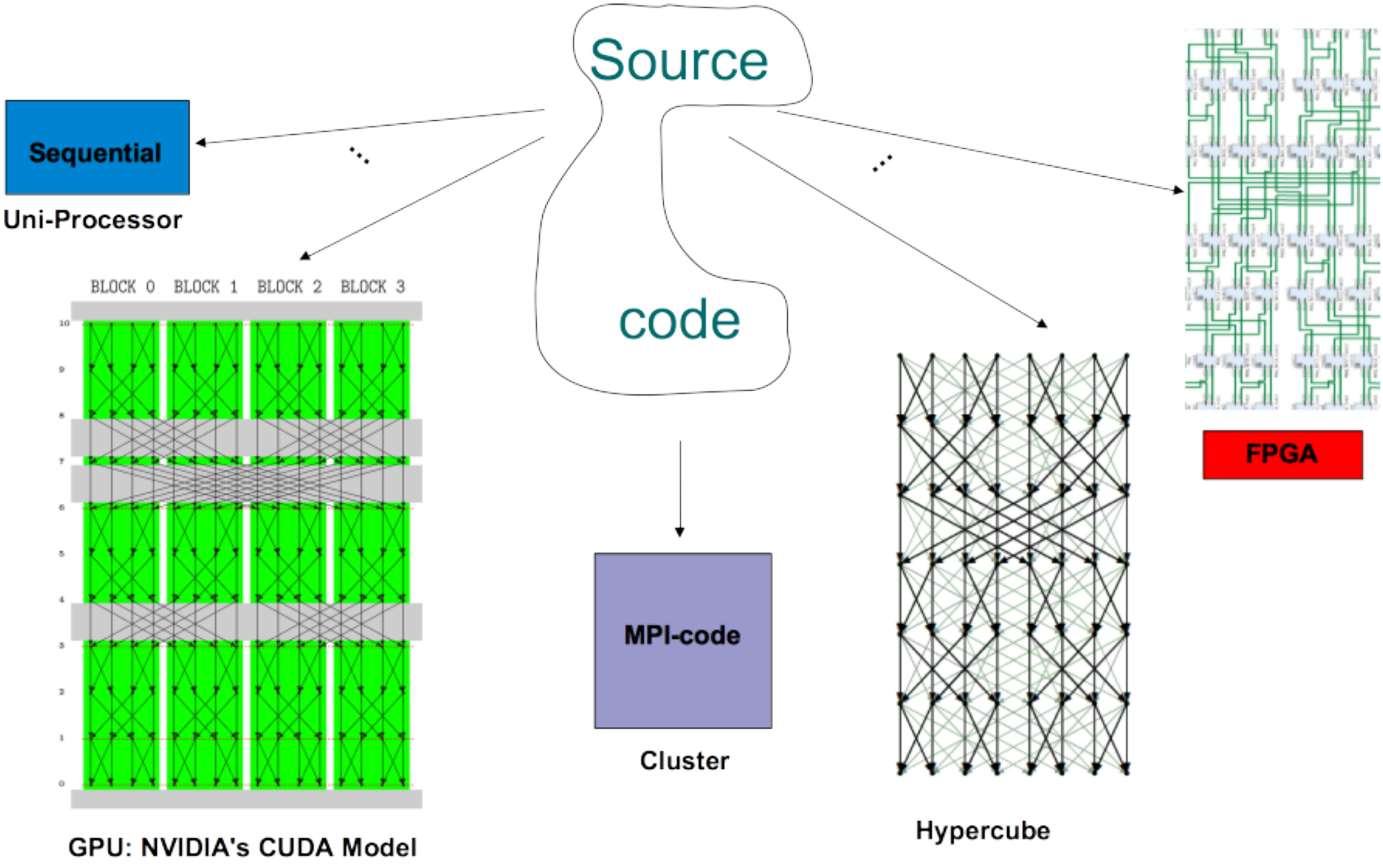
FPGAs



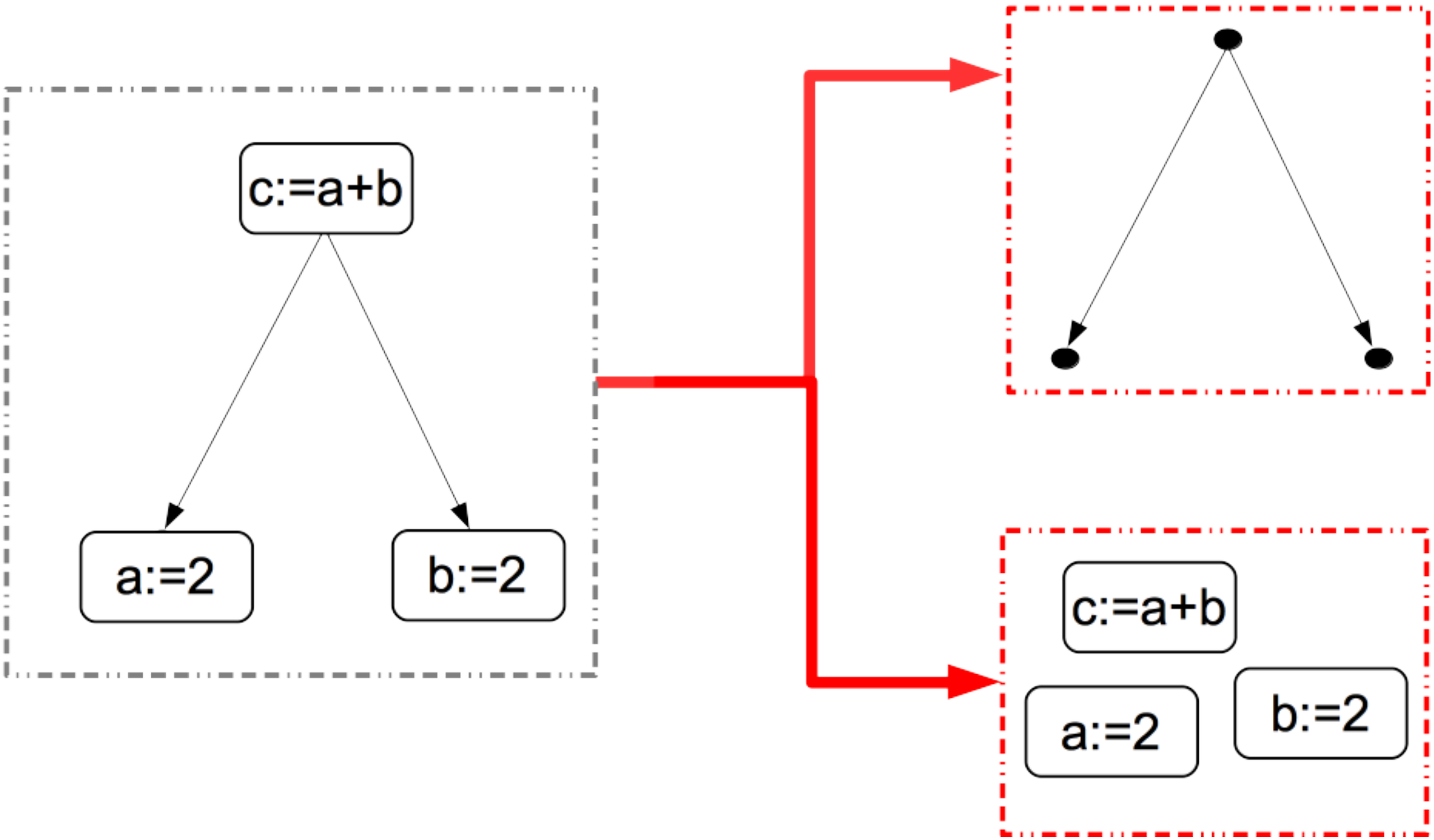
Supercomputers

becoming truly heterogeneous platforms

One Source Code for all Devices

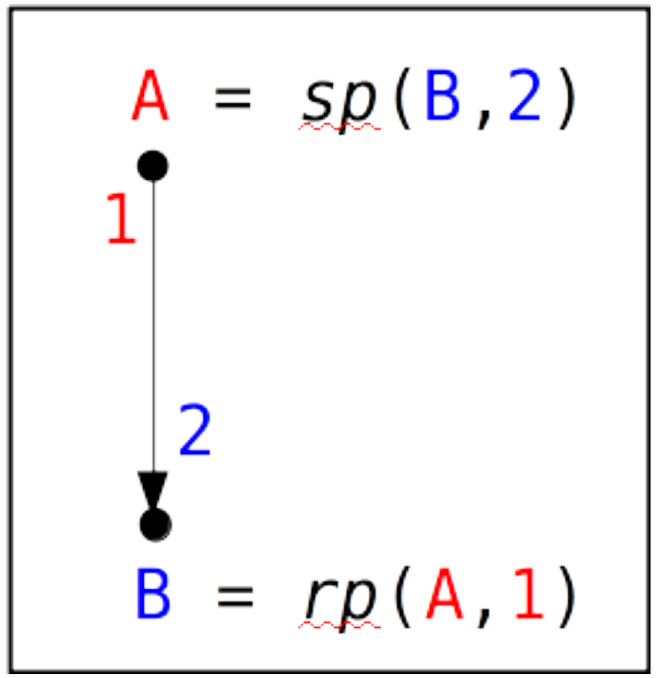


From a Data-flow Graph to a Language Abstraction

