The Dataflow Computing Model and Dataflow Architectures
The Computing Model Crossroad

- Computers today are based on the von-Neumann model
  - Program counter
  - Fetch/Decode/Execute
  - Simple LD/ST memory semantics

- Inherently sequential model
  - Parallelism achieved using threads
  - Have you ever tried to program a parallel von-Neumann machine?

- Inefficient power distribution...
Von-Neumann Power Bottleneck

- NVIDIA Tesla GTX280
- Extreme, parallel von-Neumann architecture
- 65nm process
- Taken from: Hong & Kim, “An Integrated GPU Power and Performance Model”, ISCA 2010
Von-Neumann Power Bottleneck

- Memory
  - ~60W
Von-Neumann Power Bottleneck

- Memory
  - ~60W
- Exec. Overheads
  - ~12W
Von-Neumann Power Bottleneck

- Memory
  - ~60W
- Exec. Overheads
  - ~12W
- Register file
  - ~5W
Von-Neumann Power Bottleneck

- Memory
  - ~60W
- Exec. Overheads
  - ~12W
- Register file
  - ~5W
- Computation
  - <10W !!!
Existential questions

• What is the function of a register file?

• What is the function of memory?

• Should the computation be controlled by a program counter?

• What is the best way to represent a computation?
  • Instruction stream? Graph?
The Dataflow Computing Model

[Karp & Miller, 1966]

Computations are expressed as a graphs

\[(2 + 3) \times (5 + 6)\]
The Dataflow Computing Model

*The firing rule:* A node can “fire” (execute) as soon as all its inputs are ready, and its output port is free.

\[(2 + 3) \times (5 + 6)\]
The Dataflow Computing Model

- Graph is built by composing expressions
- *Every sub-graph computes a value*

\[(2 + 3) \times (5 + 6)\]

Diagram: [Diagram of a dataflow graph with expressions and an arrow indicating the computation process.]
von-Neumann vs. Dataflow

While \((a < N)\)

\[ a = (a+b) \times (a-10); \]

**Statements**

- load r0, (a)
- load r1, (b)
- loop1:
  - add r2, r1, r0
  - sub r0, r0, 10
  - mul r0, r0, r2
  - cmp r0, N
  - jl loop1
- store (a), r0

**Expressions**

```
node = actor = instruction
arc = data path = operand
```
Dataflow vs. von-Neumann: Variables

• Von-Neumann:
  • Variable names a persistent location

\[ C = A + B \]

• Dataflow:
  • Variables names a value (write/read once)

\[ C = A + B \]
Why Dataflow?

• Implicit parallelism
  • Performance
  • Too much parallelism at times...
    • How to schedule?

• Implicit data dependencies
  • No locks

• Mathematical reasoning
  • Critical path
  • Maximum span
Static Dataflow Architecture

[Dennis, 1974]
What About Loops?

While \((a < N)\)
\[
a = (a+b) \times (a-10);
\]

```
PC  ➔
load r0, (a)
load r1, (b)
loop1:
  add r2, r1, r0
  sub r0, r0, 10
  mul r0, r0, r2
  cmp r0, N
  jl loop1
  store (a), r0
```

*node = actor = instruction
arc = data path = operand*
Dynamic Dataflow: Tagged Tokens

Gurd et al. 1978

For (i=1 to N)
\[ S_i = (a_i \times b_i) + (c_i \times d_i) \]
Dynamic Dataflow

*Token Store & Token Matching*

- The token store maps newly generated data items to their consumers
  - New instructions added dynamically
- Issues:
  - Every instruction can reside anywhere in the token store
  - Instructions added out-of-order
  - Possible deadlocks
Dynamic Dataflow: The Token Store

[Gurd et al. 1978]

\[
\begin{array}{ccccccccc}
ai & bi & ci & di & ei & fi & ej & fj & aj & bj & cj & dj \\
\hline
X_1i & X_2i & +i & +j & X_1j & X_2j
\end{array}
\]

Requires a large associative structure

**Associative + Large = Expensive**

**Execution**
Programming Dataflow Machines

• Typically use functional or Dataflow langs.
  • *SISAL*  [Manchester, early 1980s]
  • *Id*     [MIT, early 1990s]

• Computation has no side-effects
  • No side-effects -> no state
  • No state -> no persistent data structures
Programming Without Side-Effects

\[
\text{insert}(x, \text{lst}) := \\
\begin{align*}
\text{if } (x \leq \text{head(lst)}) \text{ then } & (x, \text{lst}) \quad \text{// HEAD} \\
\text{else } & (\text{head(lst)}, \text{insert}(x, \text{tail(lst)})) \quad \text{// TAIL}
\end{align*}
\]

- Reiterate: variables name values
  - Data structures continuously copied around
- What about imperative langs.? (C / C++)
Explicit Program State: Memory

• That’s easy – no side effect implies that: **Dataflow has no notion of memory**

• Why?
  
  Memory hides data dependencies

• Example

  \[*\text{ptr1} = a + 15\]
  
  \[c = *\text{ptr2} - b\]
Explicit Program State: Memory

- What is the semantically correct order?
  
