Automatic Parallelisation II

Björn Franke
Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

1st Workshop on Mapping of Applications to MPSoCs
St. Goar, Germany, June 2008

Outline

- Real Codes and Program Recovery
- Rank Modifying Code and Data Transformations
- Multi-DSP and Multiple Address Spaces
- Coarse Grain Parallelism and Task Graph Extraction
- Conclusion

Problems with Real Codes

- “Hand-optimised” for specific platform
    - E.g. pointer based array traversals for efficient address calculation
- Unusual idioms to implement frequently used operations
    - E.g. modulo operations in array index expressions for circular buffers
- Defeat auto-parallelisation!
    - “Plain C” is best for parallelisation
    - Need to recover canonical program representation

Program Recovery

Affine array index expressions are critical!

Pointer Conversion

```
int A[N],B[N],C[N];
int *ptr_a = &A[0];
int *ptr_b = &B[0];
int *ptr_c = &C[N-1];

for (i = 0; i < N; i++)
{
   *ptr_a++ = *ptr_b++ * *ptr_c--;
}
```

Program Recovery

Affine array index expressions are critical!

Pointer Conversion

```
int A[N],B[N],C[N];




for (i = 0; i < N; i++)
{
   A[i] = B[i] * C[N-i-1];
}
```

Affine Index Expressions

Program Recovery

Affine array index expressions are critical!

<u>Modulo Elimination</u>

```
int A[N], B[M];


for (i = 0; i < N; i++)
{
   X = A[i] * B[i%M];
}
```

Program Recovery

Affine array index expressions are critical!

<u>Modulo Elimination</u>

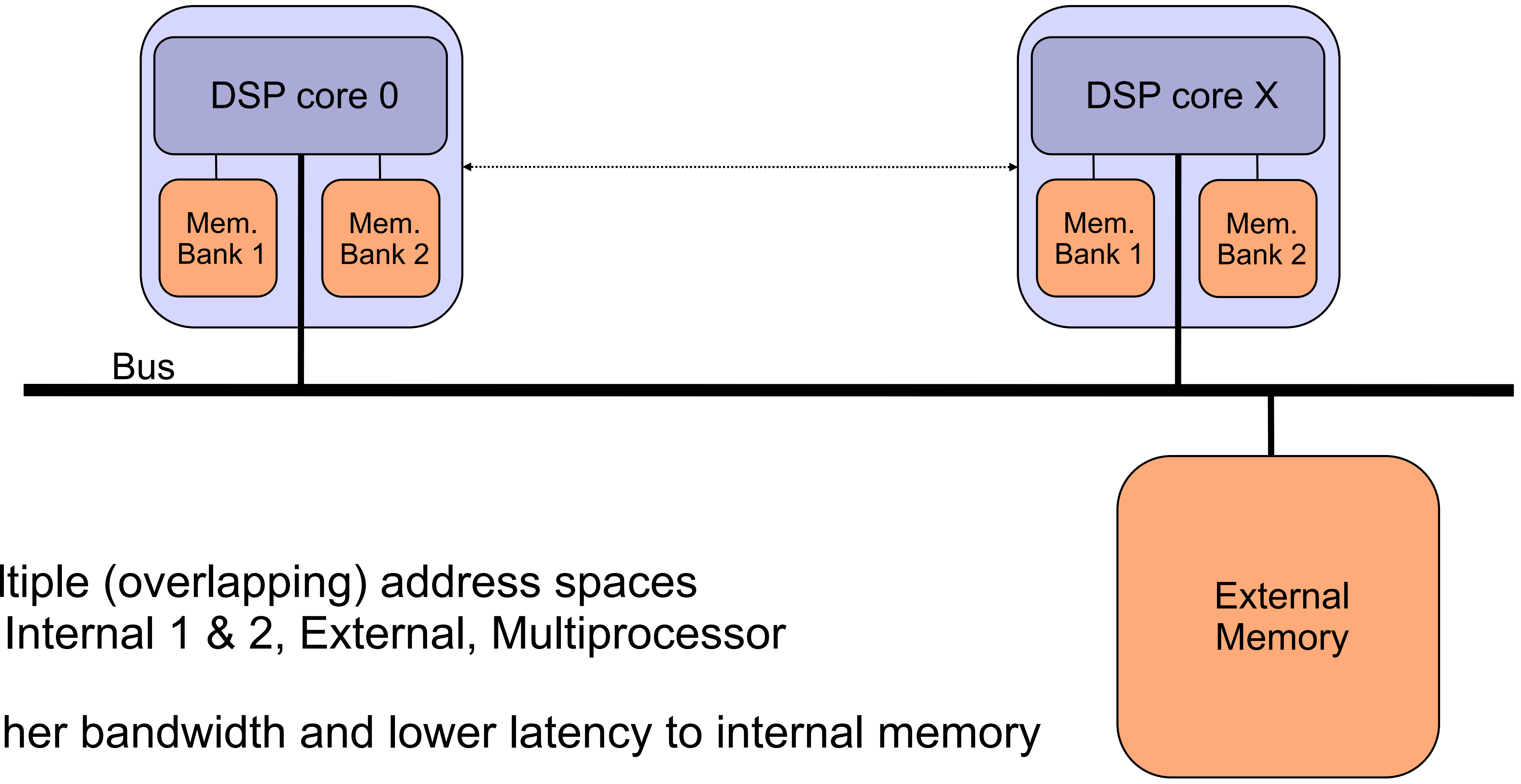
```
int A[N], B[M];


for (i = 0; i < N/M; i++)
{
   for (j = 0; j < M; j++)
   {
      X = A[i*M+j] * B[j];
   }
}
```

