High Speed Query Execution with Accelerators and C++

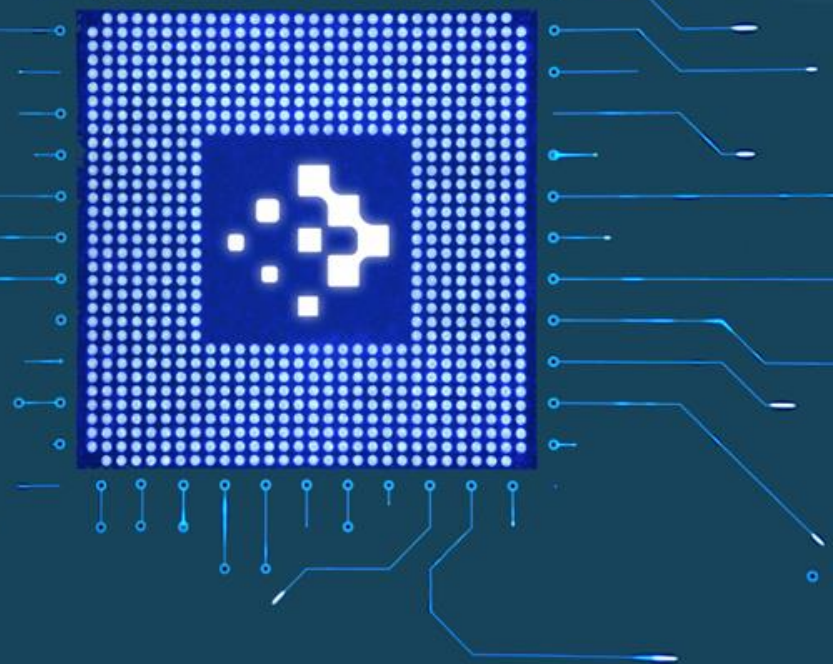
ALEX DATHSKOVSKY

About Me:

SPEEDATA

alex.dathskovsky@speedata.io

www.linkedin.com/in/alexdathskovsky



Agenda

- Why do we need Accelerator for Analytics?
- Architectural Talk :
  - History of computer architecture
  - How does APU(MFC) fits in
  - What is so special about our APU(MFC)
- How to code it ?
  - Process explanation
  - Code example for simple query
  - Code example for complex query and scale

Agenda

- APU – Analytic Processing unit
- MFC – Multi Flow Core

Why Do We Need Query Acceleration?

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

- Big Data and Analytics become more and more costly
  - Big data become really Big
- Calculation may get really long (days on multiple nodes)
- Energy consumption is a huge factor
  - CPU energy
  - Cooling the data centers
- Cost
  - More Data → More Servers → More Energy → More Expenses

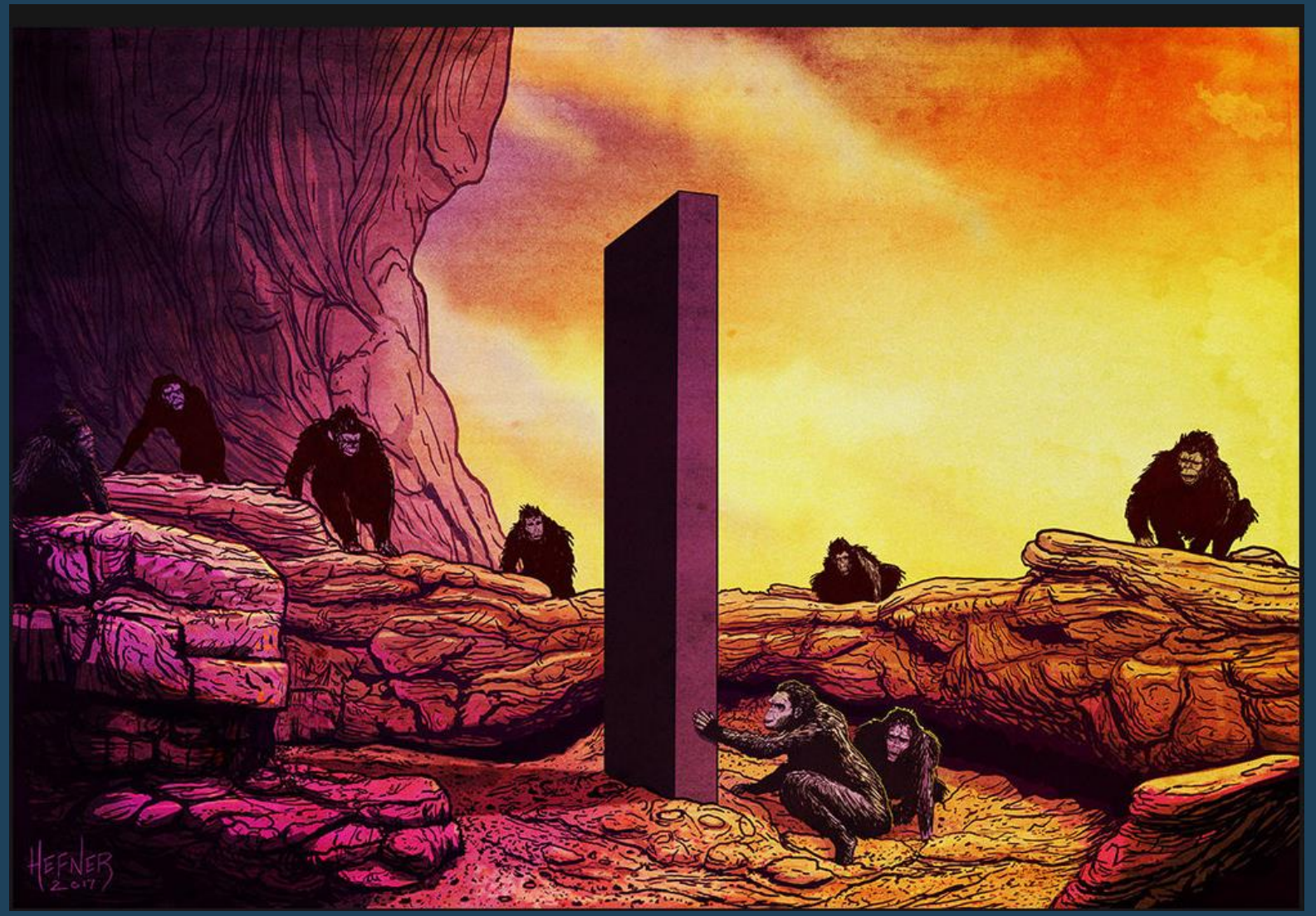
History Of Computer Architecture

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

In The Beginning:

In The Beginning:

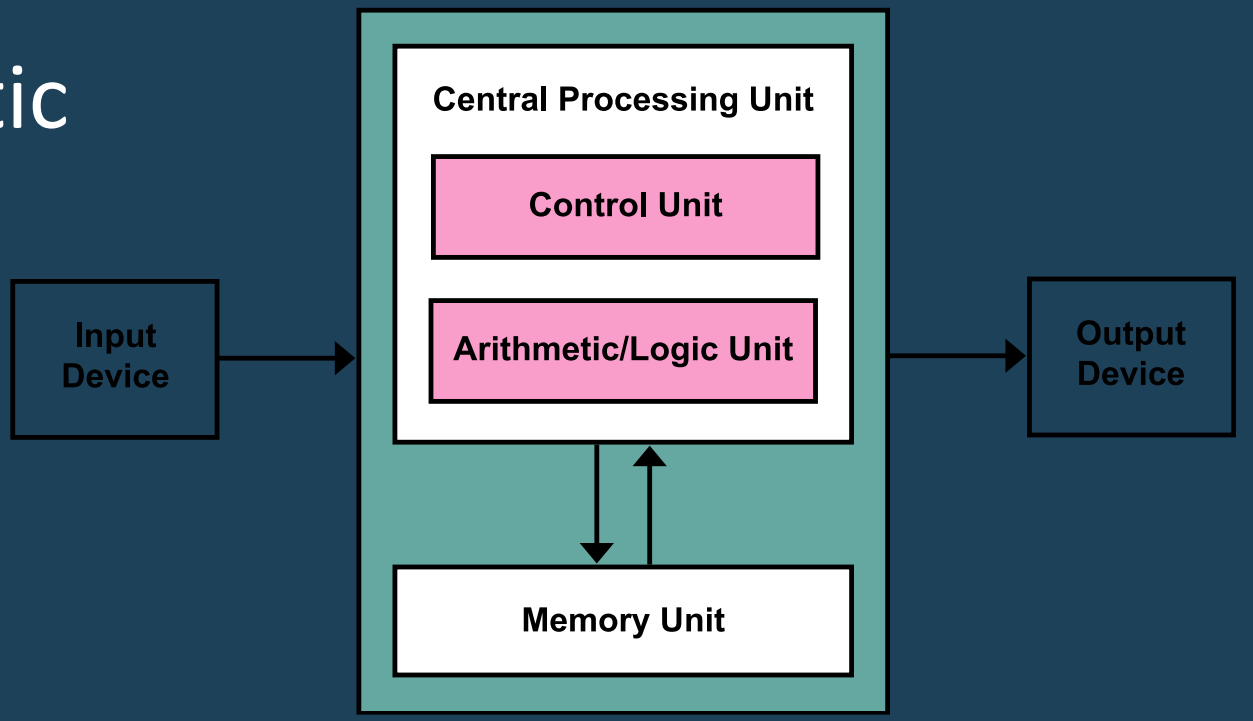


Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

Von Neumann Architecture

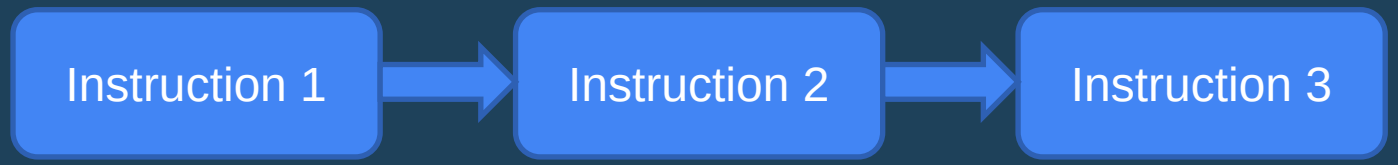
- Processing unit with arithmetic logic unit and registers
- Control Unit with a program counter
- Memory
- External mass storage
- I/O