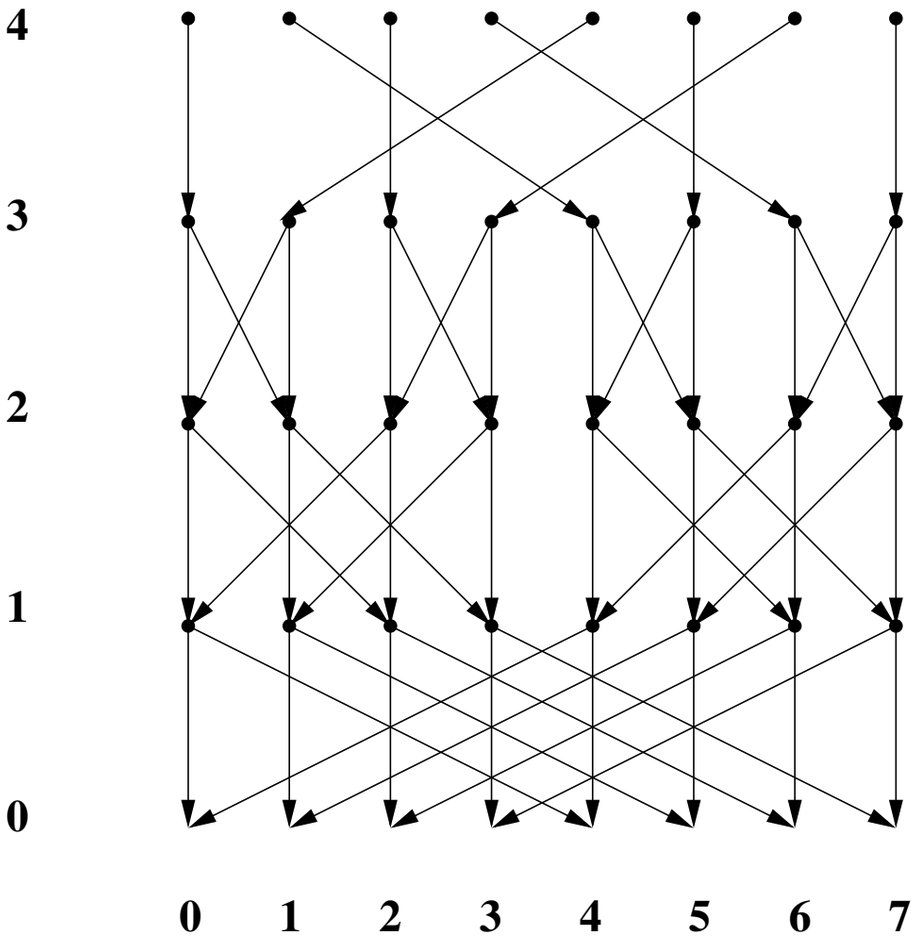


An API for Data Dependency Graphs (DDA)

- A Data Dependency Graph will be an implementation of a specific DDA API consisting of:
  - **types**: Points (P), Branches (B)
  - **functions**:
    - requests: (rp, rb, rg)
    - supplies: (sp, sb, sg)
  - **axioms**