Affine Index Expressions

Rank Modifying Code and Data Transformations

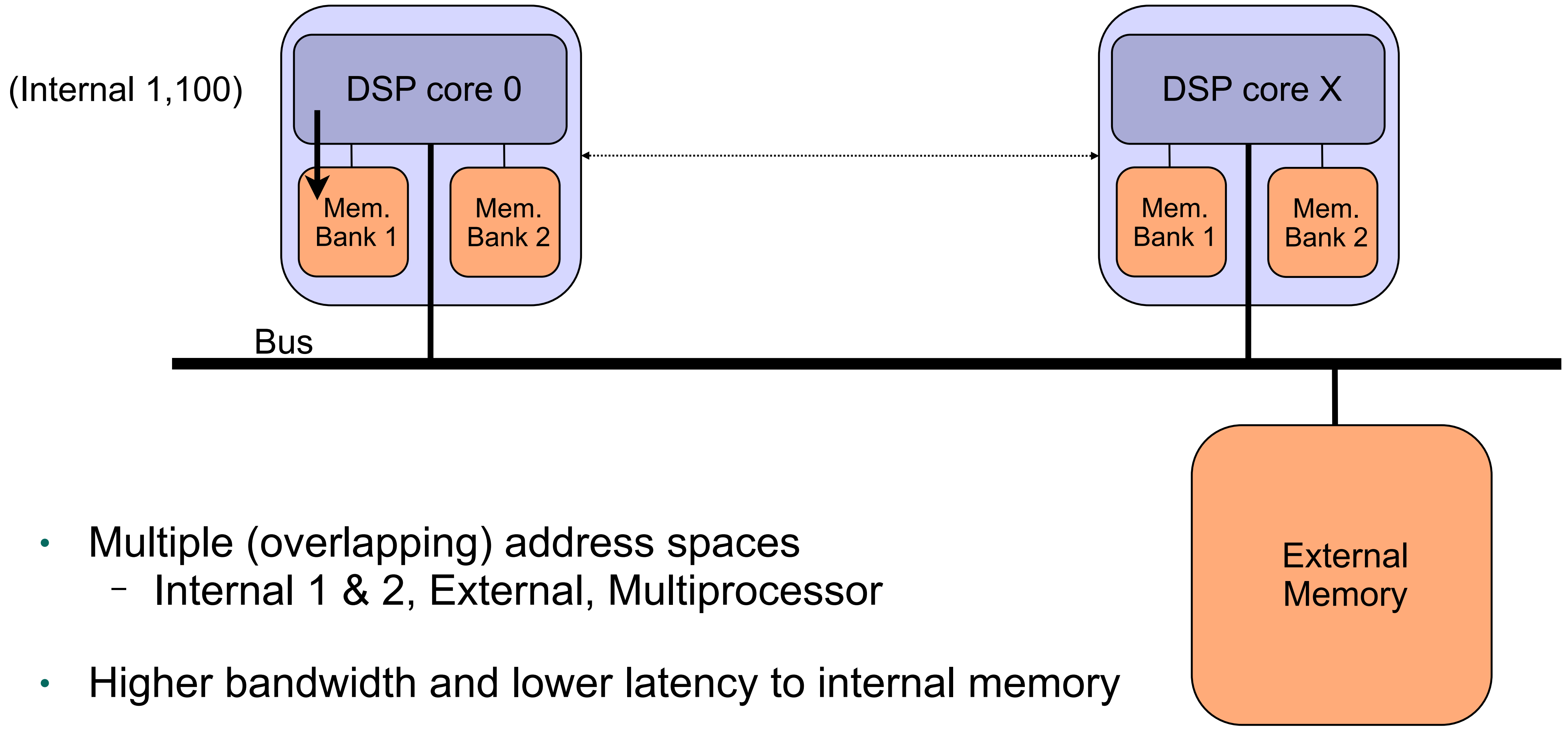
- Unified Linear Algebra Framework for Code and Data Transformations
- Extensions to Unimodular Transformations
    - Elements of transformation matrix are functions with `div` & `mod`
- Representation of additional transformations
    - Array dimensionality transformations
    - Strip-mining and linearisation of loops