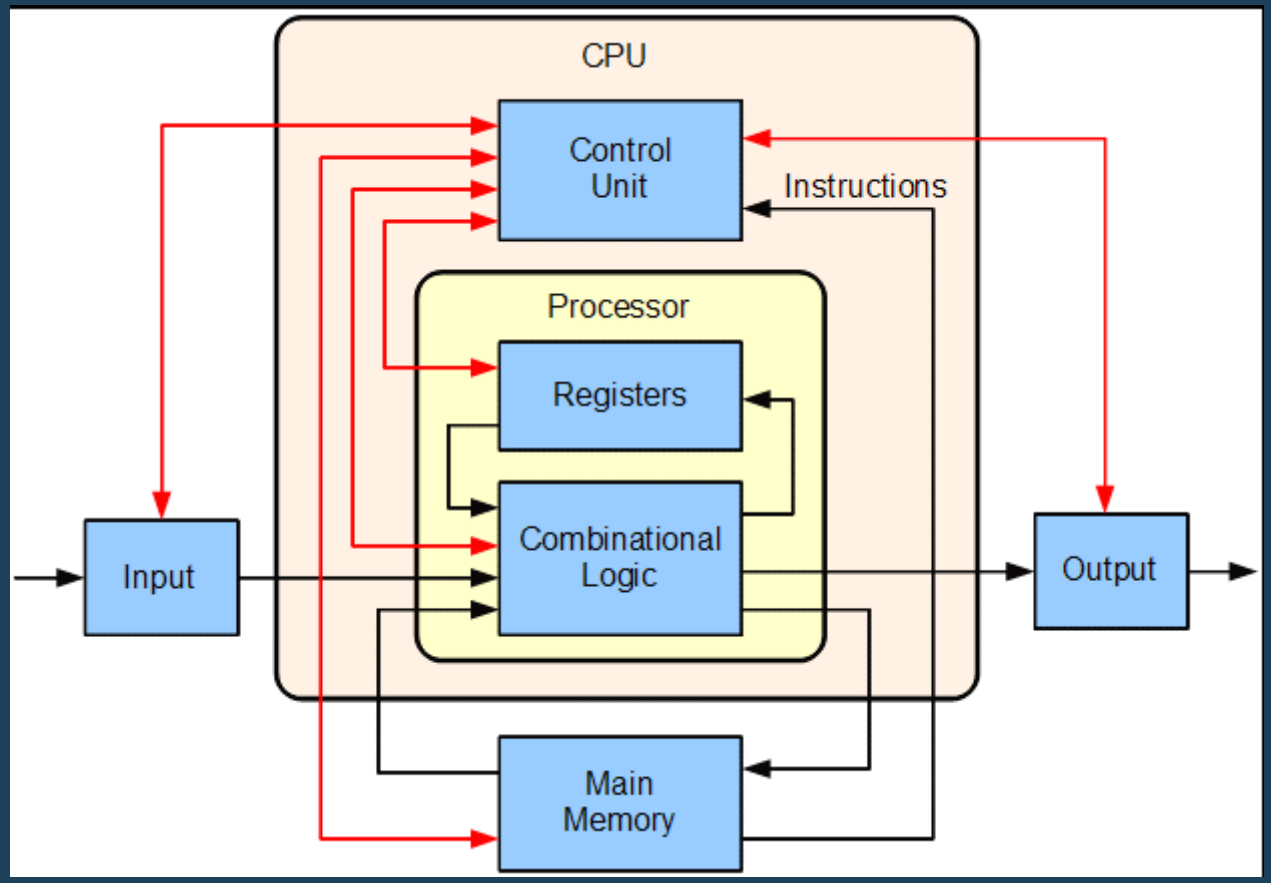
Source: https://en.wikipedia.org/wiki/Von_Neumann_architecture

CPU

- Executes Instruction sequentially
- Sequence of instructions called a program



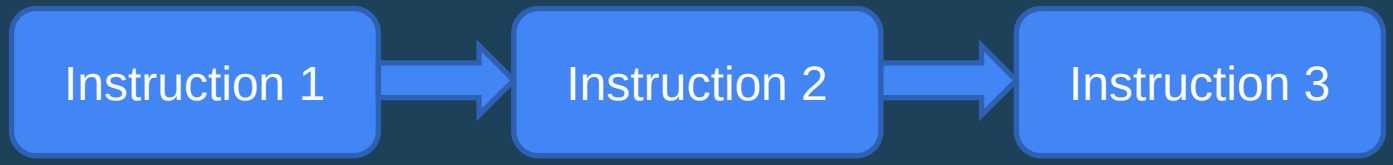
Simple CPU Diagram



Source: https://en.wikipedia.org/wiki/Central_processing_unit

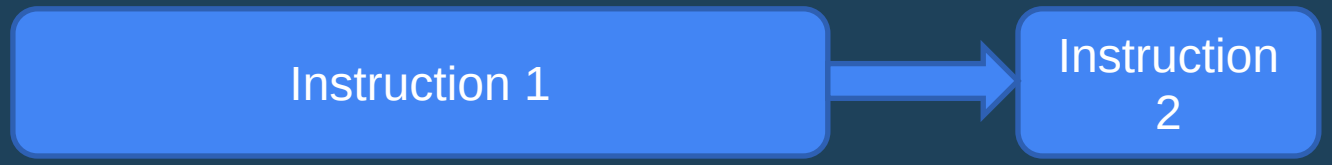
CPU Types : RISC AND CISC

- RISC – Reduced Instruction Set
  - Fixed length instructions
  - Single operation instruction (ADD, SHL, MUL)



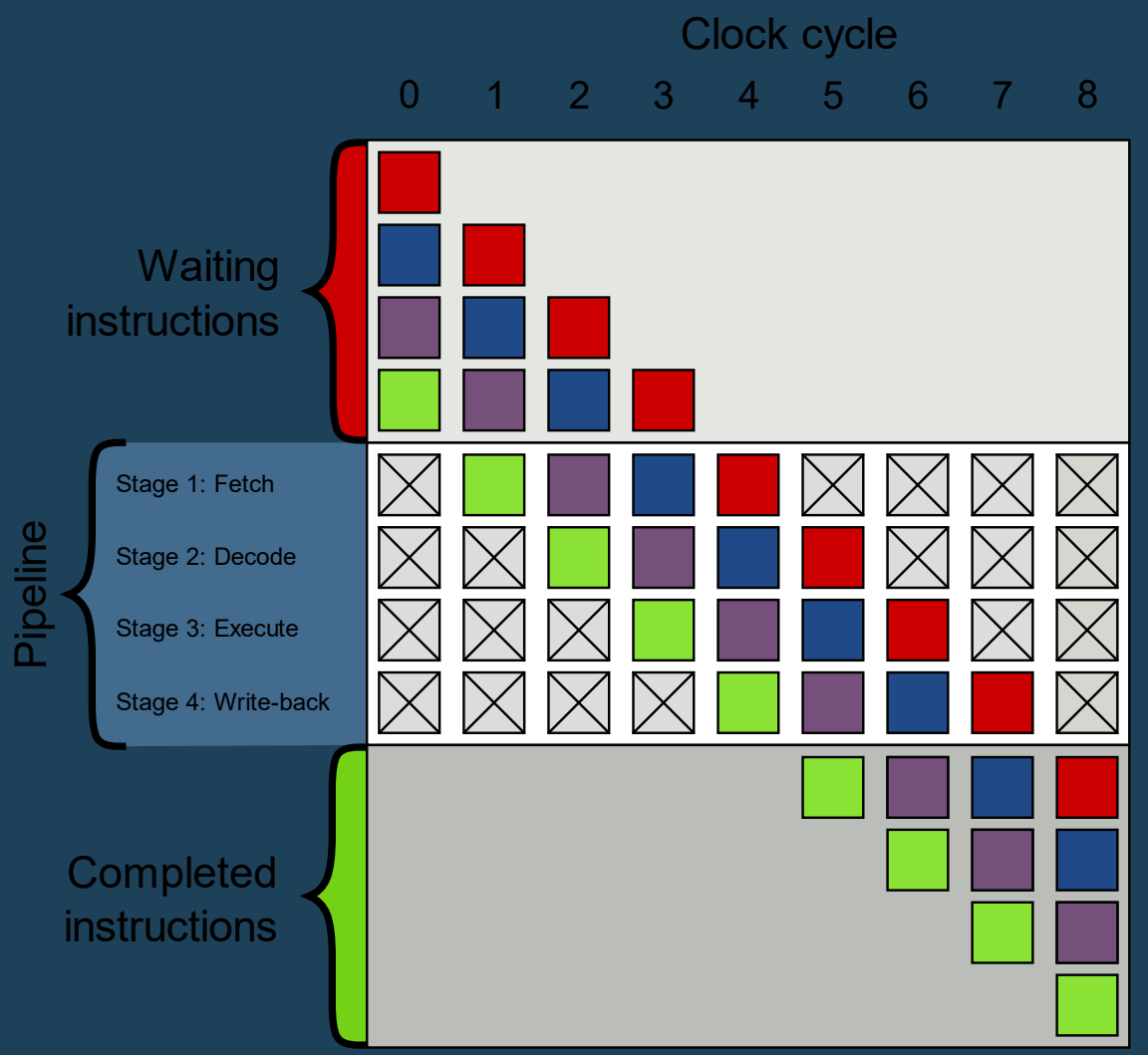
CPU Types : RISC AND CISC

- CISC – Complex Instruction Set Computing
  - Variable length instructions
  - Complex Operation instructions (LOAD + ADD + STORE)



