An embedded language for data-parallel programming

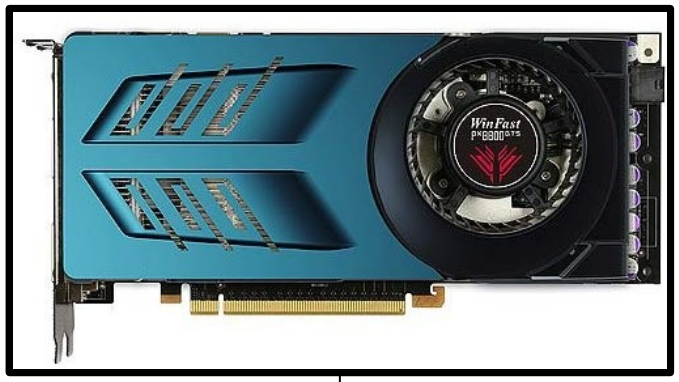
Master of Science Thesis in Computer Science
By Joel Svensson

Department of Computer Science and Engineering

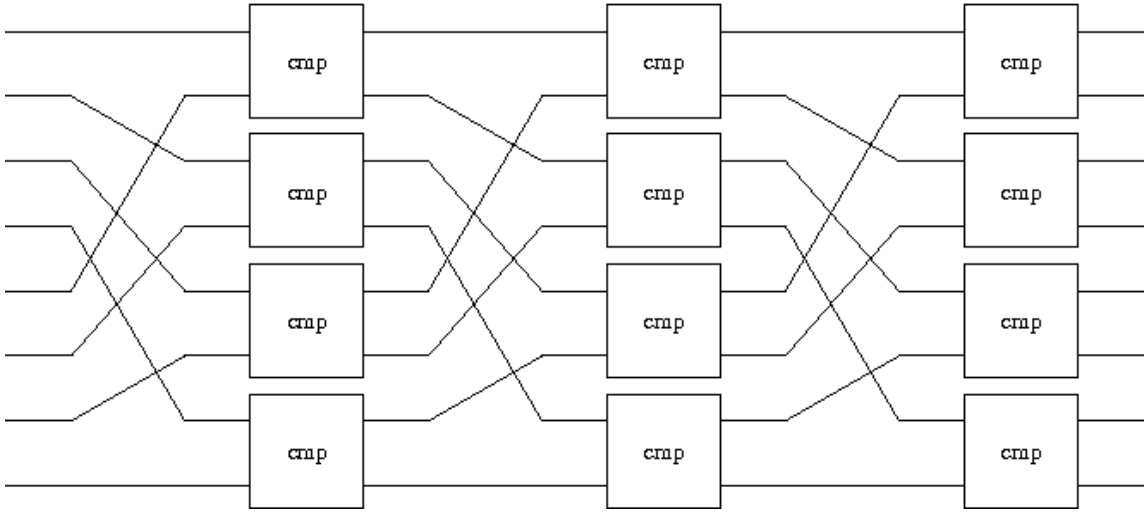
CHALMERS UNIVERSITY OF TECHNOLOGY
GÖTEBORGS UNIVERSITY
Göteborg, Sweden

Obsidian: an embedded language for data-parallel programming

- Data-parallel programming
- General-Purpose computations on the GPU (GPGPU)
- Lava



NVIDIA 8800 GPU



Project Outline

- An embedded language for data-parallel programming
- Lava programming style using combinators
- Generate C code for NVIDIA GPU

Data-parallel programming

- Single sequential program
- Executed by a number of processing elements
- Operating on different data

```
for j := 1 to log(n) do
   for all k in parallel do
      if ((k+1) mod 2^j) = 0 then
         x[k] := x[k-2^(j-1)] + x[k]
      fi
   od
od
```