Multi-DSP



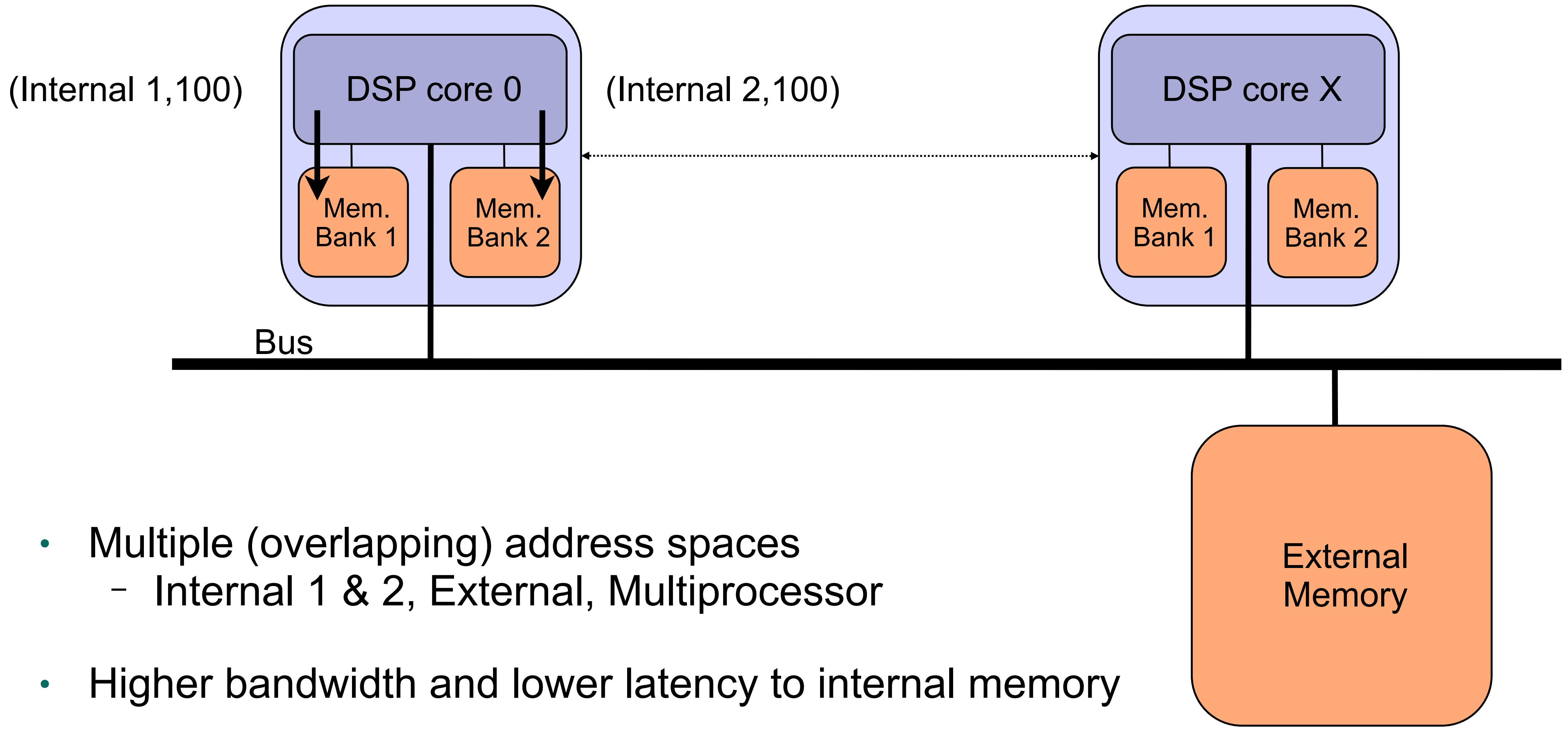
- Multiple (overlapping) address spaces
  - Internal 1 & 2, External, Multiprocessor
- Higher bandwidth and lower latency to internal memory
- Globally accessible memory
  - Remote memory accesses, but need to know ID of remote processor and local offset
  - DMA based bulk data transfers