Computation Gets More Complex Or Please Be Faster

- Piped Execution
  - Run more then one instruction
  - May stall



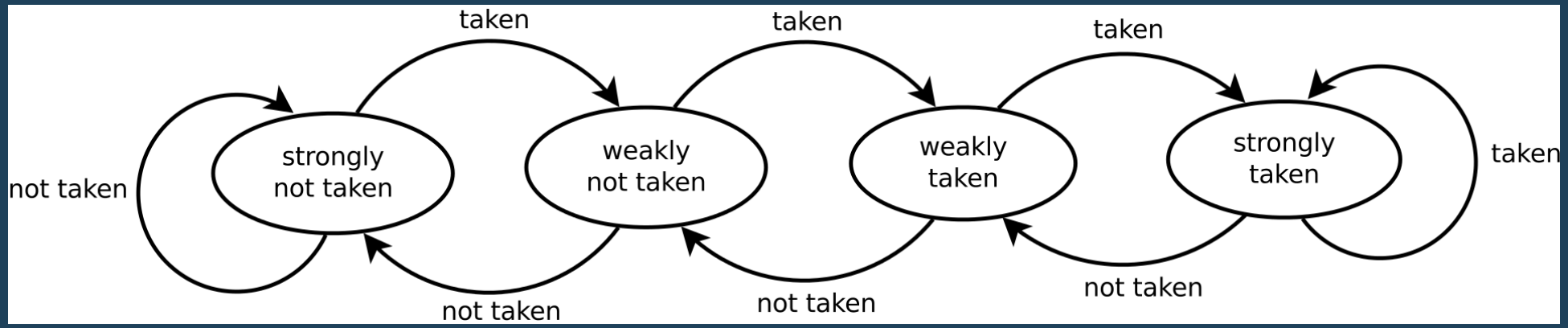
Source: https://en.wikipedia.org/wiki/Instruction_pipelining

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

Computation Gets More Complex Or Please Be Faster

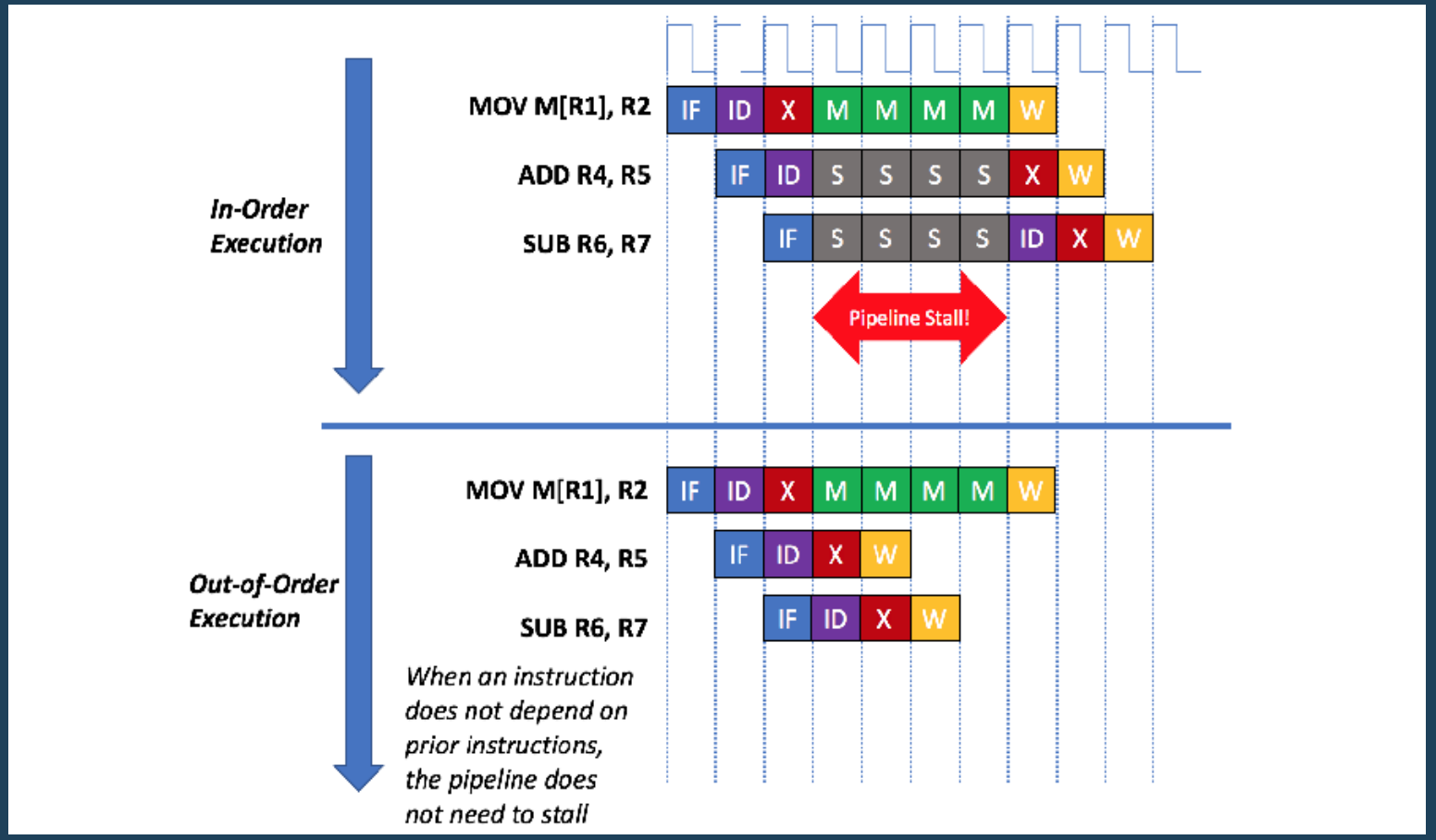
- Branch Prediction
  - Predicts if branch is taken
  - If prediction is wrong flush calculation
  - Uses branch history



Source:: https://en.wikipedia.org/wiki/Branch_predictor

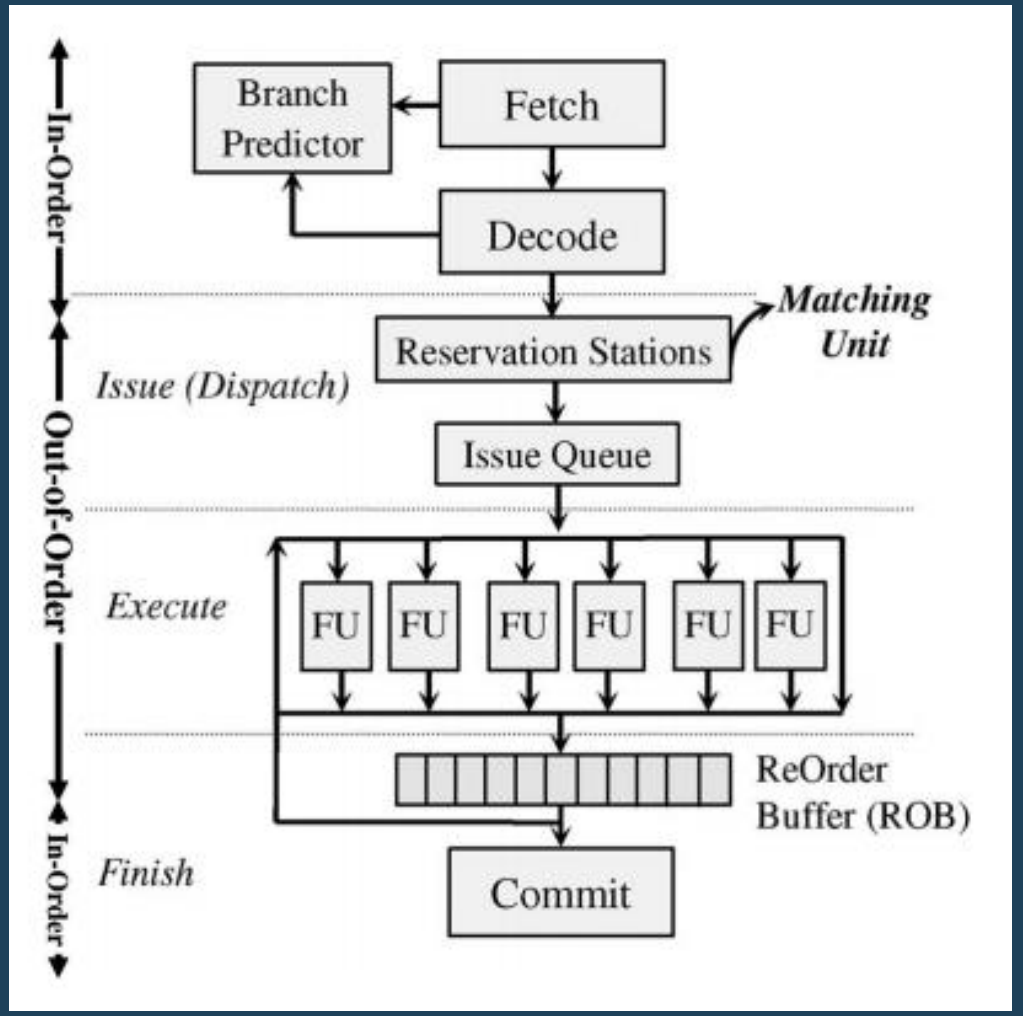
Computation Gets More Complex Or Please Be Faster