GPGPU

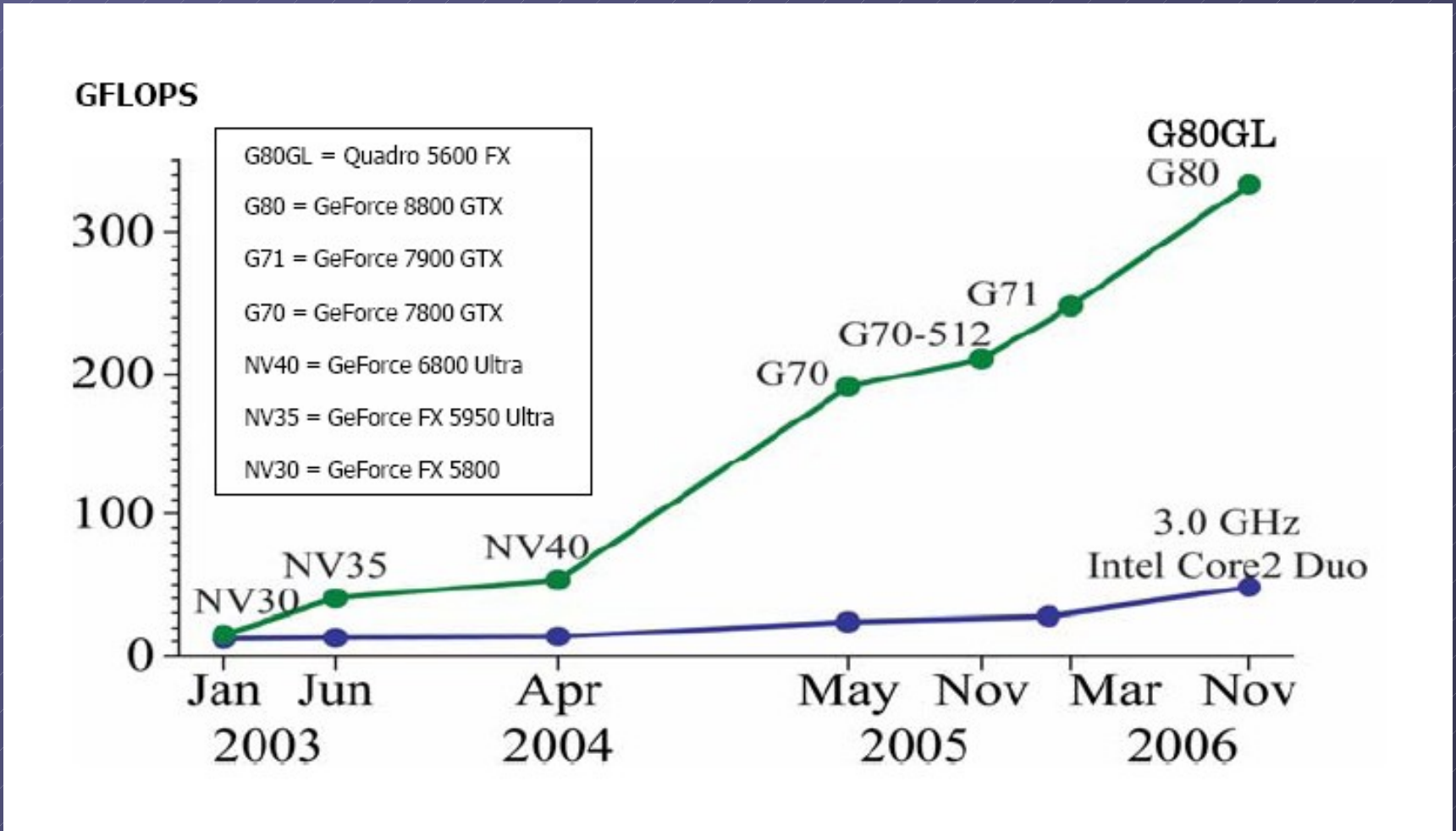
- GPUs are relatively cheap
  - High performance (Hundreds of GFLOPS)

Applications:

- Physics simulation
- Bioinformatics
- Sorting

www.gpgpu.org

GPU vs CPU GFLOPS Chart

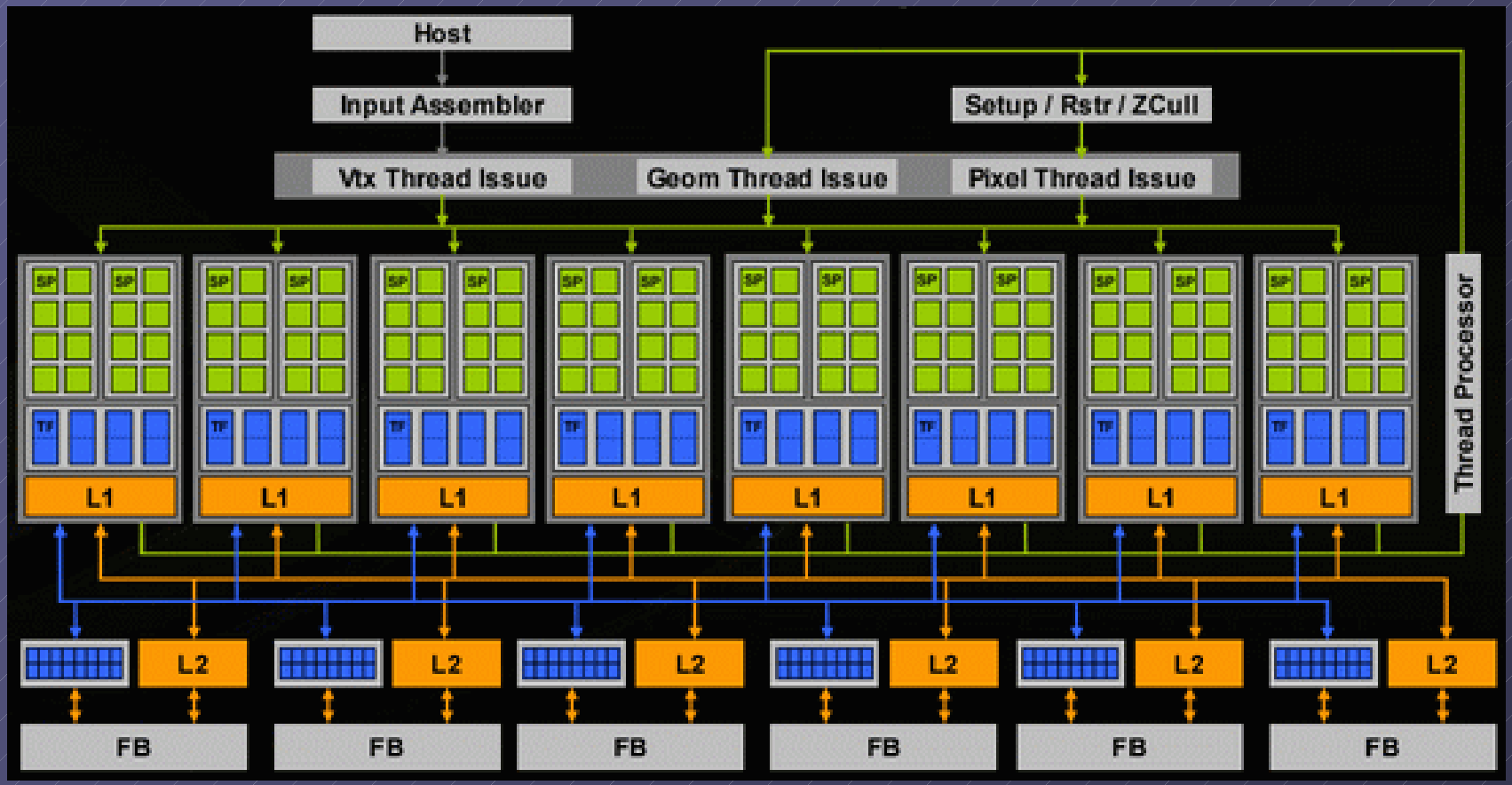


NVIDIA 8800 GPUs

- A set of SIMD multiprocessors
- 8 SIMD processing elements per Multiprocessor
- Up to 16 multiprocessors in one GPU
- Giving 128 processing elements total

www.nvidia.com

NVIDIA 8800 GPUs



NVDIA Compute Unified Device Architecture

- C compiler and libraries for the GPU
- GPU as a highly parallel co-processor
- for use with NVIDIA's 8800 series GPUs

www.nvidia.com/cuda

CUDA Programming model

- High number of threads
  - Divided into Blocks
- Thread block
  - 512 Threads
  - Divided into Warps
  - Executed on one multiprocessor

CUDA Synchronisation

- CUDA supplies a synchronisation primitive, __syncthreads()
  - Barrier synchronisation
  - Across all the threads of a block
- Coordinate communication

Obsidian

- Embedded in Haskell
- Presents a high level programmers interface
- Parallel computations described using combinators
- CUDA C code is generated

Obsidian

Describes computations on arrays:

- Length homogeneous
  - Sorting algorithms
- Integer values

Limitations:

- Currently limited to iterative sorting algorithms

Obsidian Programming

- Basics
  - Sequential composition of programs: `->-`
  - Parallel composition of programs: `parl`
  - Index operations:
    - `rev`
    - `riffle`
    - `unriffle`
  - Array operations:
    - `halve`
    - `conc`
  - Apply or Map: `fun`

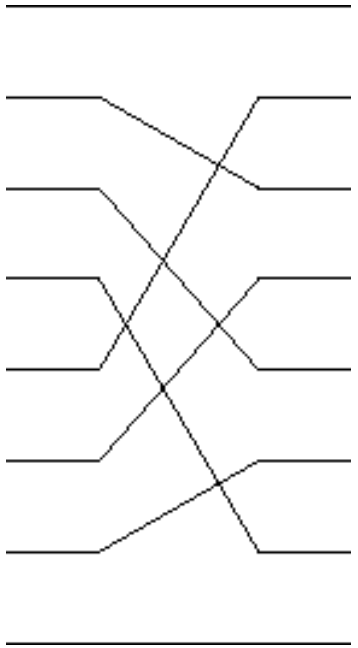
Obsidian Programming

- Array Operations
  - `halve`
  - `conc`
  - `oeSplit`
  - `shuffle`

Obsidian Programming

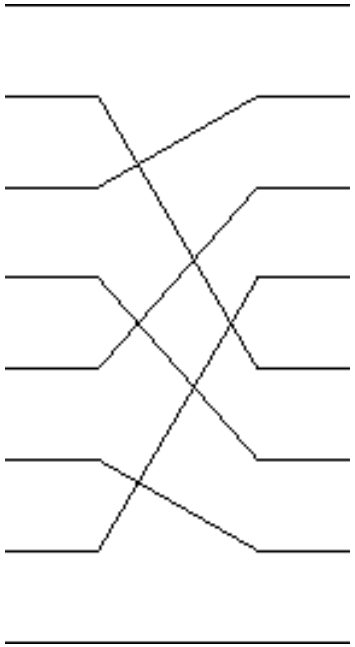
- Index Operations
  - `rev`
  - `riffle`
  - `unriffle`

```
riffle = halve ->-
shuffle
```



unriffle

```
unriffle = oeSplit ->- conc
```



Obsidian Programming

- Apply or Map: `fun`
- Sequential composition of programs: `->-`
- Parallel composition of programs: `parl`

Obsidian Programming: an example

```
rev_incr :: Arr (Exp Int) -> W (Arr (Exp Int))
rev_incr = rev ->- fun (+1) ->- sync
```

```
*Obsidian> execute rev_incr [1,2,3]
[4,3,2]
```

Obsidian Synchronisation

- Synchronisation primitive: `sync`
  - All array elements are updated after a `sync`
  - Only applicable at top-level
    - Inherits behavior from CUDA's `__syncthreads()`