An Example: DDA-based Fast Fourier Transform

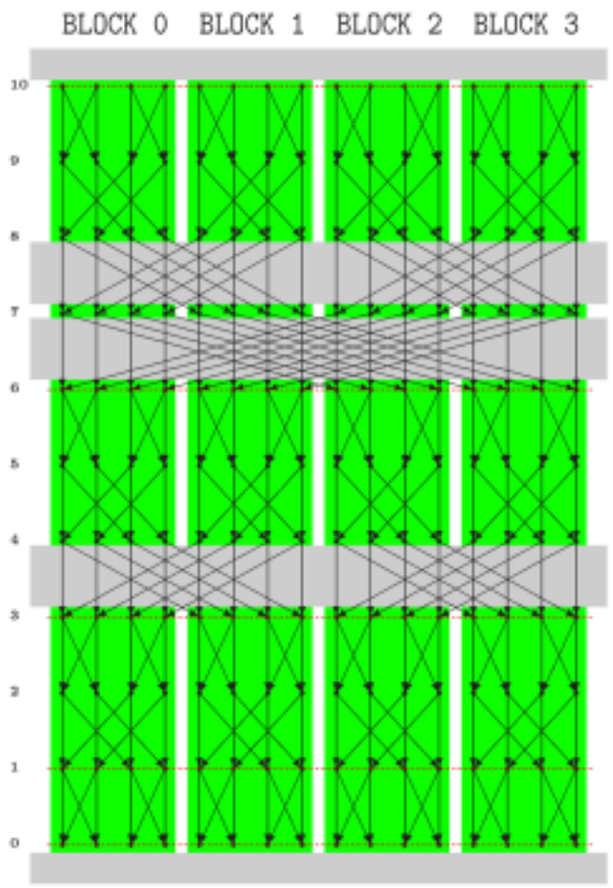


- let $DFFT_h$ be the DDA defined for the FFT graph
- let V be a global `complex` array indexable by the point type FFT_h of the DDA:

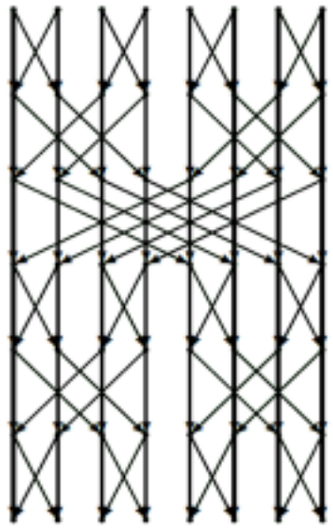
```
1 repeat p:FFTh along DFFTh from V in
2 V[p]=
3   if (row(p)<h+1)
4     if (col(p)<col(rp(p,1)))
5         V[rp(p,0)]+V[rp(p,1)]*W^revh(col(p)>>h-row(p))
6     else V[rp(p,1)]+V[rp(p,0)]*W^revh(col(p)>>h-row(p))
7   else V[rp(p,0)]
```

Embeddings from DDA to the HW

Uni-Processor

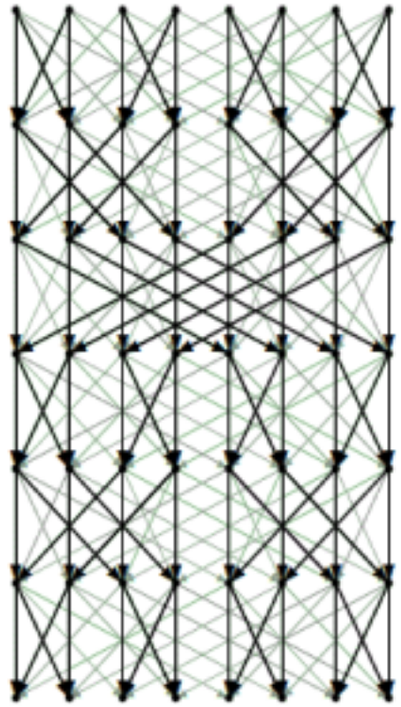


GPU: NVIDIA's CUDA Model

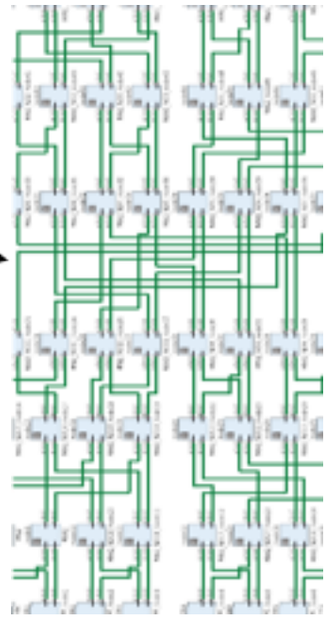


DATA DEPENDENCY GRAPH

Cluster



Hypercube



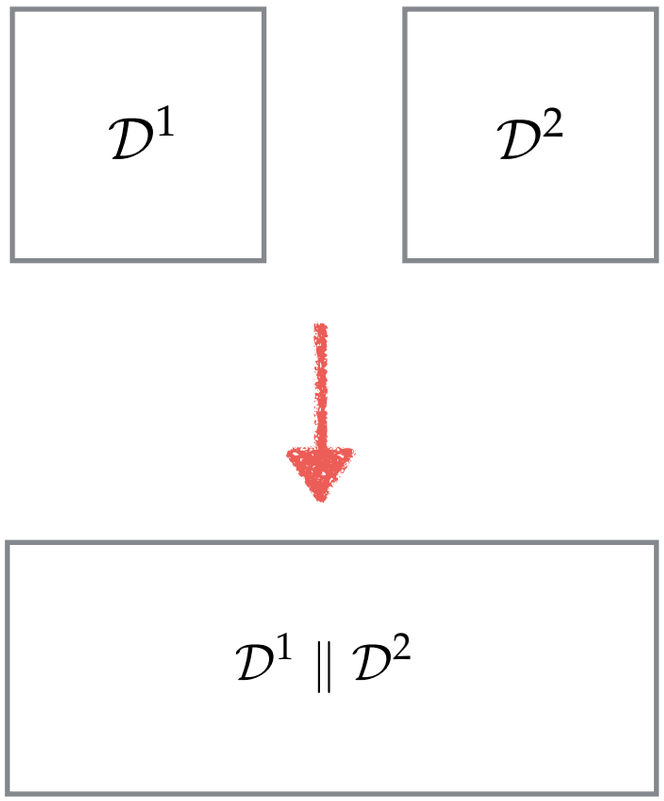
Compound DDAs

- complex DDAs from existing DDA implementations using *algebraic combinators on the DDA API*
    - parallel, serial, sub-DDA, nesting, etc…
- compound DDAs are declared in the source code, but generated at compile time
- repeat statements and embeddings are defined on compound DDAs of custom complexity
- compound DDAs are genuine = combinators preserve the DDA API axioms