- Out Of Order
  - Some instruction may be reordered
  - Less stalling
  - Cannot reorder if RAW



Source:: https://www.semanticscholar.org/paper/RISC-V-Reward:-Building-Out-of-Order-Processors-in-Zekany-Tan/f7f6d27f334604c3c85f0b8d21d2a9b4df22a983

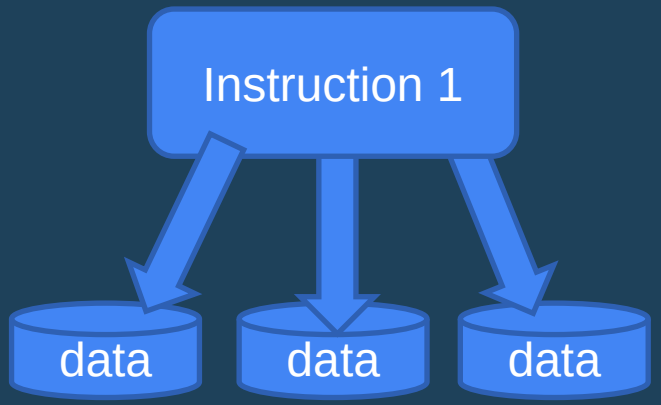
Computation Gets More Complex Or Please Be Faster



From F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez and Y. Etsion, "Hybrid Dataflow/von-Neumann Architectures," in IEEE Transactions on Parallel and Distributed Systems

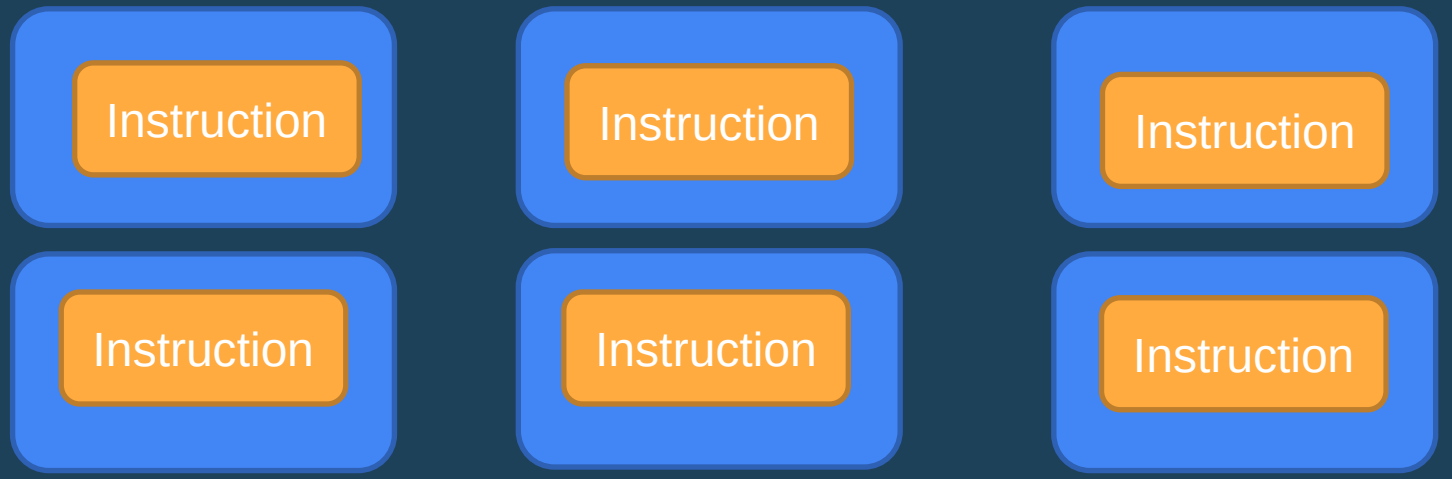
Computation Gets More Complex Or Please Be Even Faster

- Processing more data by each instruction
- Single Instruction Multiple Data
- Vector Machines

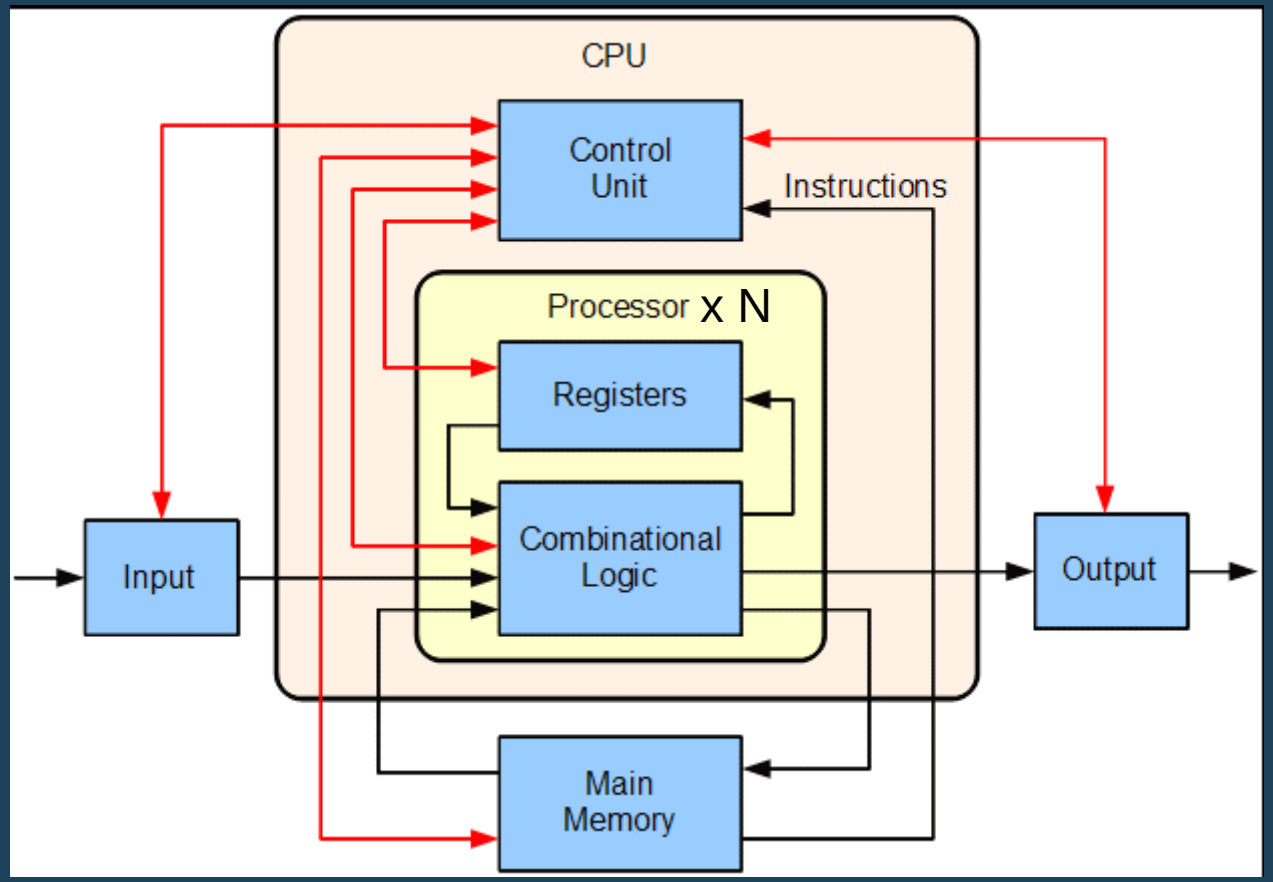


GPUs Do It Better (Sometimes)

- Built to accelerate parallel application especially video games
- GPGPU changed it from video games to general parallelism
- Single Instruction Multiple Threads