Generating C Code

- Generate CUDA C Code for NVIDIA GPU
  - Executed as one block of threads
    - Pros
      - Communication and synchronisation possible
    - Cons
      - Upper limit of 512 threads per block
      - Does not use entire GPU

Generating C Code

- Each thread is in charge of calculating one array element
  - Limits array size to 512 elements
  - Leads to some redundancy
    - Swap operation performed by two threads in cooperation

Generating C Code

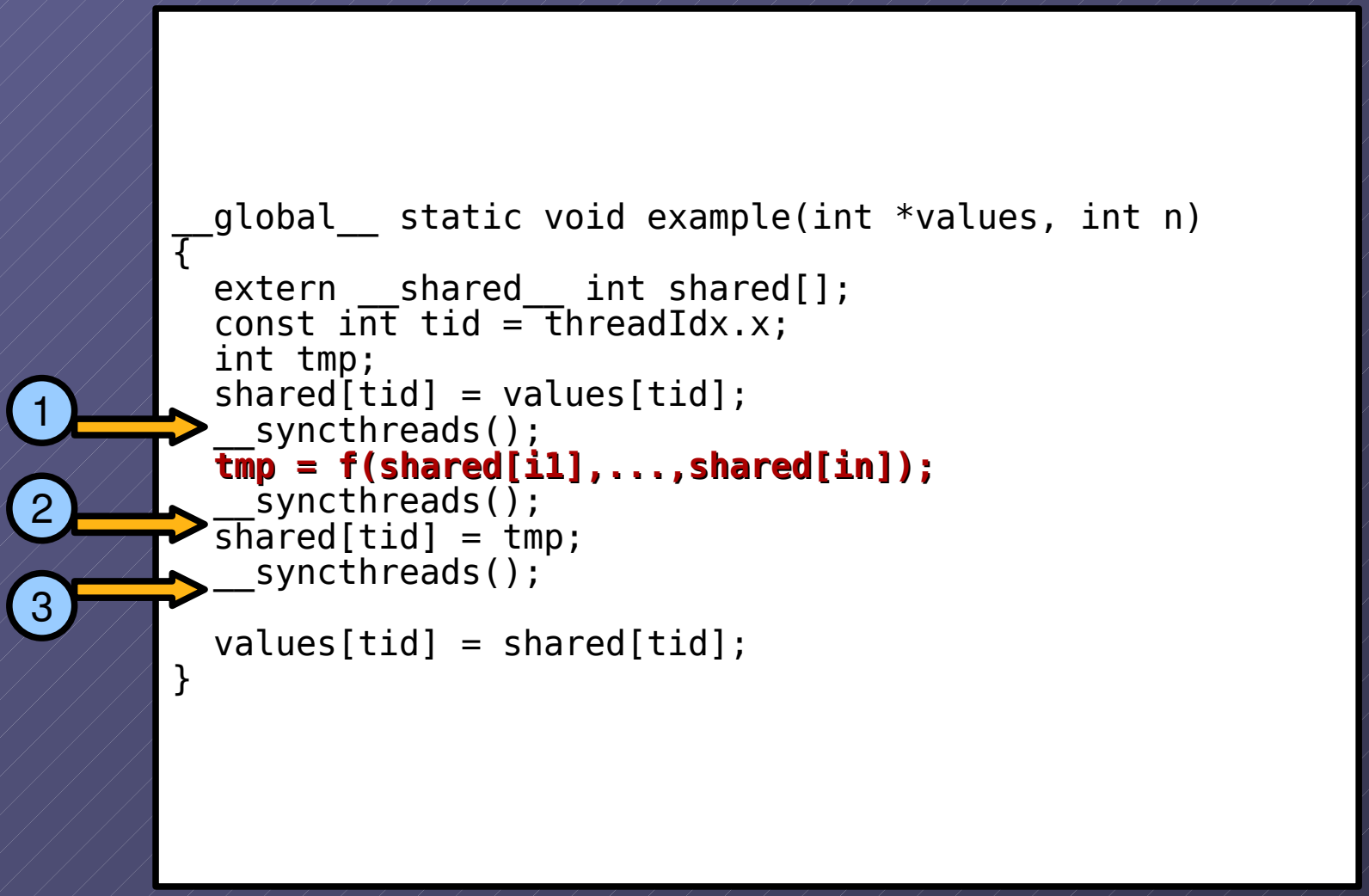
```
reverse = rev ->- sync
```

```
__global__ static void reverse(int *values, int n)
{
  extern __shared__ int shared[];
  const int tid = threadIdx.x;
  int tmp;
  shared[tid] = values[tid];
  __syncthreads();
  tmp = shared[((n - 1) - tid)];
  __syncthreads();
  shared[tid] = tmp;
  __syncthreads();

  values[tid] = shared[tid];
}
```

Generating C Code

```
__global__ static void example( int *values, int n)
{
  extern __shared__ int shared[];
  const int tid = threadIdx.x;
  int tmp;
  shared[tid] = values[tid];
  __syncthreads();
  tmp = f(shared[i1],...,shared[in]);
  __syncthreads();
  shared[tid] = tmp;
  __syncthreads();

  values[tid] = shared[tid];
}
```