Multi-DSP



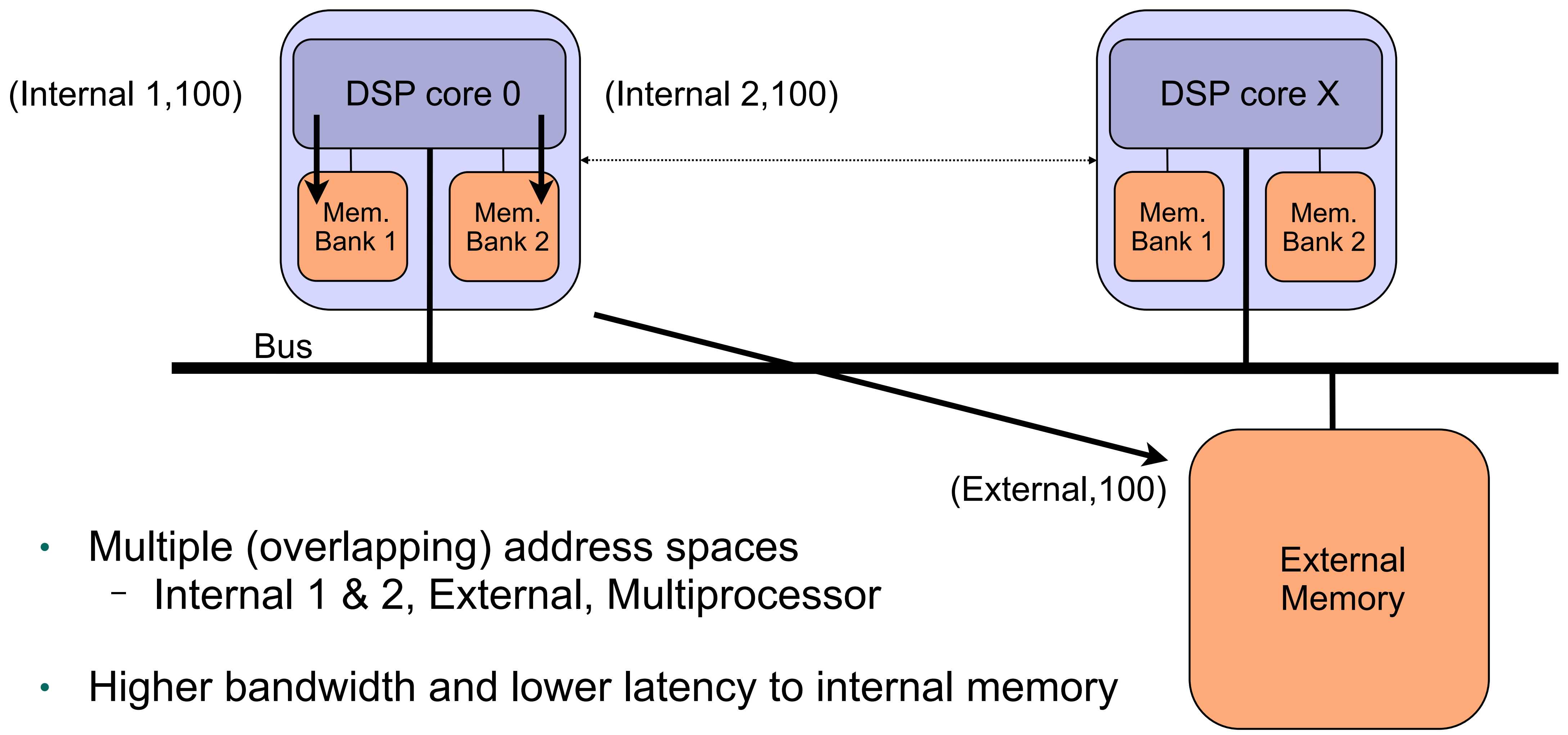
- Multiple (overlapping) address spaces
  - Internal 1 & 2, External, Multiprocessor
- Higher bandwidth and lower latency to internal memory
- Globally accessible memory
  - Remote memory accesses, but need to know ID of remote processor and local offset
  - DMA based bulk data transfers