Parallel Combinator

- Given two DDAs: $\mathcal{D}^1 = \langle P^1, B^1, req^1, sup^1 \rangle \qquad \mathcal{D}^2 = \langle P^2, B^2, req^2, sup^2 \rangle$
- Parallel combination: $\mathcal{D}^1 \parallel \mathcal{D}^2 \stackrel{Def}{=} \mathcal{P} = \langle P^{\mathcal{P}}, B^{\mathcal{P}}, req^{\mathcal{P}}, sup^{\mathcal{P}} \rangle$
- Source code:

```
P = D1 par D2
```



Serial Combinator

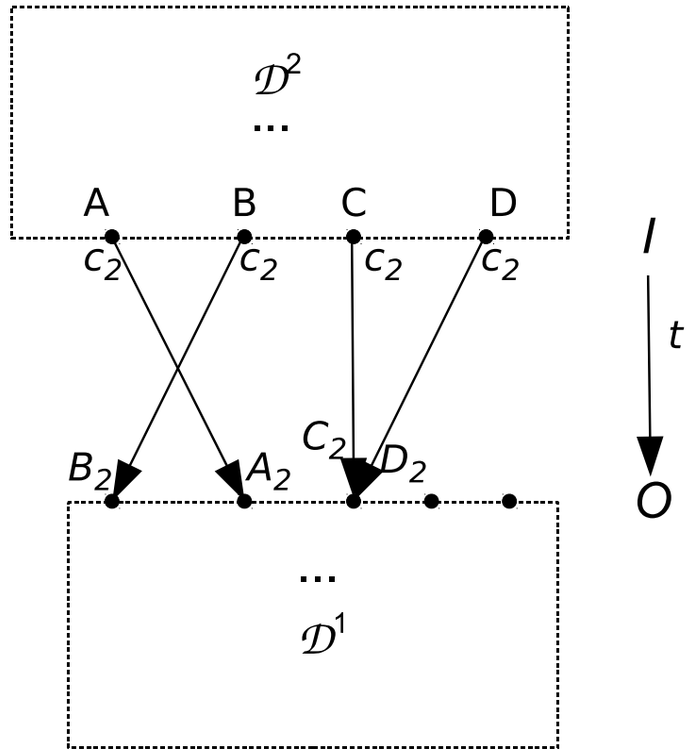
- Given two DDAs: $\mathcal{D}^1 = \langle P^1, B^1, req^1, sup^1 \rangle \qquad \mathcal{D}^2 = \langle P^2, B^2, req^2, sup^2 \rangle$

- Serial combination wrt. a total function:

$$\mathcal{D}^1 \overset{t}{\mapsto} \mathcal{D}^2 \overset{Def}{=} \mathcal{S}_t = \langle P^{\mathcal{S}_t}, B^{\mathcal{S}_t}, req^{\mathcal{S}_t}, sup^{\mathcal{S}_t} \rangle$$

- Source code:

```
S = D1 seq D2 via t
or
S = D1 seq D2 bij t
```



SubDDA Combinator

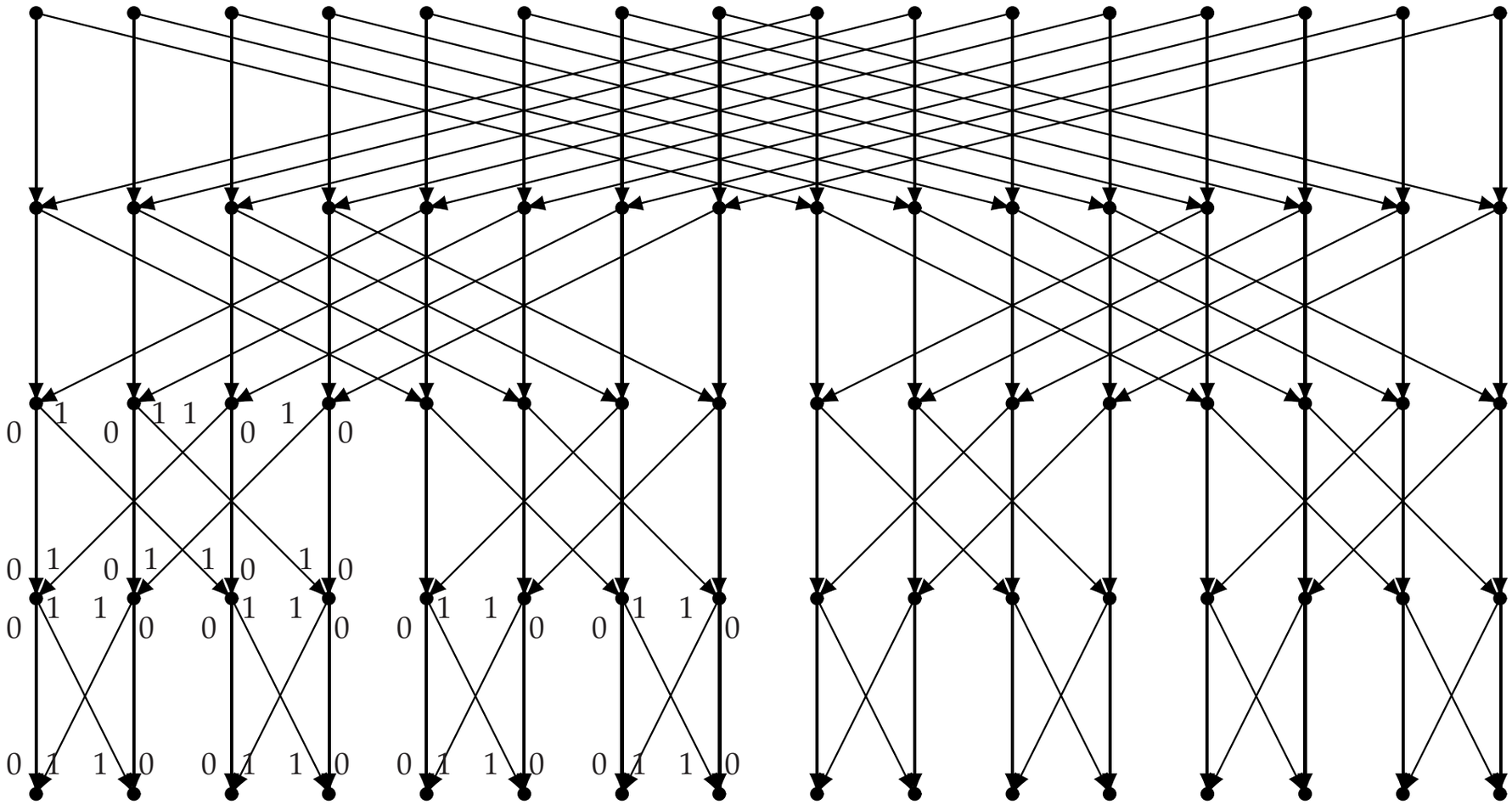
- Given a DDA: $\mathcal{D} = \langle P, B, req, sup \rangle$
- SubDDA combinator defined wrt. subsets of P and B:

$$\mathcal{D}\big|_{P',B'} \stackrel{Def}{=} \mathcal{D}' = \langle P', B', req', sup' \rangle$$

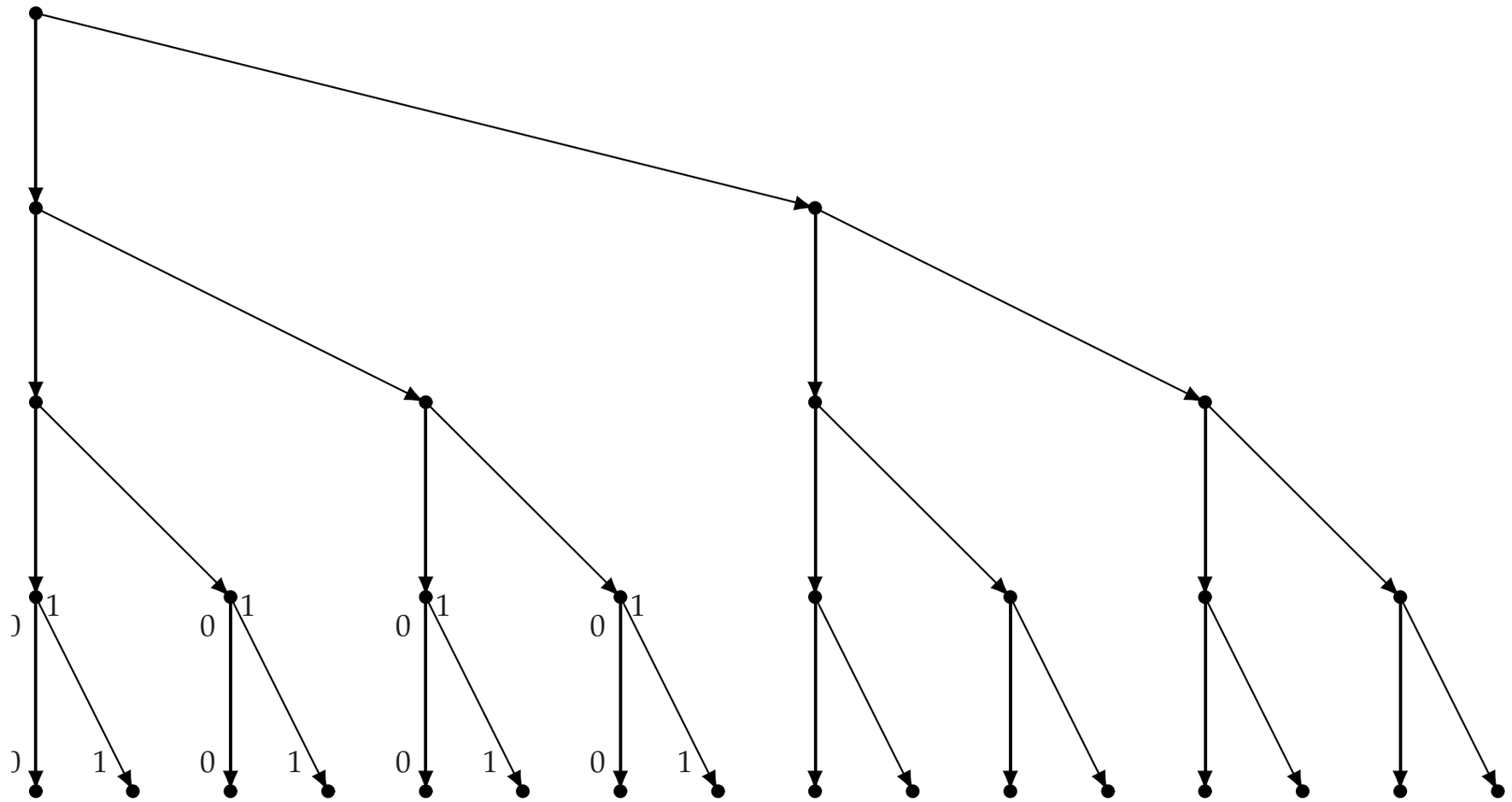
- Acts like “forgetting” parts of the graph
- Source code:

```
D’= sub D along (P’,B’)
```