Generating C Code




```
__global__ static void example(int *values, int n)
{
  extern __shared__ int shared[];
  const int tid = threadIdx.x;
  int tmp;
  shared[tid] = values[tid];
  __syncthreads();
  tmp = f(shared[i1],...,shared[in]);
  __syncthreads();
  shared[tid] = tmp;
  __syncthreads();

  values[tid] = shared[tid];
}
```

Implementing a sorter

A two-sorter sorts a pair of values:

```
cmpSwap op (a,b) = ifThenElse (op a b) (a,b) (b,a)
```

Sort each pair of elements in an array:

```
sort2 = (pair ->- fun (cmpSwap (<*)) ->- unpair ->- sync)
```

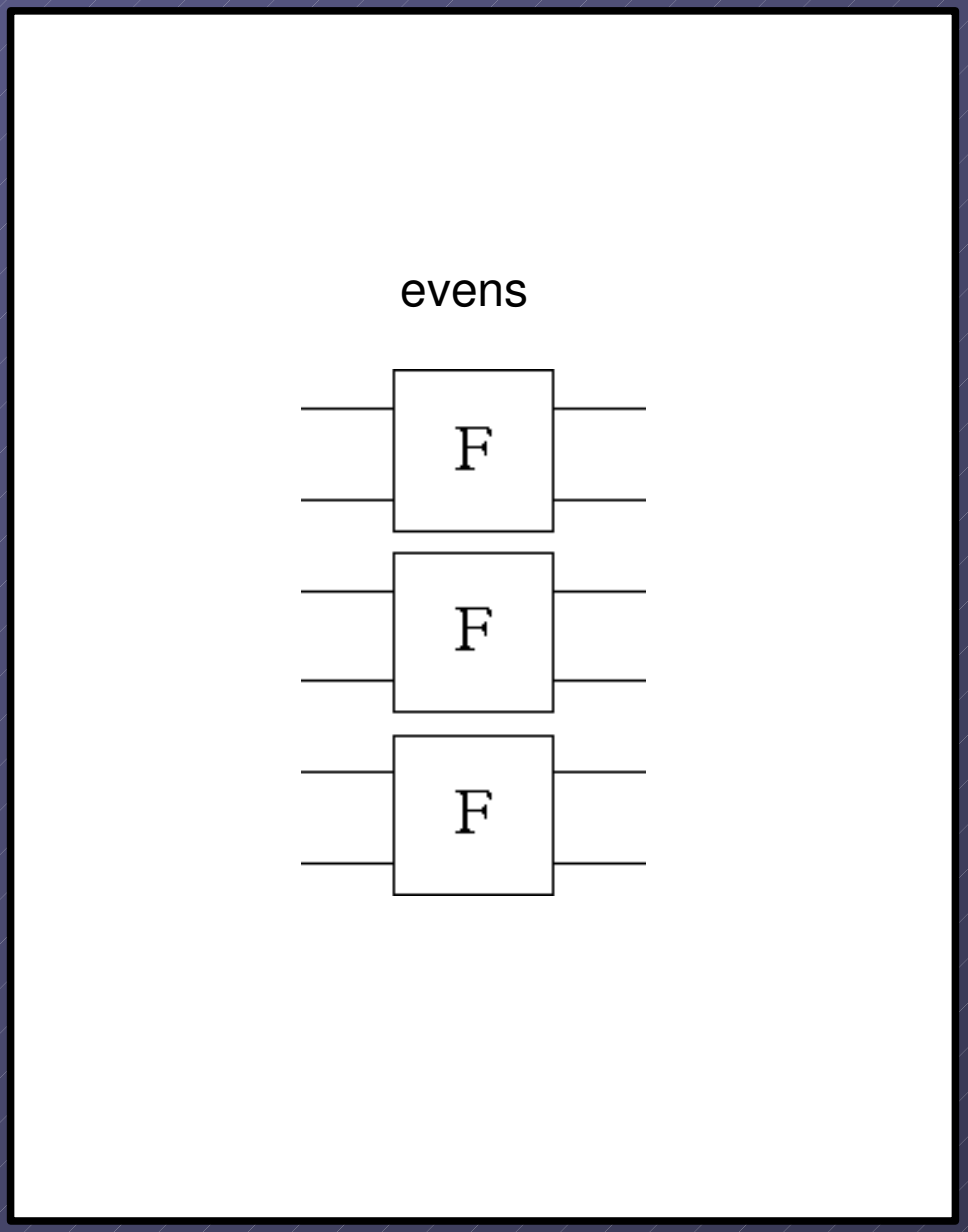
```
*Obsidian> execute sort2 [2,3,5,1,6,7]
[2,3,1,5,6,7]
*Obsidian> execute sort2 [2,1,2,1,2,1]
[1,2,1,2,1,2]
```

Implementing a sorter

A more efficient pairwise sort:

```
sortEvens = evens (cmpSwap (<*)) ->- sync

*Obsidian> execute sortEvens [2,3,5,1,6,7]
[2,3,1,5,6,7]
*Obsidian> execute sortEvens [2,1,2,1,2,1]
[1,2,1,2,1,2]
```