Multi-DSP



- Multiple (overlapping) address spaces
  - Internal 1 & 2, External, Multiprocessor
- Higher bandwidth and lower latency to internal memory
- Globally accessible memory
  - Remote memory accesses, but need to know ID of remote processor and local offset
  - DMA based bulk data transfers

Multi-DSP

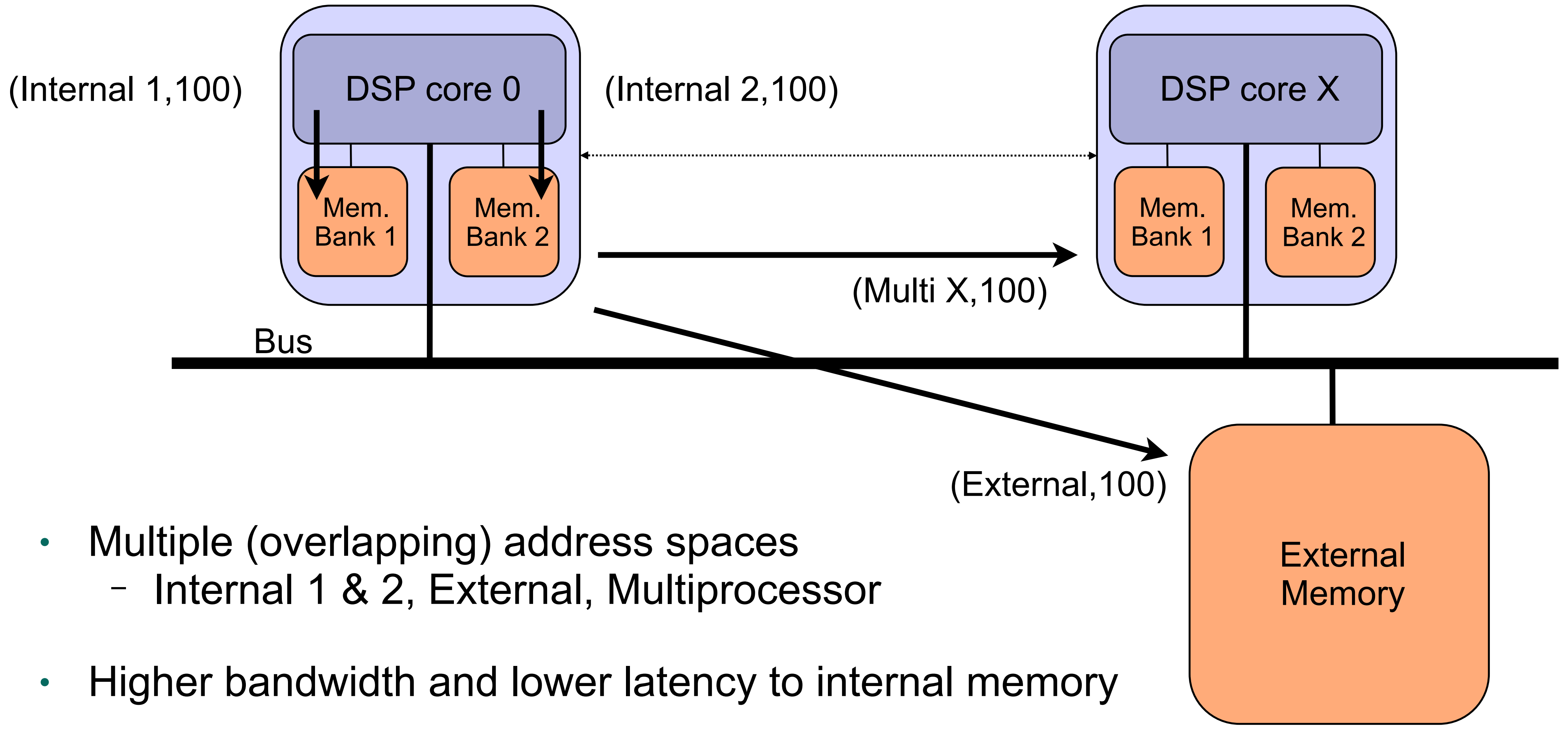


- Multiple (overlapping) address spaces
  - Internal 1 & 2, External, Multiprocessor