SubDDA: from FFT to …



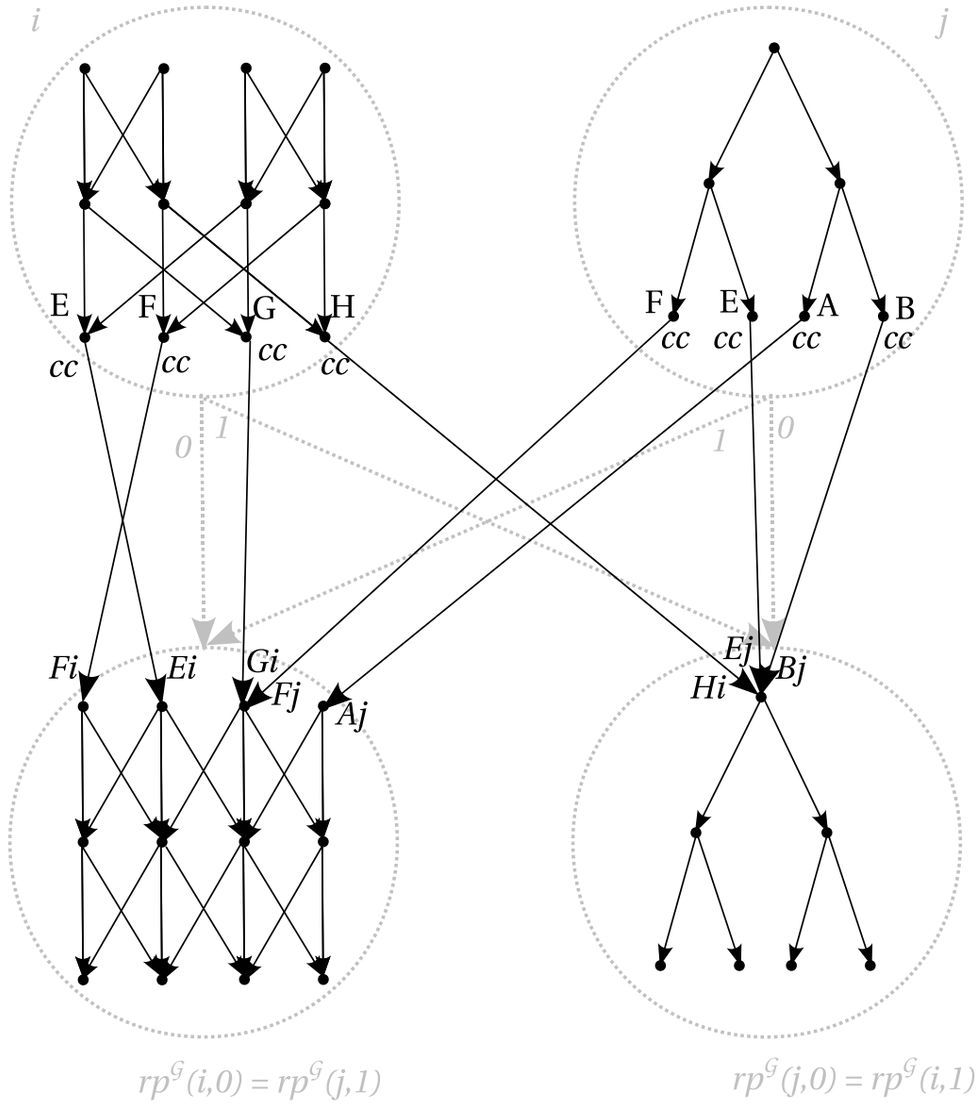
SubDDA: from FFT to Binary Tree



Nesting DDA Combinator

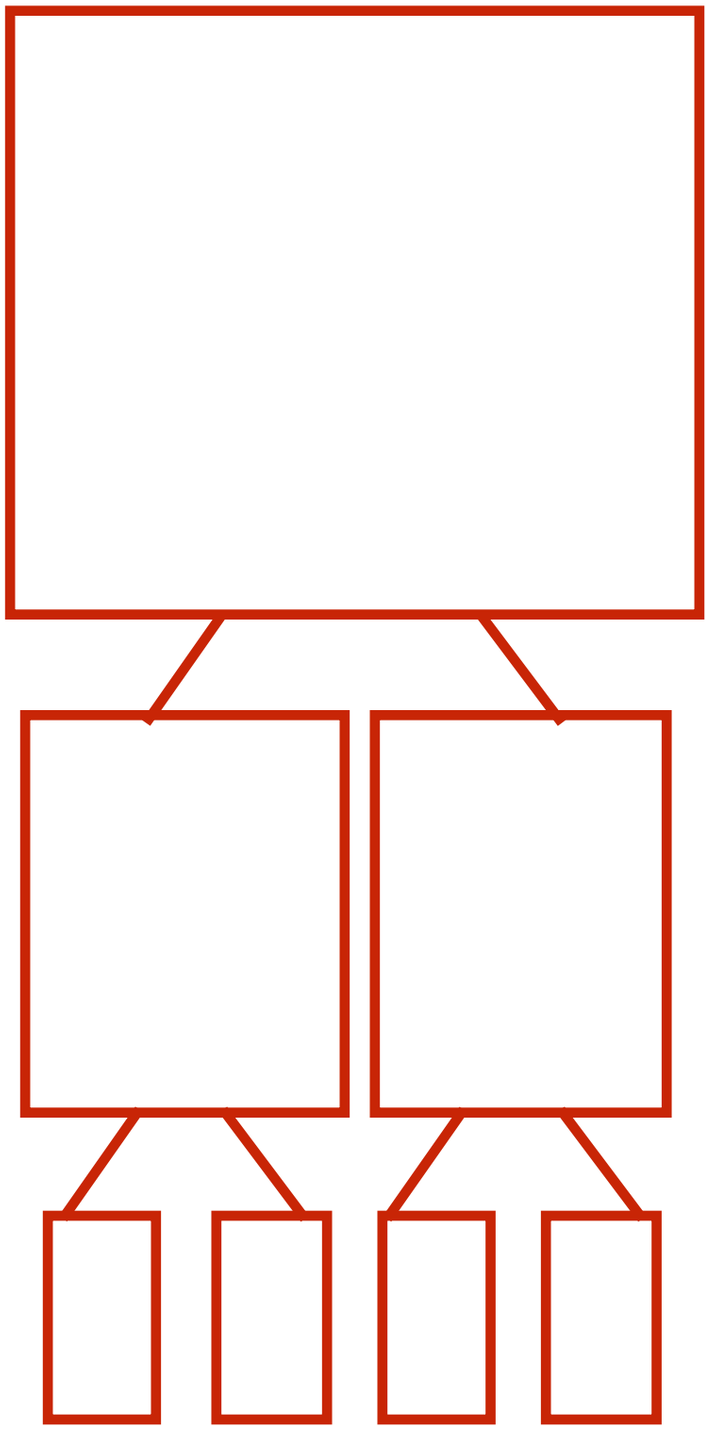
- Global DDA: $\mathcal{G} = \langle P^{\mathcal{G}}, B^{\mathcal{G}}, req^{\mathcal{G}}, sup^{\mathcal{G}} \rangle$
- Family of (local) DDAs, one for each point in $P^{\mathcal{G}}$
- Nested DDA defined wrt. a family of total function