Implementing a sorter

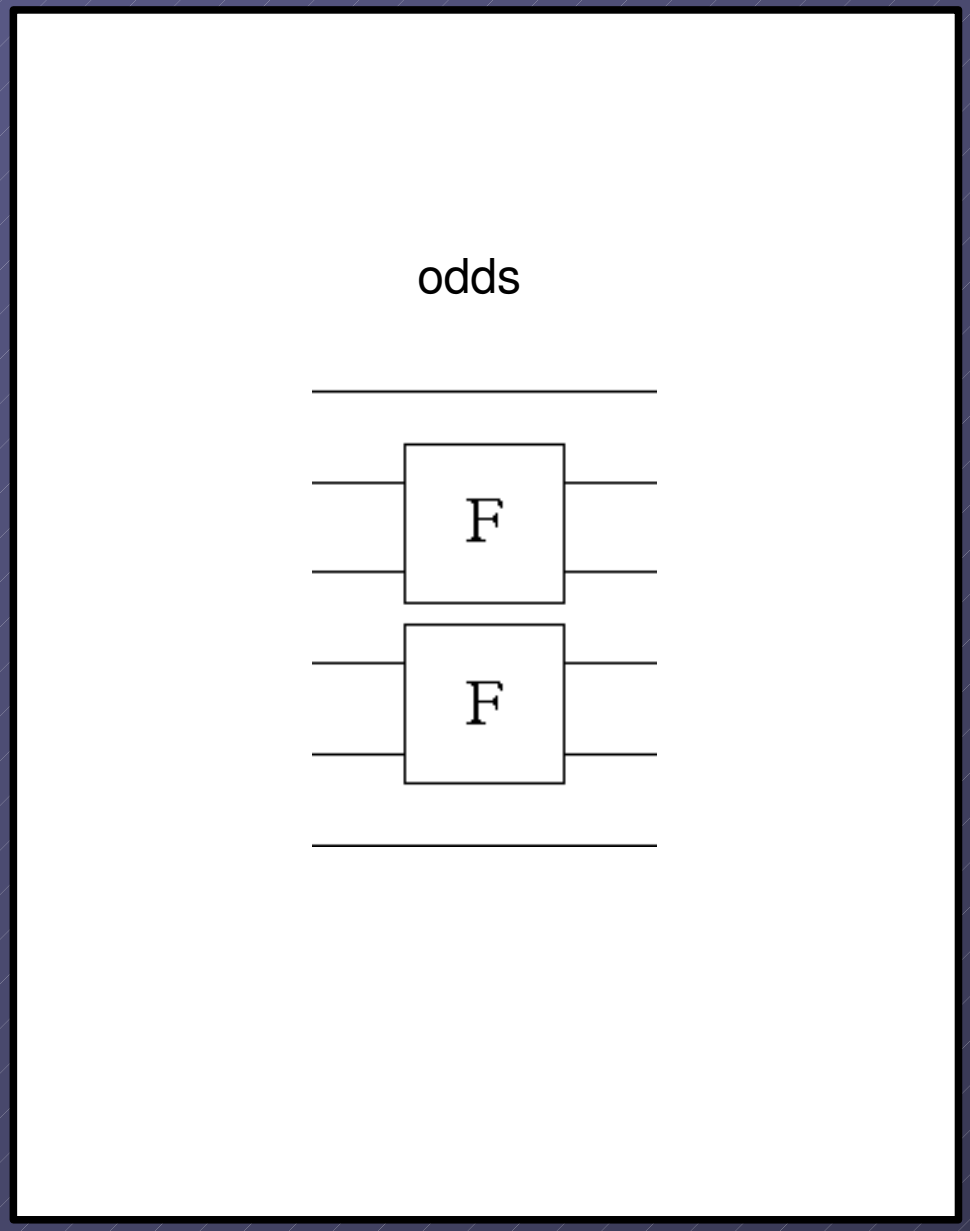


Implementing a sorter

A close relative of `evens` is `odds`:

```
sortOdds = odds (cmpSwap (<*)) ->- sync

*Obsidian> execute sortOdds [5,3,2,1,4,6]
[5,2,3,1,4,6]
*Obsidian> execute sortOdds [1,2,1,2,1,2]
[1,1,2,1,2,2]
```