- Higher bandwidth and lower latency to internal memory

- Globally accessible memory
  - Remote memory accesses, but need to know ID of remote processor and local offset
  - DMA based bulk data transfers

Multi-DSP



- Multiple (overlapping) address spaces
  - Internal 1 & 2, External, Multiprocessor
- Higher bandwidth and lower latency to internal memory
- Globally accessible memory
  - Remote memory accesses, but need to know ID of remote processor and local offset
  - DMA based bulk data transfers

Transformations for Parallelisation

- Program Recovery
- Data partitioning, e.g. to minimise communication, owner computes
  - Data delinearisation and loop strip-mining (1 outer iteration = 1 processor)
- Mapping
  - Split data and distribute over processors, drop auxiliary outer loop
  - Introduce address desciptors for multiple memory spaces and replace array accesses with lookups (costly!)

Transformations for Parallelisation (continued)

- Local memory access optimisation
  - Drop lookups for provably local accesses
- DMA bulk data transfers
  - Hoist remote accesses out of compute loops, introduce local buffers and auxiliary load loops, convert load loops into DMA transfers
- Single transformation framework, avoid recomputation of critical information
- Very efficient for compute-intensive DSP kernels

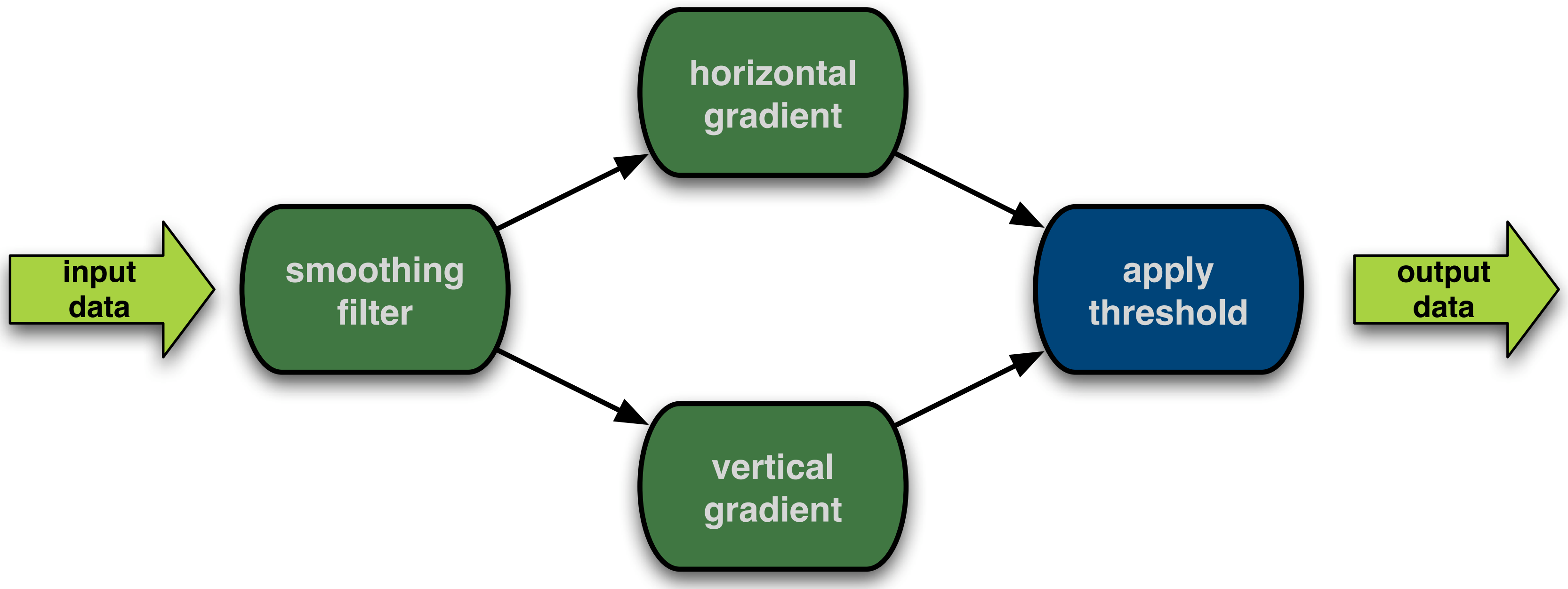
Coarse Grain Parallelism & Task Graph Extraction