Simple GPU Diagram



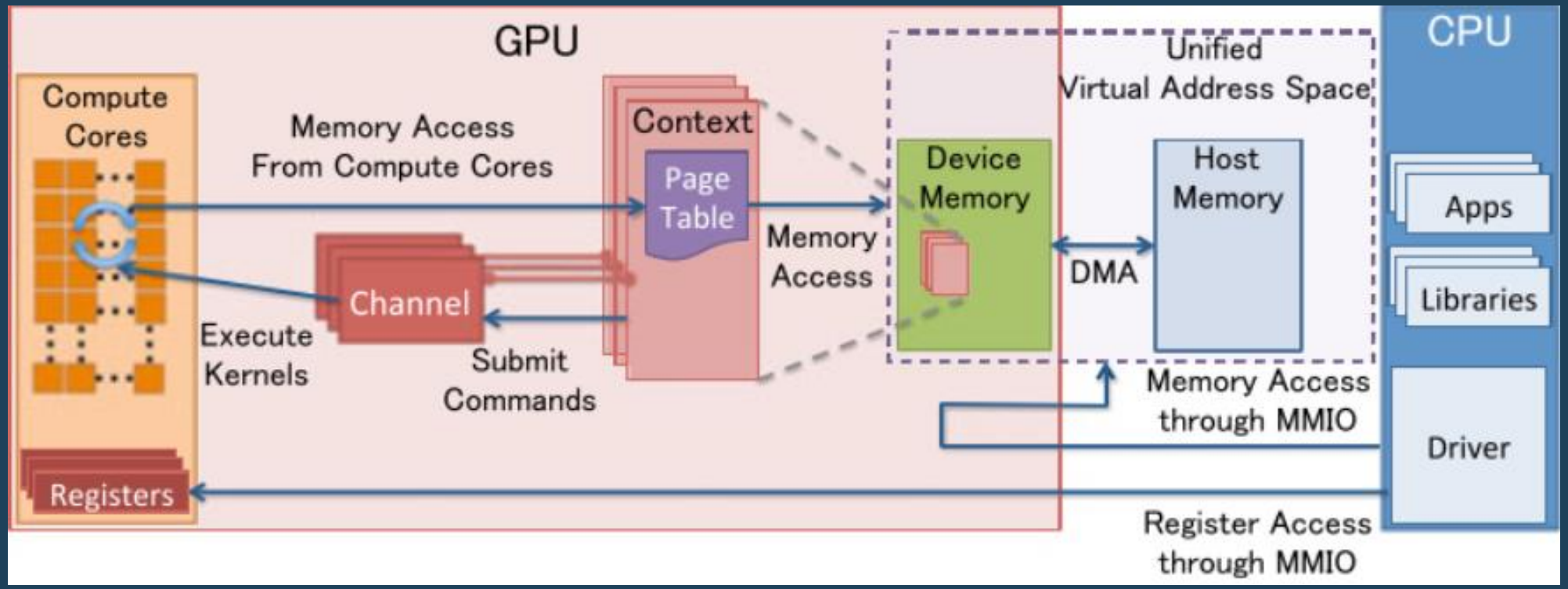
GPU Evolution

GH100 GPU with 144 SMs

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

GPU System Architecture



Source: https://insujang.github.io/2017-04-27/gpu-architecture-overview/

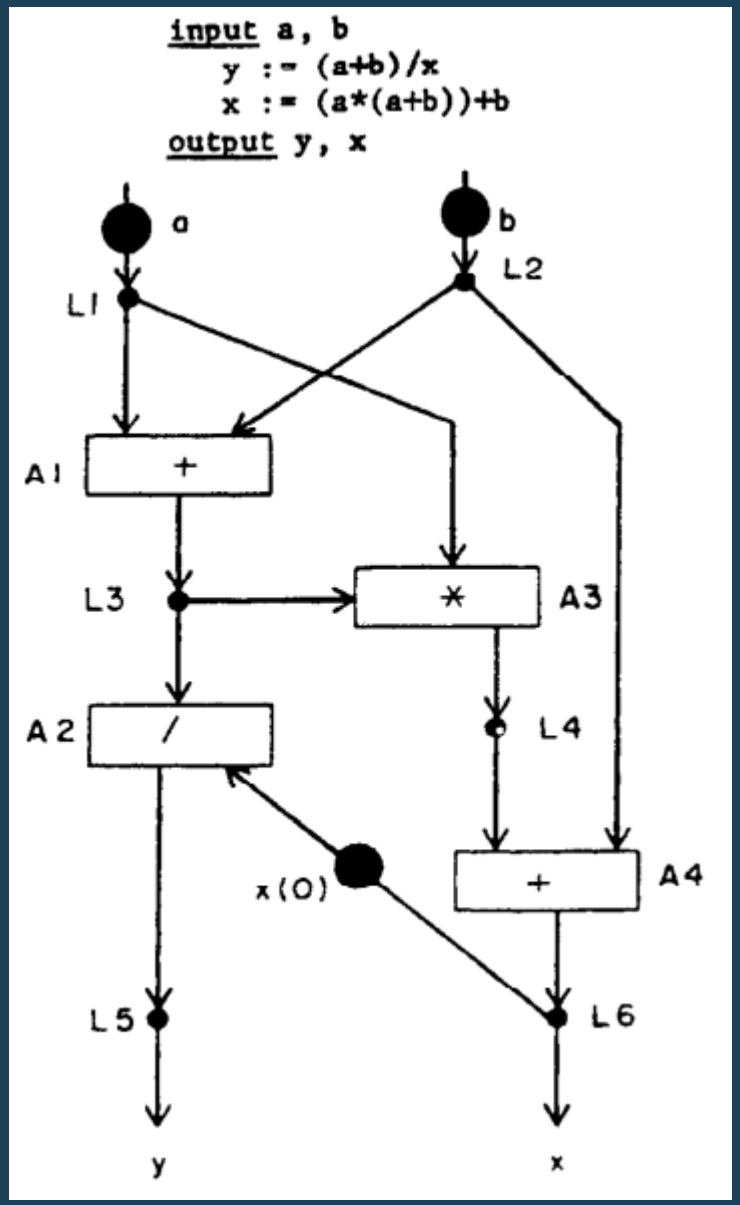
Dataflow Architectures

Alex Dathskovsky | alex.dathskovsky@speedata.io | https://www.linkedin.com/in/alexdathskovsky

SPEEDATA

Dataflow Architectures

- No Program Counter
- Execution occurs when data is ready
- Can be non-deterministic



J. Dennis, David Misunas
"A Preliminary Architecture for a Basic Data Flow Processor"
ISCA 1974

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

Dataflow Architectures

- Pros:
  - Natively parallel
  - Reduces data hazards

Dataflow Architectures

- Cons:
  - Functions, Loops, Recursion
  - Memory
  - Interrupts and Exceptions
  - Debugging
  - Programming

What Can We do?

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

MFC Architecture

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

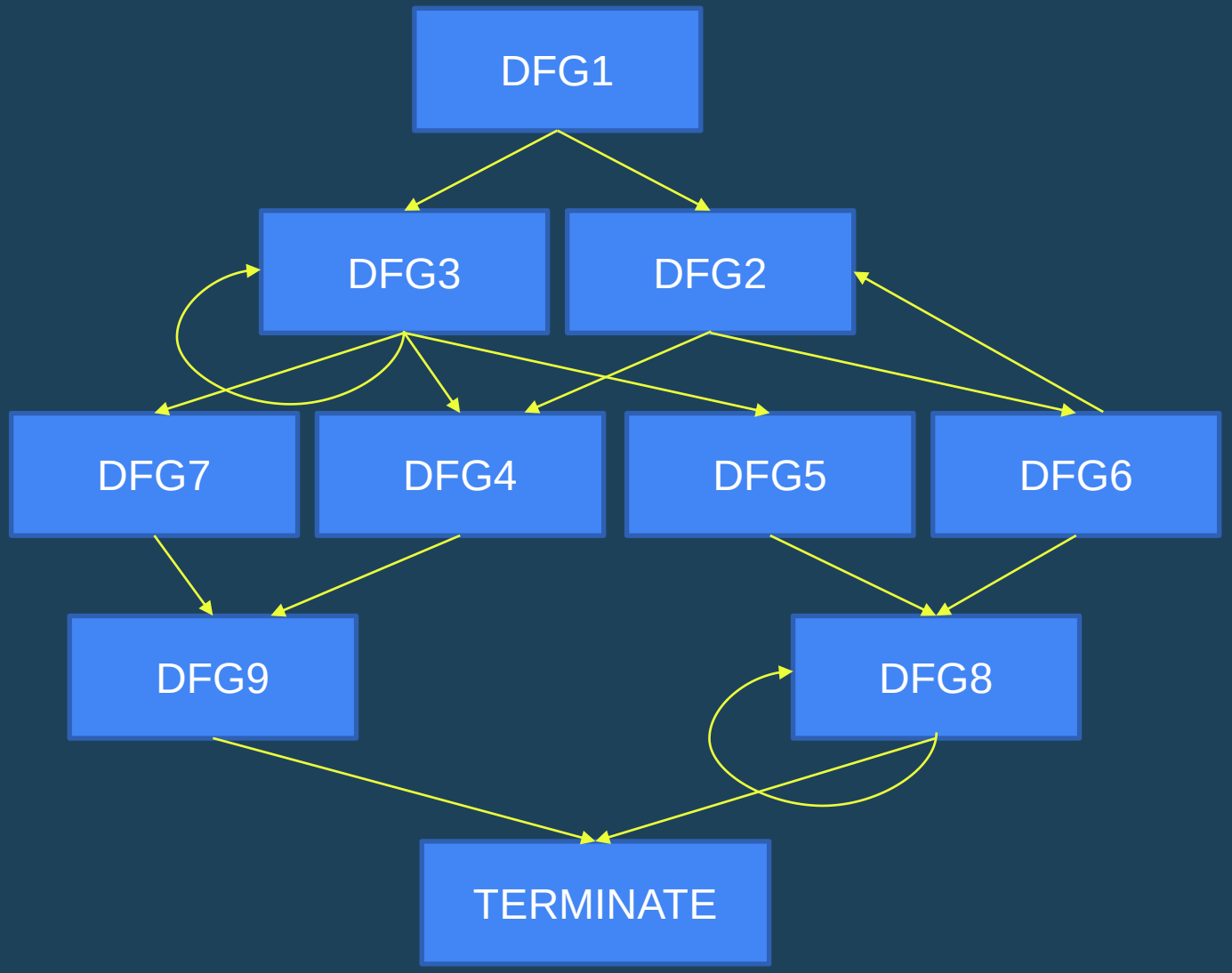
MFC

- Performs very complex operations in one instruction
- Handle large amounts of data
- Vector Instructions (threads)
- Support divergence between flows