Implementing a sorter

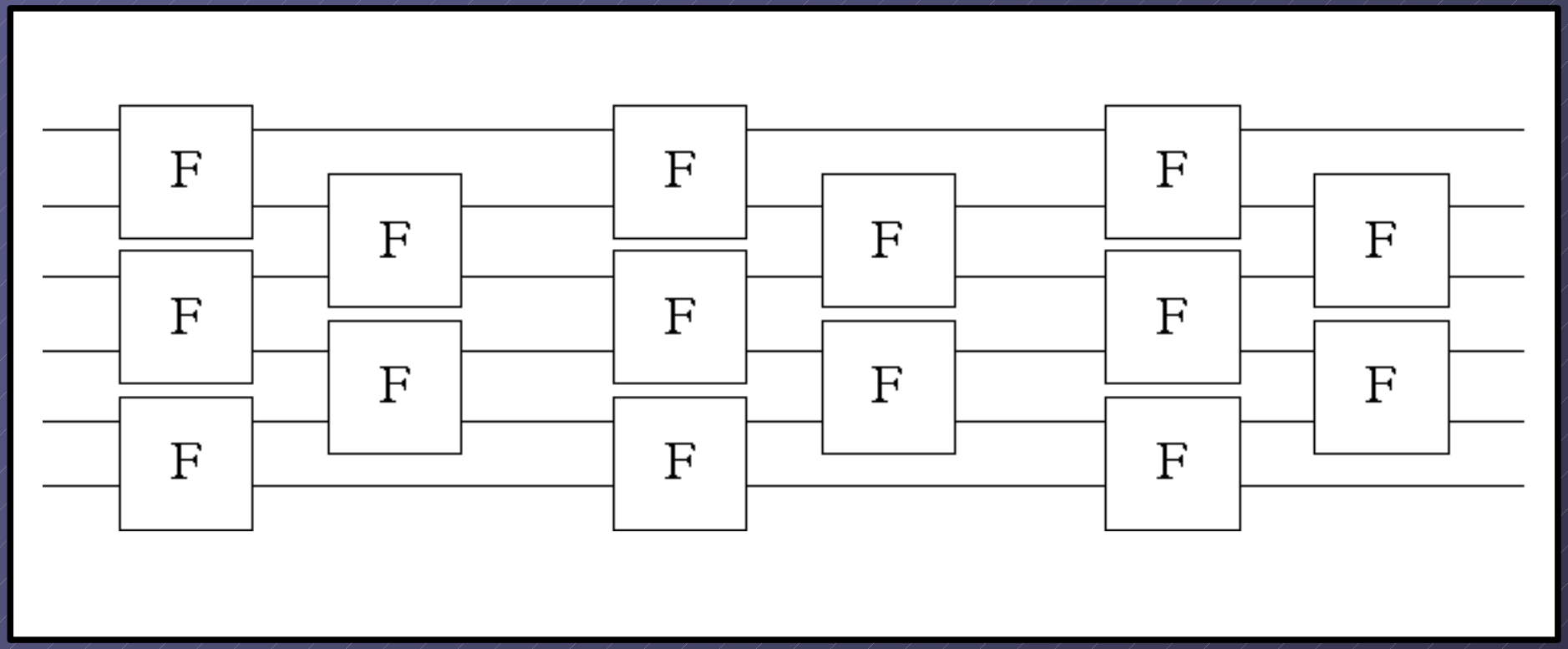


Odd Even Transposition Sort

Sorter implemented using odds and evens:

```
sortOETCore = sortEvens ->- sortOdds

sortOET arr =
    let n = len arr
    in  (repE (idiv (n+1) 2) sortOETCore) arr
```

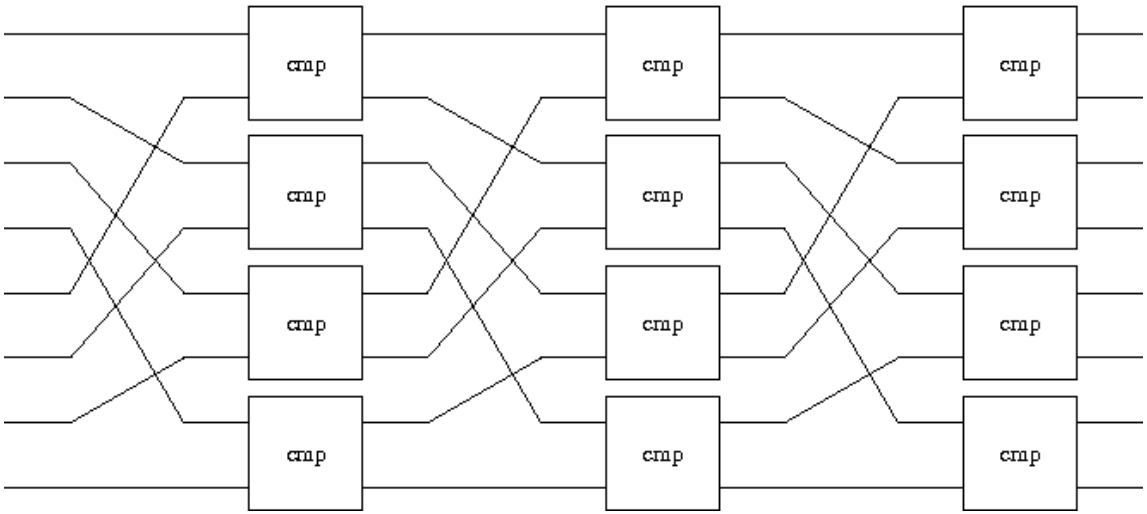
Odd Even Transposition Sort



VSort

- Another iterative sorter
- $\log^2(n)$ depth
- Built around a *shuffle exchange network:*

```
shex f n = rep n (riffle ->- evens f ->- sync)
```



VSort

Merger implemented using shex:

```
bmergeIt n = shex (cmpSwap (<*)) n
```

```
*Obsidian> execute (shex (cmpSwap (<*)) 3) [2,4,6,8,7,5,3,1]
[1,2,3,4,5,6,7,8]
```