  **RaW:**  \[ A \rightarrow D \& B \rightarrow C \]
  
  **RaW & WaR:**  \[ A \rightarrow C \rightarrow B \rightarrow D \]

- Loads must wait until stores are committed
- And vice versa...
Memory Deps: *I-Structure*  
[Arvind et al. 1983]

- Write-once memory
- Loads are blocked if data not written yet

RaW:  
\[
A \rightarrow C \& B \rightarrow D 
\]

RaW & WaR:  
\[
A \rightarrow C \rightarrow B \rightarrow D 
\]
Memory Deps: *M-Structure*

[Barth et al. 1991]

- Full/empty bits
- Memory must be read before written to

RaW:  
A -> C & B -> D

RaW & WaR:  
A -> C -> B -> D
In-Memory Data Dependencies

• Problem: Imperative languages rely on memory ordering to maintain dependencies
  \[
  \*ptr1 = a + 15 \\
  c = \*ptr2 - b 
  \]

• Dataflow has no inherent memory ordering

• How can the two be resolved?
  • Open question...
Memory Ordering in WaveScalar

[Swanson et al. 2003]

\[
a = *\text{ptr1} \quad \text{// LD: memop 1}
\]

if(cond) {
    \[
    *\text{ptr2} = a + 5 \quad \text{// ST: memop 2}
    \]
} else {
    \[
    *\text{ptr3} = a - 5 \quad \text{// ST: memop 3}
    \]
} 

\[
b = *\text{ptr4} \quad \text{// LD: memop 4}
\]

• Code may execute out-of-order
• Memory operations must commit in order!
Memory Ordering in WaveScalar

[Swanson et al. 2003]
Out-of-Order Processors

• Dynamically builds the dataflow graph for a window of instructions
• Identifies parallelism in a sequential stream
• Referred to as "Restricted Dataflow"
• Inspired by Tomasulo’s algorithm
• Nowadays, present in most GP processors

[Patt et al. 1985]
Out-of-Order Procs. (the good)

[ Patt et al. 1985 ]

- Inst. added in-order
- No deadlocks

- Register renaming creates inst. graph
- Tagged token store (phys. reg = tag)

- In-order commit
- Preserves memory order (via LSQ)
Out-of-Order Procs. (the bad) [Patt et al. 1985]

• Does not affect memory semantics
  • Dataflow contained in the execution pipeline
• Non-scalable, associative token store
  • Broadcast-based bypass network
  • All RSs notified after each instruction
• No control flow variability
  • Aggressive speculation
  • Requires good branch predictors
  • Rollback on mispredictions
SGMF – Single Graph Multiple Flows

[Voitsechov, Etsion 2013]

• Dataflow architecture for massively parallel execution models

• Execute many threads in parallel by driving multiple flows through the computation graph - Every thread composes a flow through the graph

• Once the input token is sent a new thread enters the grid - pipelining

• Blocked thread can be bypassed by a ready one – dynamic dataflow

• Total parallelism is achieved by the combination of both methods
  • Parallelism = pipelining + dynamic dataflow
SGMF – Execution Model

- **CUDA** by Nvidia
- All threads in execute the same kernel program
- Threads have **thread id** numbers within block

```c
__global__ void KernelFunc(int* a, int* b, int* c)
{
    if (threadIdx.x > 100)
        c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
    else
        c[threadIdx.x] = a[threadIdx.x] * b[threadIdx.x];
}
KernelFunc<<< ... >>>(...);
```

- Massive parallel execution model
- **SIMD Execution** in each node
- **Multithreaded Execution** across the grid
Execution Overview: Pipelining

- Parallelism is determined by critical path through the graph
Execution Overview: Dynamic Dataflow

- Pipelining enables parallelism on the critical path, but is insufficient.

- Memory may stall a specific flow and block all the succeeding threads.
  - Reducing the utilization of the functional units.

- Solution: dynamic dataflow
  - Overcome memory latencies by interleaving execution of graph flows.
  - Similar to in-order vs. out-of-order processors.
Load Operation Bypassing

- Dynamic dataflow – enables bypass of memory instructions
  - May cause hardware deadlocks
  - Requires token matching mechanism
Execution Overview: Dynamic Dataflow

- Each flow is associated with a token
- Execute the operation when tokens match
- Parallelism is determined by the number of tokens in the system
Design Issues: Preventing Deadlocks

- Imbalanced out-of-order memory responses may trigger deadlocks

**Solution:** *load-store units limit bypassing to the size of the token buffer*
Design issues: Executing Branches

- Code branches are assigned to functional units spatially rather than temporally

```plaintext
if (cond)
  a = b + c
else
  a = b - c
```

- Supports diverging branches
  - But requires more functional units
The SGMF Architecture

• Coarse grained reconfigurable architecture

• Designed for heavy multithreading
  • A grid of computational and control units surrounded by LDST units on the perimeter
  • The nodes on the grid are connected by an interconnect implemented by reconfigurable switches
Architecture overview

- Heterogeneous grid of tiles
  1. Compute tiles: very similar to CUDA cores
  2. LD/ST tiles: buffer and throttle data
  3. Control tiles: pipeline buffering and join ops.
  4. Special tiles: deal with non-pipelined operations

- Reference point:
  - A single grid is the equivalent of a single NVIDIA Streaming Multiprocessor (SM)
  - Total buffering capacity in SGMF is less than 30% of that of an NVIDIA Fermi register file
Architecture overview
Interconnect

- Switches are connected using a folded cube

[Properties and performance of folded hypercubes., El-Amawy et al.]

- 8 “almost-NN”
- Static Switching
- Determined at compile time
Execution Model

- CUDA kernels are compiled as data-flow graphs
- Mapped to the grid
- NoC configured for the kernel
- Threads are fed to the grid in a pipelined manner
- Each thread in the kernel has its own ID
- Loops – handled during execution, each iteration acts as a thread
Dataflow: verdict?

• Exposes huge amounts of parallelism
• Holds great performance (power?) potential
• Ubiquitously used in OoO processors
  • Albeit in a very wasteful form

• Can we improve programmability?
• Can we improve architectures?
• Can we improve OoO processors?
Questions?