$$\mathcal{N}^{\mathcal{G}}_{\mathcal{D},t} \stackrel{Def}{=} \mathcal{N} = \langle P^{\mathcal{N}}, B^{\mathcal{N}}, req^{\mathcal{N}}, sup^{\mathcal{N}} \rangle$$



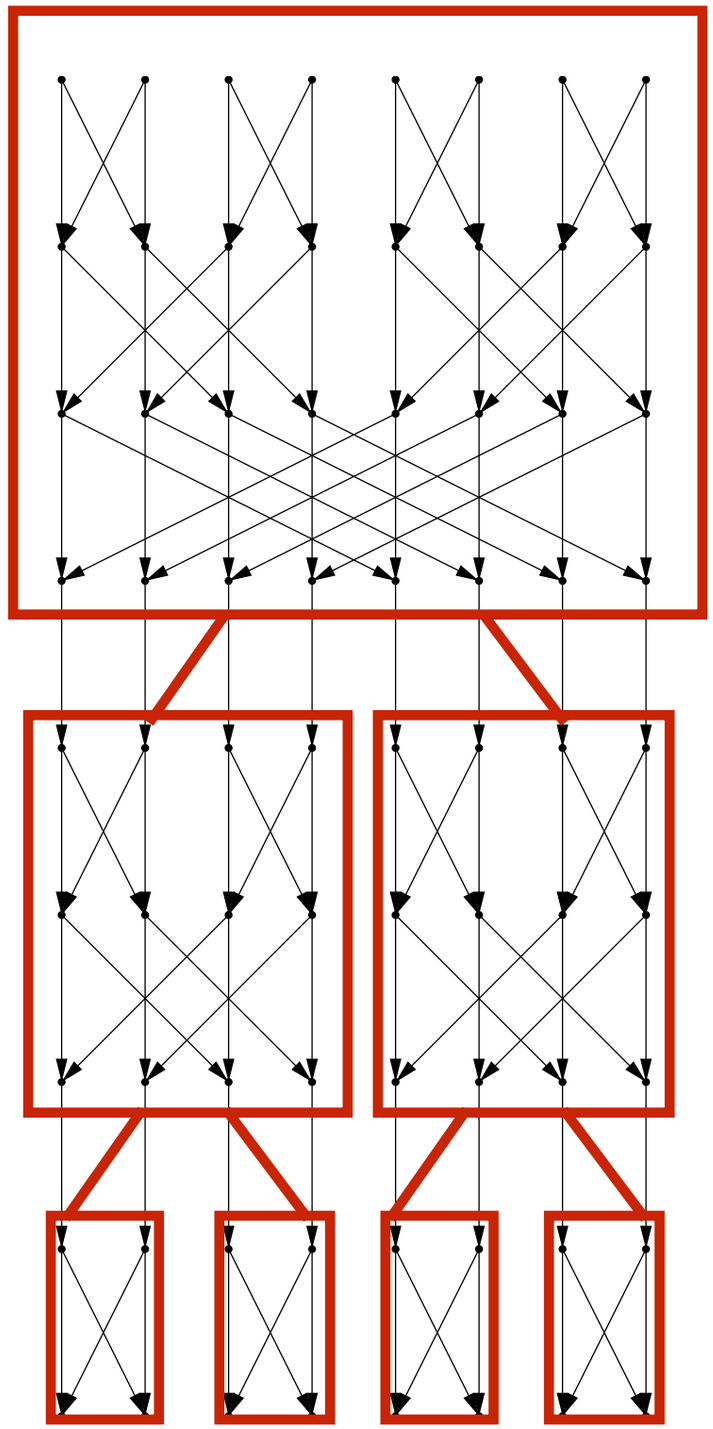
Nesting Combinator for Divide and Conquer

- Global DDA: Binary Tree (“Divide”)