VSort

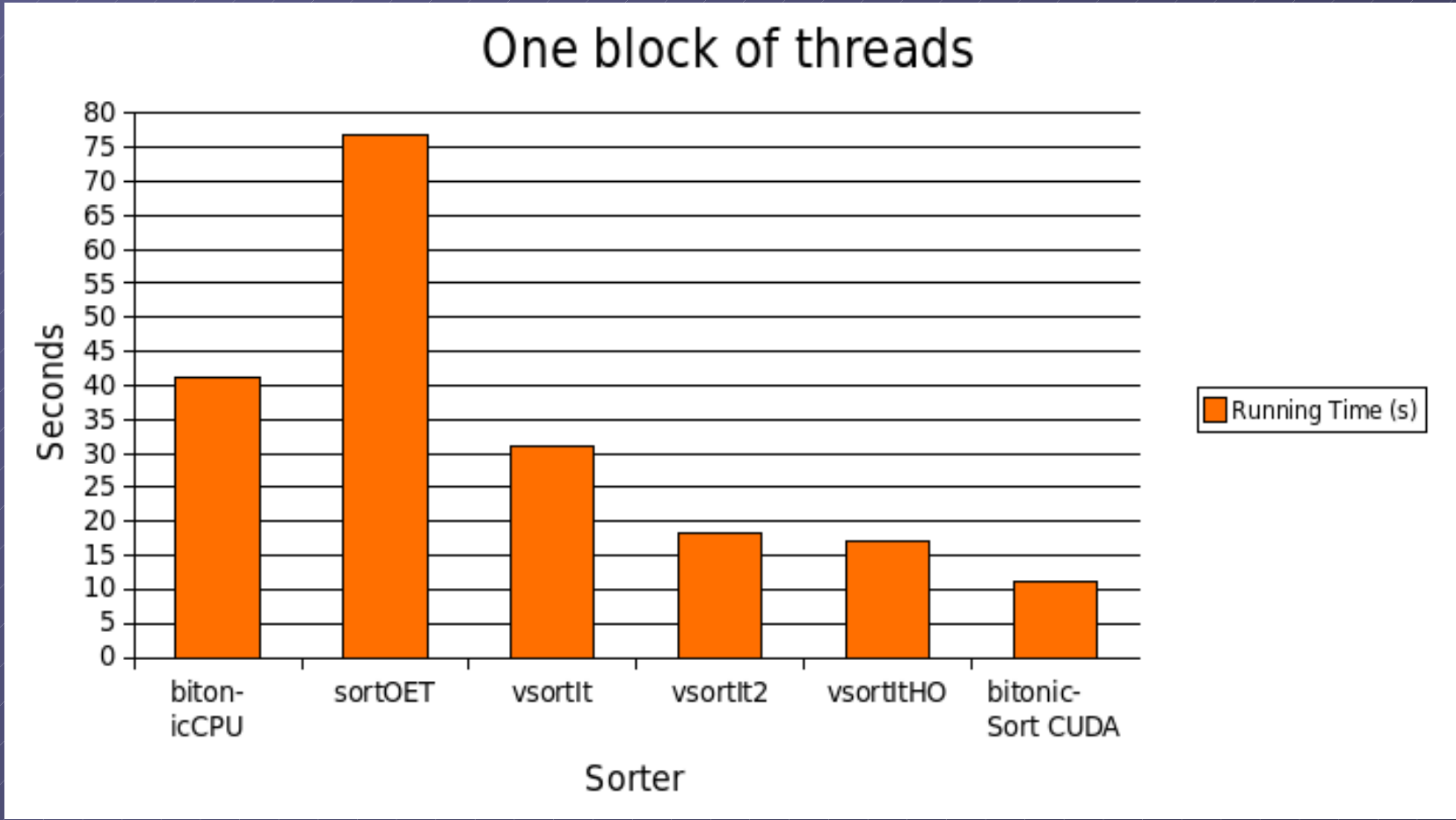
Sorter implemented using `bmergeIt`:

```
vmergeIt n = tblLook tautab ->- sync —>- bmergeIt n

VsortIt n = rep n (vmergeIt n)
```

Comparison of sorters

- Six different sorters
  - Bitonic sort on CPU
  - Odd Even Transposition sort
  - Three versions of VSort
  - CUDA Bitonic sort on GPU
- Data and Hardware
  - 288 Mb of random data
  - CPU: 2.4GHz Intel Core 2
  - GPU: 1.2GHz NVIDIA 8800 GTS (shader clock)

Comparison of sorters



Related work

- Pan
  - Embedded in Haskell
  - Image synthesis
  - Generates C code
- Vertigo
  - Embedded in Haskell
  - Describes *Shaders*
  - Generates GPU programs

Related work

- PyGPU
  - Embedded in Python
  - Uses Pythons introspective abilities
  - Graphics applications

Related work

- NESL
  - Functional language
  - Nested data-parallelism
  - Compiles into VCode
- Data Parallel Haskell
  - Nested data-parallelism in Haskell

Future work

- Solve the recursion dilemma
  - Enable the description of recursive sorters
    - Bitonic Sort
- Make use of entire GPU
- Optimise the generated code
- More generality
  - Not just sorters
- Other target platforms

Future work

- More generality
  - Arr a -> Arr b (not just Arr Int -> Arr Int)
  - Matrices
  - Pairs of arrays to arrays
  - Arrays of pairs to arrays
  - Throw away length homogeneity demand