MFC - Program

- Each Basic Block of the Program is a DFG (Handling Multiple Data)
- At The end of Each DFG there's a jump to a new DFG
- All flows start at DFG1
- Program Ends when all flows are at Terminator DFG

MFC - Program



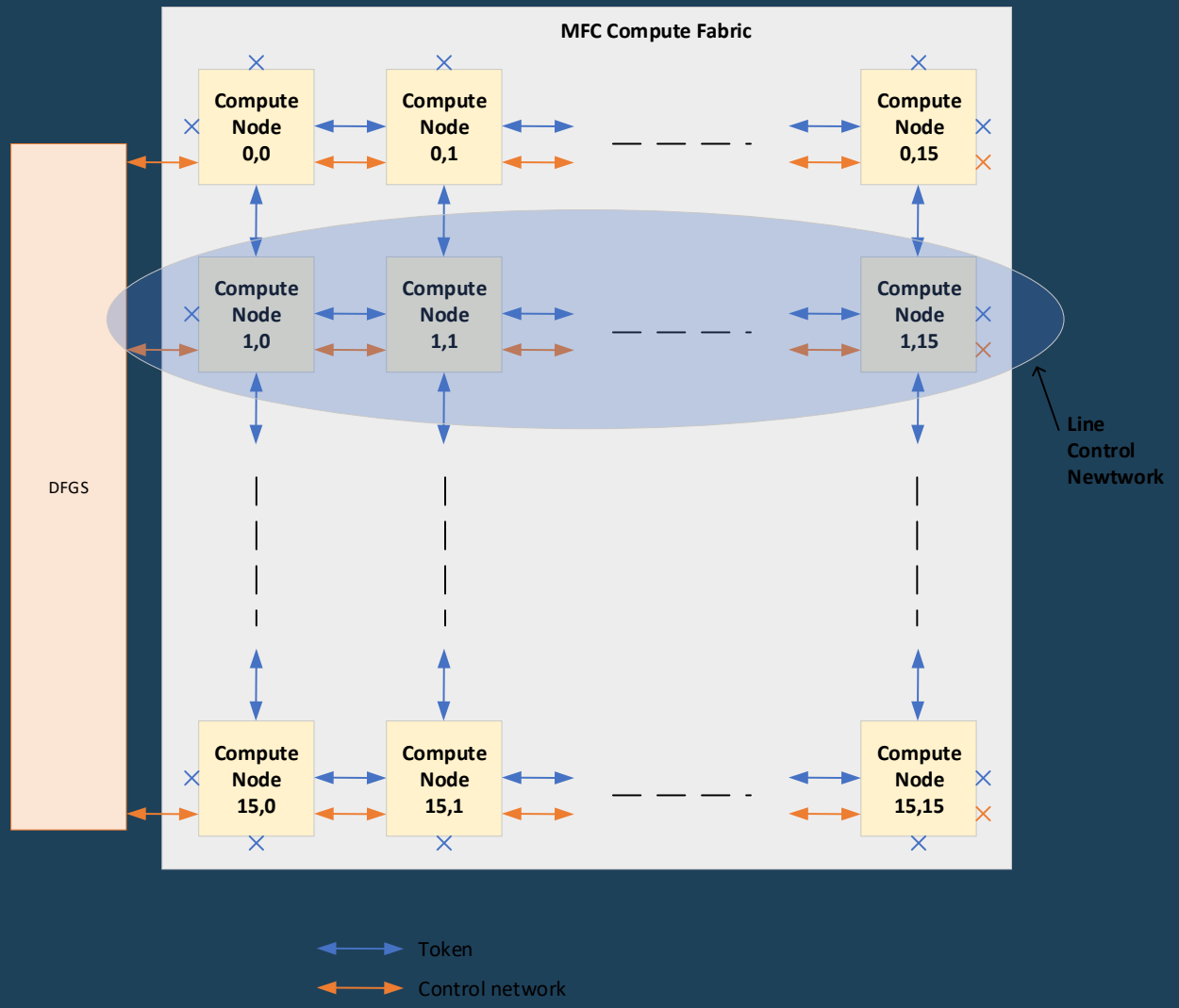
Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

MFC - CGRA

- Mesh of nodes
- Nodes connected with static switches
- Every Node has its own operation
- Tokens Correspond to flow
- Tokens pass Through execution Nodes and end in TERM
- TERM node determines next DFG for each flow
- 3 different state storges:
  - LVC
  - Memory
  - Streaming Buffer

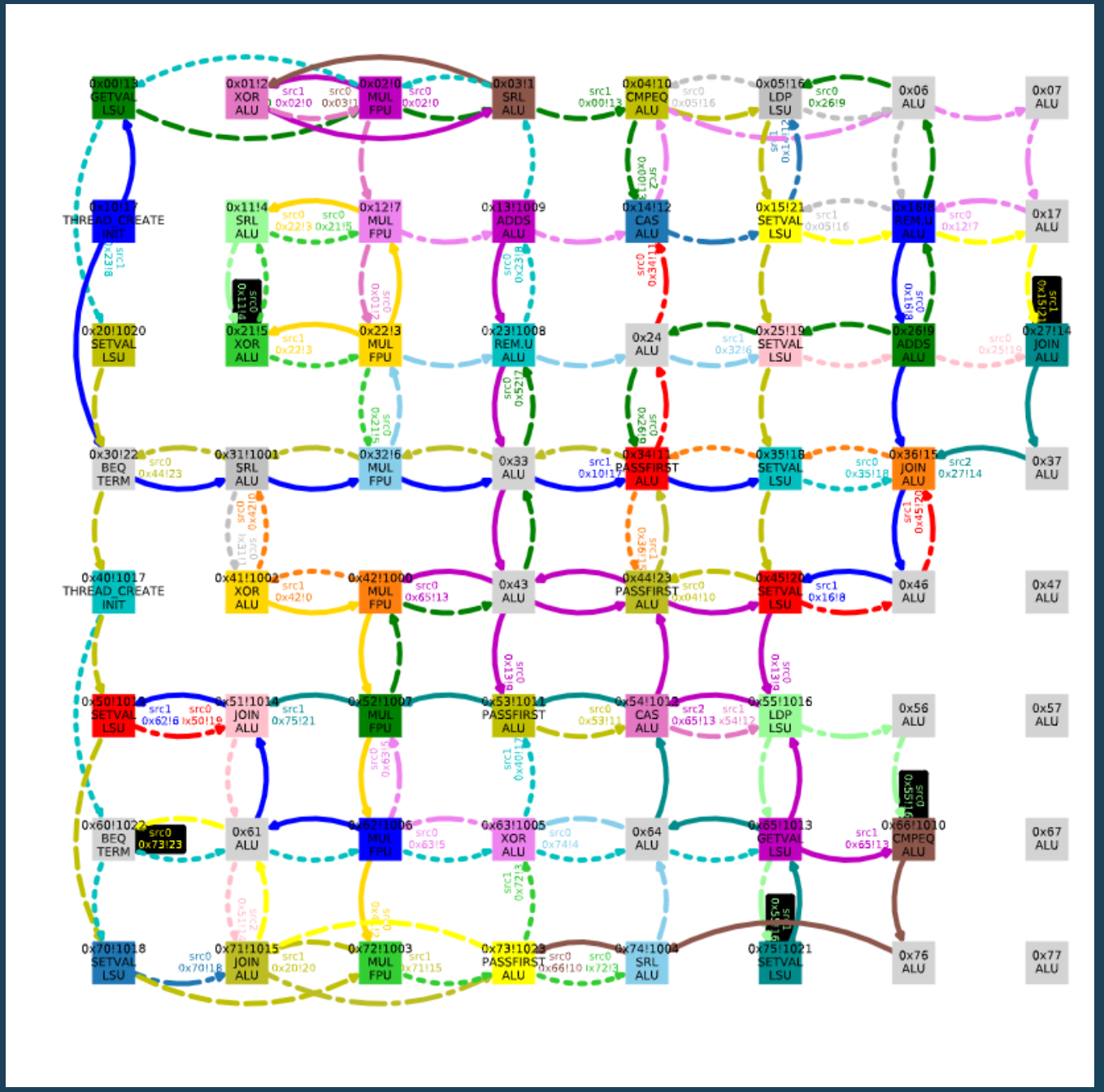
MFC – Compute Fabric (CGRA)



Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

MFC - CGRA



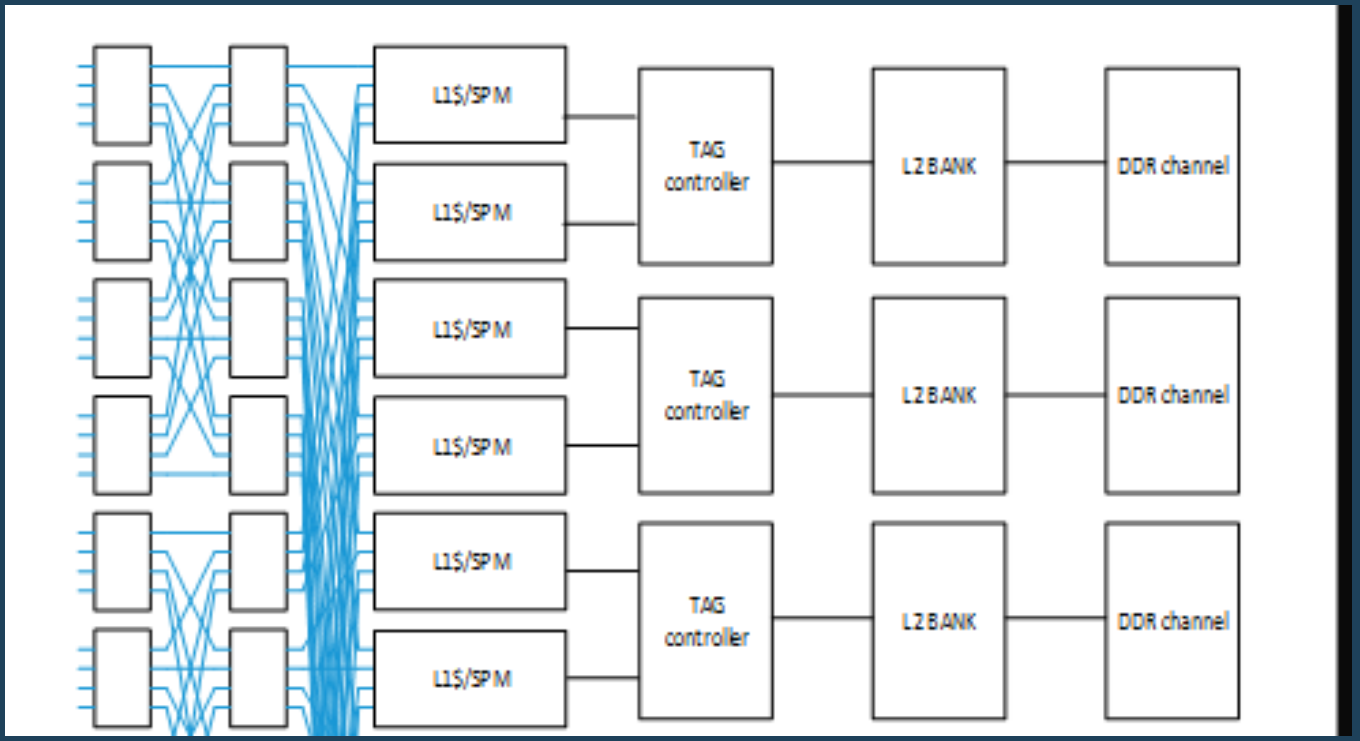
Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

Memory Sub-System

- L1 – HW managed cache backed by L2 and main DRAM
- LVC – Register (if we had registers) used to pass information between DFGS
- Streaming Buffer – SW managed and can be used to exchange information between components of the SoC
- System accessed by large number of LSUs
- Allows Large Number Of accesses

Memory Sub-System



Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

How to Code It?

DOGE- Dataflow Oriented Graph Execution

- A low-level interface to the APU
- Similar to CUDA
- Gives the user to ability to access low level instruction and device built-ins
- Clang frontend and LLVM backend

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

DOGE- Dataflow Oriented Graph Execution

Kernel:

```
KERNEL_DEFINE(ex1_single_dfg, long* d_out0) {
  int  tid = get_flow_id();
  d_out0[tid] = 14*tid + 11;
  return;
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

DOGE- Dataflow Oriented Graph Execution

Host Code:

```
long* d_out0 = apu::memory_mgr::DeviceMemMgr::GetInstance().Allocate(
                NUM_OF_ELEMS*sizeof(long));
long* h_out0 = new long[NUM_OF_ELEMS]; //output array
for (int i = 0; i < NUM_OF_ELEMS; i++) {
  h_out0[i] = EMPTY_KEY_64;
}
apu::memory_mgr::DeviceMemMgr::GetInstance().CopyToDevice(
                h_out0, d_out0, NUM_OF_ELEMS*sizeof(long));

KERNEL_LAUNCH(ex1_single_dfg,1,BLOCK_SIZE,d_out0);

// Copy the GPU memory back to the CPU here
apu::memory_mgr::DeviceMemMgr::GetInstance().CopyFromDevice(
                h_out0, d_out0, NUM_OF_ELEMS*sizeof(long));
fmt::print("h_out[{}] = {}\n", 1, h_out0[1]);
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

DOGE- Dataflow Oriented Graph Execution

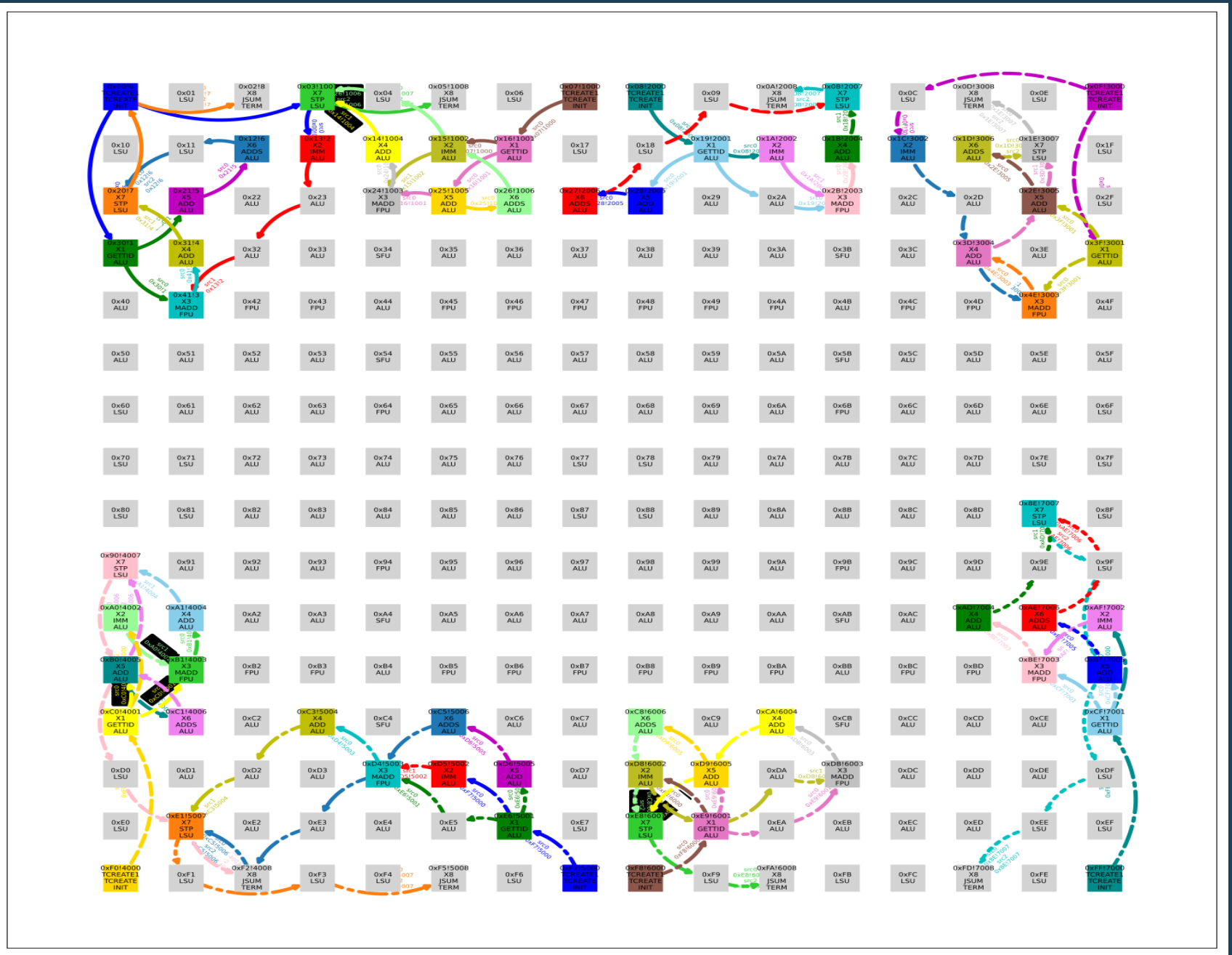
DFGs:

```
#DFG_ID,1,entry:
        TCREATE1 = TCREATE
        REG0 = GETFID.64D
        REG1 = IMM.U $14
        REG2 = MADD.64D.U REG0, REG1, $11
        REG5 = ADDS.64D.U REG0, $3, &11
        REG6 = STP.PANY.L.64M.U REG5, REG2, TCREATE1
        REG7 = THALT REG6
# End-Function
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

DOGE:



Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

DOGE- Dataflow Oriented Graph Execution

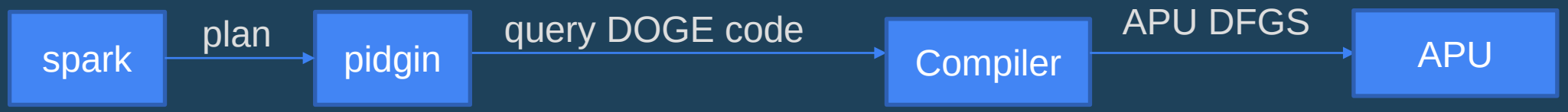
- Writing parallel code is hard
- Implementing SQL cases is even harder

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

PIDGIN

- Code generation for Query Execution



PIDGIN : SQL Simple Case

ID	Planet
1	Mars
42	Neptune
2	Mars
5	Venus
None	Mars
None	Venus
41	Neptune
99	None
19	None

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

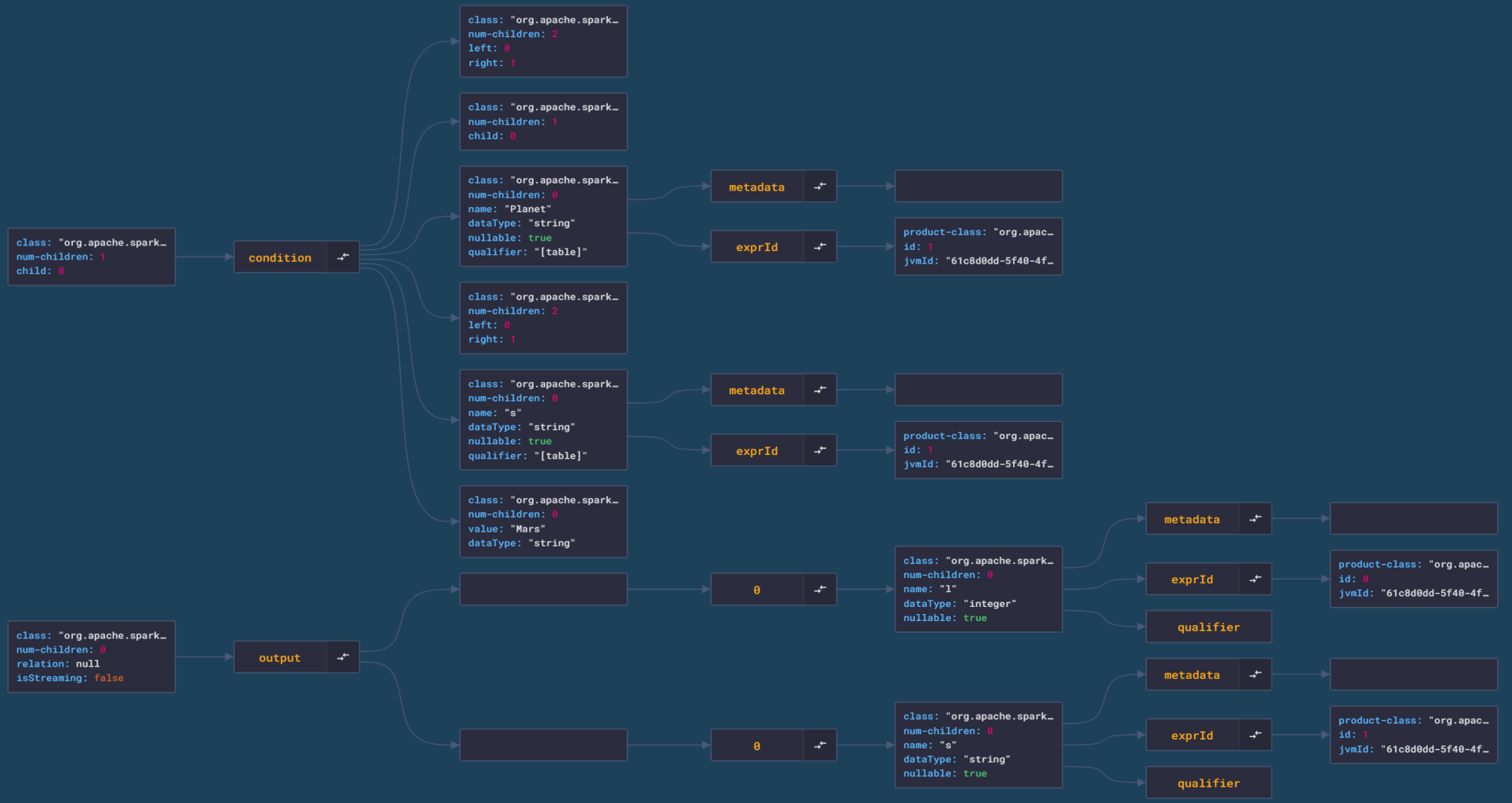
PIDGIN : SQL Simple Case