Nesting Combinator for Divide and Conquer

- Local DDAs : “Combine”, e.g. FFTs



Nesting Combinator for Divide and Conquer

- Compound DDA : Bitonic Sort

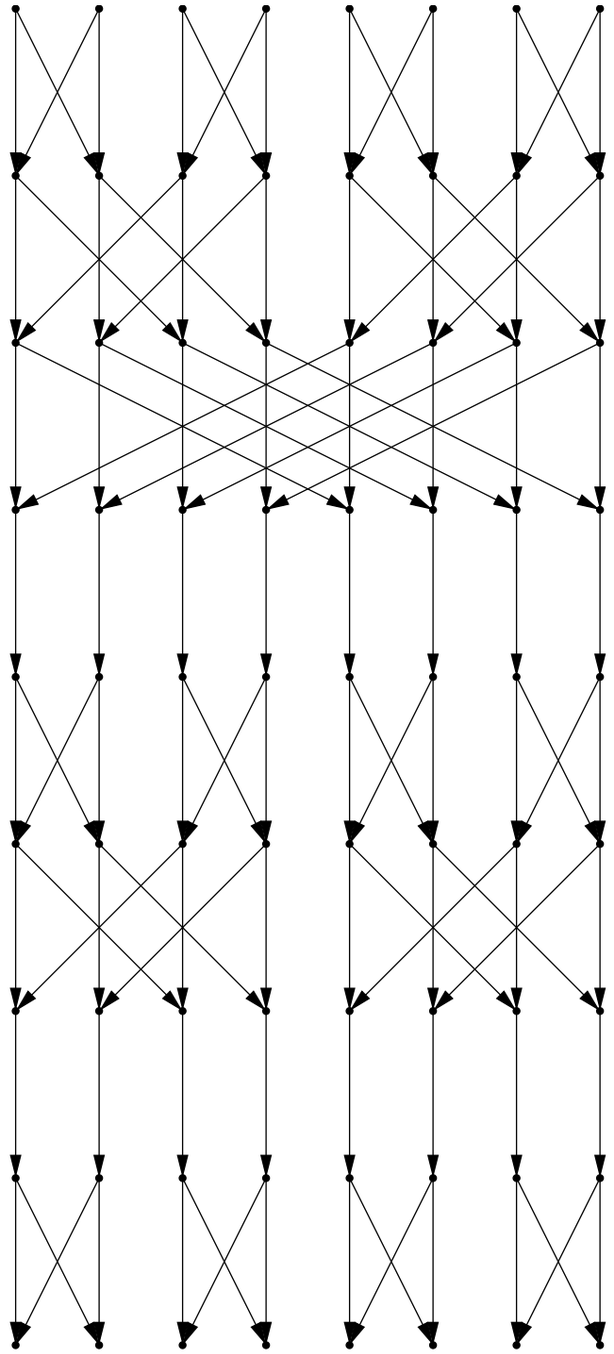
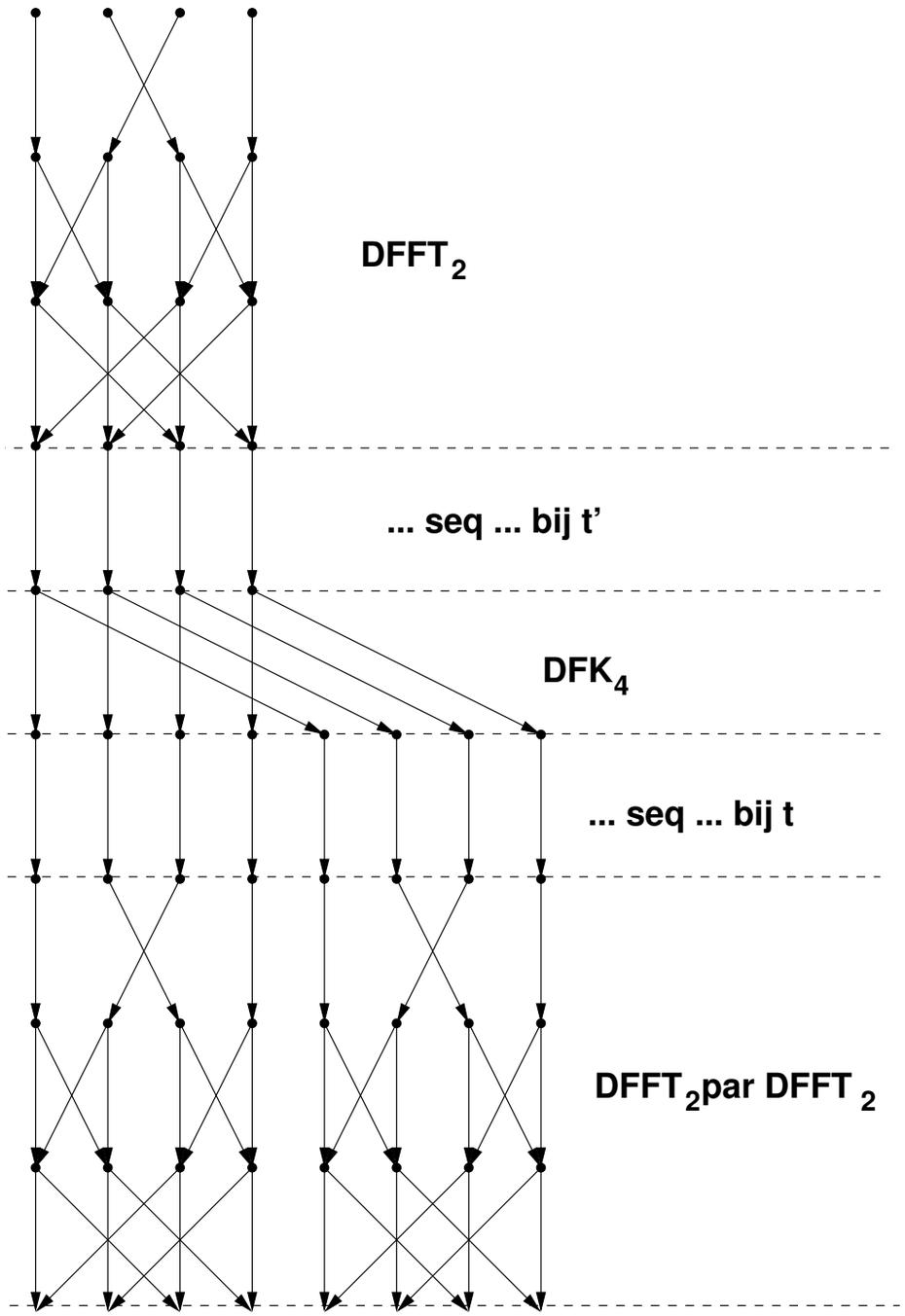


Illustration: Polynomial Multiplication



2 $DFFT_2$ to evaluate each polynomial via FFT

- efficient FFT-based algorithm
- underlying dependency can be expressed as a compound DDA:

$$DPM_{2^h} = ((DFFT_h \text{ par } DFFT_h) \text{ seq } DFK_{2^h} \text{ bij t}) \text{ seq } DFFT_h \text{ bij t'}$$

- repeat statement:

```
1 repeat p:PM2h along DPM2h from V for TP in
2 if ((tag(p)=4)&&(row(v4(p))=0)) V[rp(p,i3(c))]
3  else if ((tag(p)=3)&&(row(v3(p))=0)) V[rp(p,i2(c))]
4   else if (tag(p)=1) exp1FFT
5    else if (tag(p)=2) exp2FFT
6     else if (tag(p)=3) V[rp(p,i1(0))]*V[rp(p,i1(1))]
7      else exp3INV-FFT
```

Summary

- DDA-based approach for programmability and portability across various parallel HW
- algebraic combinators to work on top DDA API to ease the definition of complex DDAs in the source code while implementations are generated at compile time
- combinators preserve the DDA API axioms, hence all DDA-based parallelisation strategies apply to the compound DDA