- Loop parallelisation minimises latency
    - Good for scientific applications, but **throughput** often more important for embedded systems, e.g. multimedia applications
    - Embedded applications have often additional **constraints** on latency and efficiency
    - Recent survey shows **only 12.5%** of execution time of EEMBC and MediaBench benchmarks is spent in statically detectable DOALL loops
- Exploit streaming nature of embedded applications and (semi-)static algorithm behaviour with distinct phases!

Coarse Grain Parallelism & Task Graph Extraction

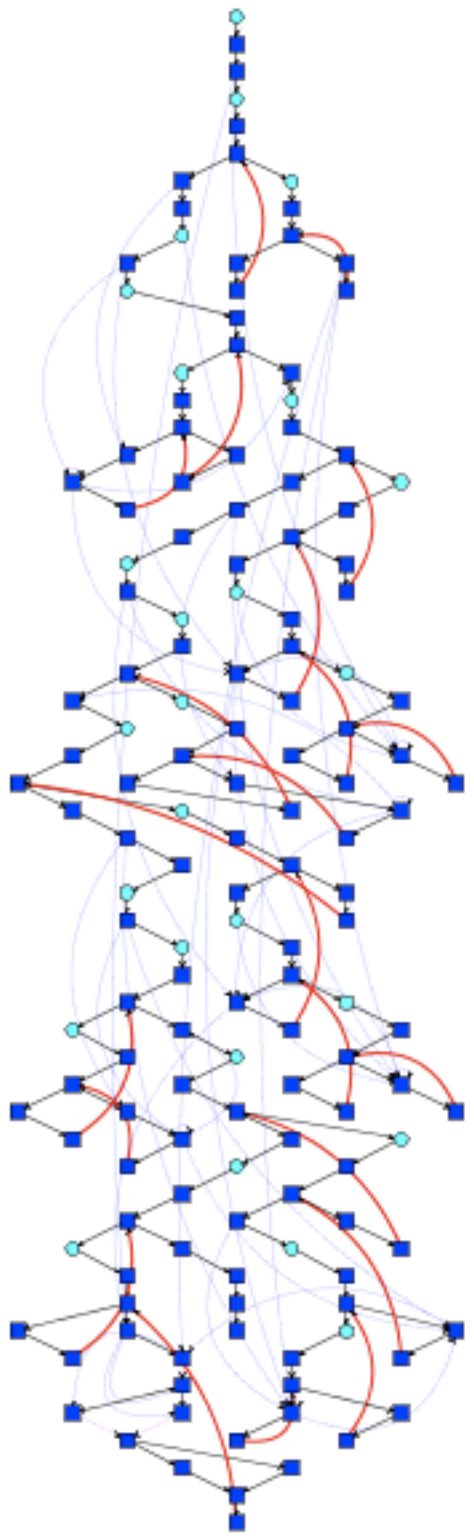
- Debugging
    - Run program in debugger, observe what happens.
    - NEVER seen any proof of correctness! Caveat: Safety-critical apps.
- Automatic Parallelisation
    - NEVER run program, analyse conservatively what might happen.
    - ALWAYS proof data/control independence!
- Right approach?