```
1 SELECT *
2 FROM TBL WHERE planet=='Mars'
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

PIDGIN : PLAN



simple_filter_plan

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

PIDGIN : Kernels

```
// SchemaType3 columns: i, Planet
using SchemaType3 = Schema<Int64Type, FLBAType<7>>;

using RowFormatType19 = RowFormat<SchemaType3>;

using InputBufferType18 = TileBufferOperator<RowFormatType19>;
// MFCLiteralType11: Mars
using MFCLiteralType11 = StringLiteral<'M', 'a', 'r', 's'>;

using RowFormatType2 = RowFormat<SchemaType3>;
// MFCColumnValueType17: planet
using MFCColumnValueType17 = ColumnValue<RowFormatType2, 1>;
// MFCEqualsType16: (planet) MFCEquals (Mars)
using MFCEqualsType16 = Equals<MFCColumnValueType17, MFCLiteralType11>;
```

PIDGIN : Kernels

```
// MFCColumnValueType15: Planet
using MFCColumnValueType15 = ColumnValue<RowFormatType2, 1>;
// MFCIsNotNullType14: MFCIsNotNull(planet)
using MFCIsNotNullType14 = IsNotNull<MFCColumnValueType15>;
// MFCLogicalAndType13: (MFCIsNotNull(planet))
// MFCLogicalAnd ((Planet) MFCEquals (Mars))
using MFCLogicalAndType13 = LogicalAnd<MFCIsNotNullType14, MFCEqualsType16>;

using FilterType5 = Filter<RowFormatType2, MFCLogicalAndType13>;
using ResultTableBufferType1 = TileBufferOperator<RowFormatType2>;
```

PIDGIN : Kernels

```
using OutputFormat = RowFormatType2;
static const vector<std::string> OutputFormatNames{"i","planet"};

template<class NextProcessor>
using FilterRowProcessorType4 = FilterRowProcessor<FilterType5, NextProcessor>;

template<class NextProcessor>
using RowScanType12 = RowScan<RowFormatType19, NextProcessor>;

using InputFormats = std::tuple<RowFormatType19>;
```

PIDGIN : Kernels

```
// RowScan->FilterRowProcessor->RowSink
using CompositionType20 = ComposeProcessors<RowScanType12,
                                            FilterRowProcessorType4,
                                            RowSinkType0>;

// RowScan->FilterRowProcessor->RowSink
KERNEL_DEFINE(kernel0, size_t num_rows, uint8_t* arg0,
uint8_t* arg1, uint8_t* arg2) {
  CompositionType20 kernel0_plan(arg0, arg1, arg2);
  auto thread_id = get_flow_id();
  auto stride_size = get_flow_block();
  for (size_t row_idx = thread_id; row_idx < num_rows; row_idx += stride_size){
    kernel0_plan.Process(row_idx);
  }
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

PIDGIN

- That was an example of a very simple kernel
- Pidgin can scale out to any Query

PIDGIN : Complex Case

```
SELECT c.contactIDX
FROM [contacts] c
INNER JOIN [orders]  o
ON c.contactIDX = o.contactIDX
WHERE  o.orderdate >=  date_add(current_date,-365)
AND c.country = 'Imaginary Country'
GROUP BY c.contactIDX;
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

PIDGIN : Complex Case kernels

```
// RowScan->FilterRowProcessor->ProjectionRowProcessor->HashJoinSinkProcessor
KERNEL_DEFINE(kernel0, size_t num_rows, uint8_t* arg0, uint8_t* arg1, uint8_t* arg2, uint8_t* arg3) {
  CompositionType42 kernel0_plan(arg0, arg1, arg2, arg3);
  auto thread_id = (blockDim.x * threadIdx.y + threadIdx.x);
  auto stride_size = blockDim.x * blockDim.y;
  for (size_t row_idx = thread_id; row_idx < num_rows; row_idx += stride_size) {
    kernel0_plan.Process(row_idx);
  }
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

PIDGIN : Complex Case kernels

```
// RowScan->FilterRowProcessor->ProjectionRowProcessor->
// HashJoinPipeProcessor->ProjectionRowProcessor->GroupByRowProcessor
KERNEL_DEFINE(kernel1, size_t num_rows, uint8_t* arg0, uint8_t* arg1, uint8_t* arg2,
uint8_t* arg3, uint8_t* arg4, uint8_t* arg5) {
  CompositionType99 kernel1_plan(arg0, arg1, arg2, arg3, arg4, arg5);
  auto thread_id = (blockDim.x * threadIdx.y + threadIdx.x);
  auto stride_size = blockDim.x * blockDim.y;
  for (size_t row_idx = thread_id; row_idx < num_rows; row_idx += stride_size) {
    kernel1_plan.Process(row_idx);
  }
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

PIDGIN : Complex Case kernels

```
// GroupByRowScan->ProjectionRowProcessor->RowSink
KERNEL_DEFINE(kernel2, size_t num_rows, uint8_t* arg0, uint8_t* arg1, uint8_t* arg2) {
  CompositionType60 kernel2_plan(arg0, arg1, arg2);
  auto thread_id = (blockDim.x * threadIdx.y + threadIdx.x);
  auto stride_size = blockDim.x * blockDim.y;
  for (size_t row_idx = thread_id; row_idx < num_rows; row_idx += stride_size) {
    kernel2_plan.Process(row_idx);
  }
}
```

Alex Dathskovsky | alex.dathskovsky@speedata.io | www.linkedin.com/in/alexdathskovsky

SPEEDATA

SPEEDATA

Questions ?

SPEEDATA

THANK YOU!