Coarse Grain Parallelism & Task Graph Extraction

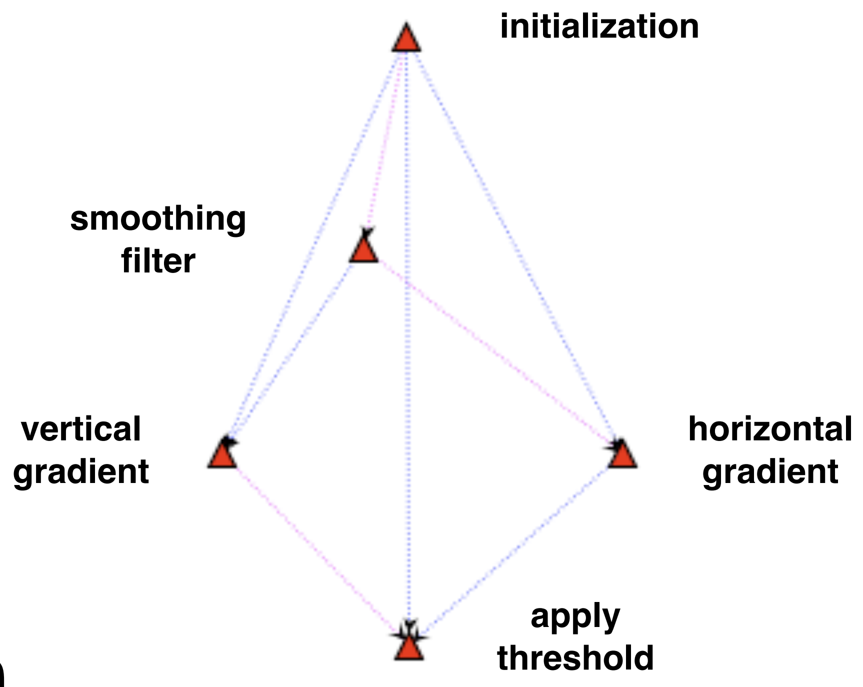


UTDSP Edge Detector

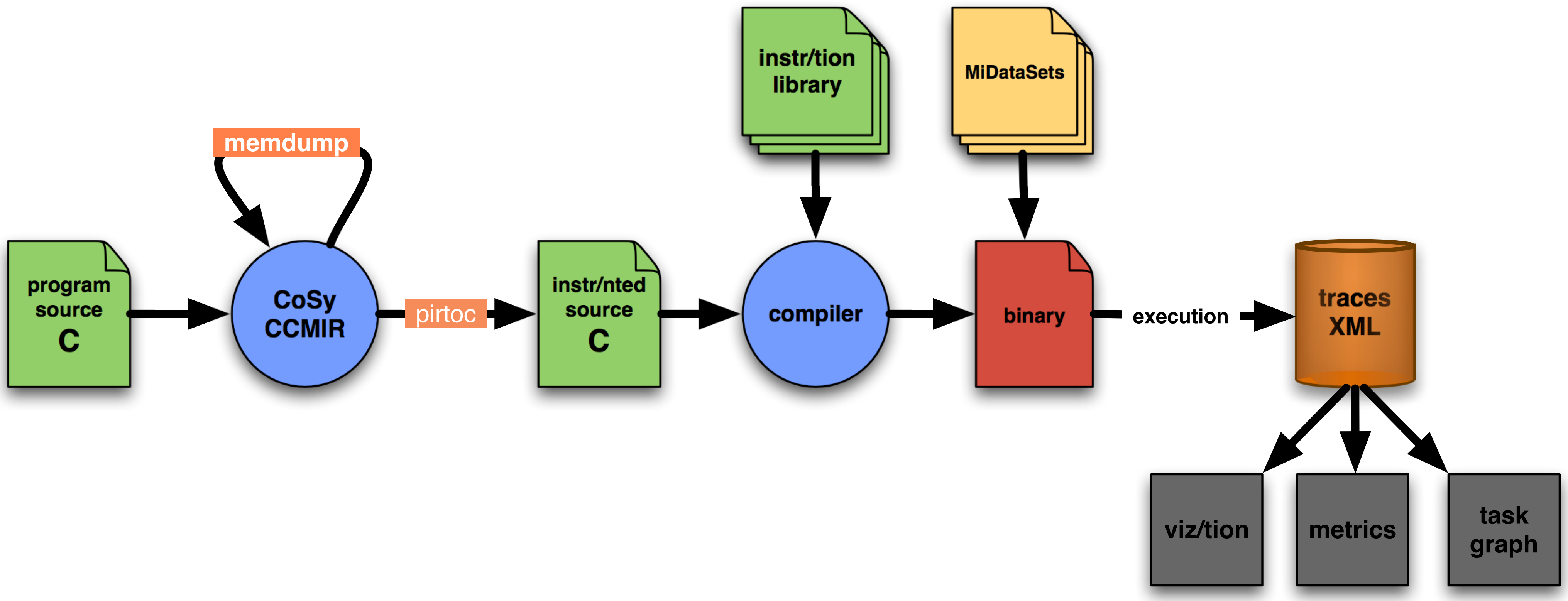
Coarse Grain Parallelism & Task Graph Extraction



The MAPS Approach

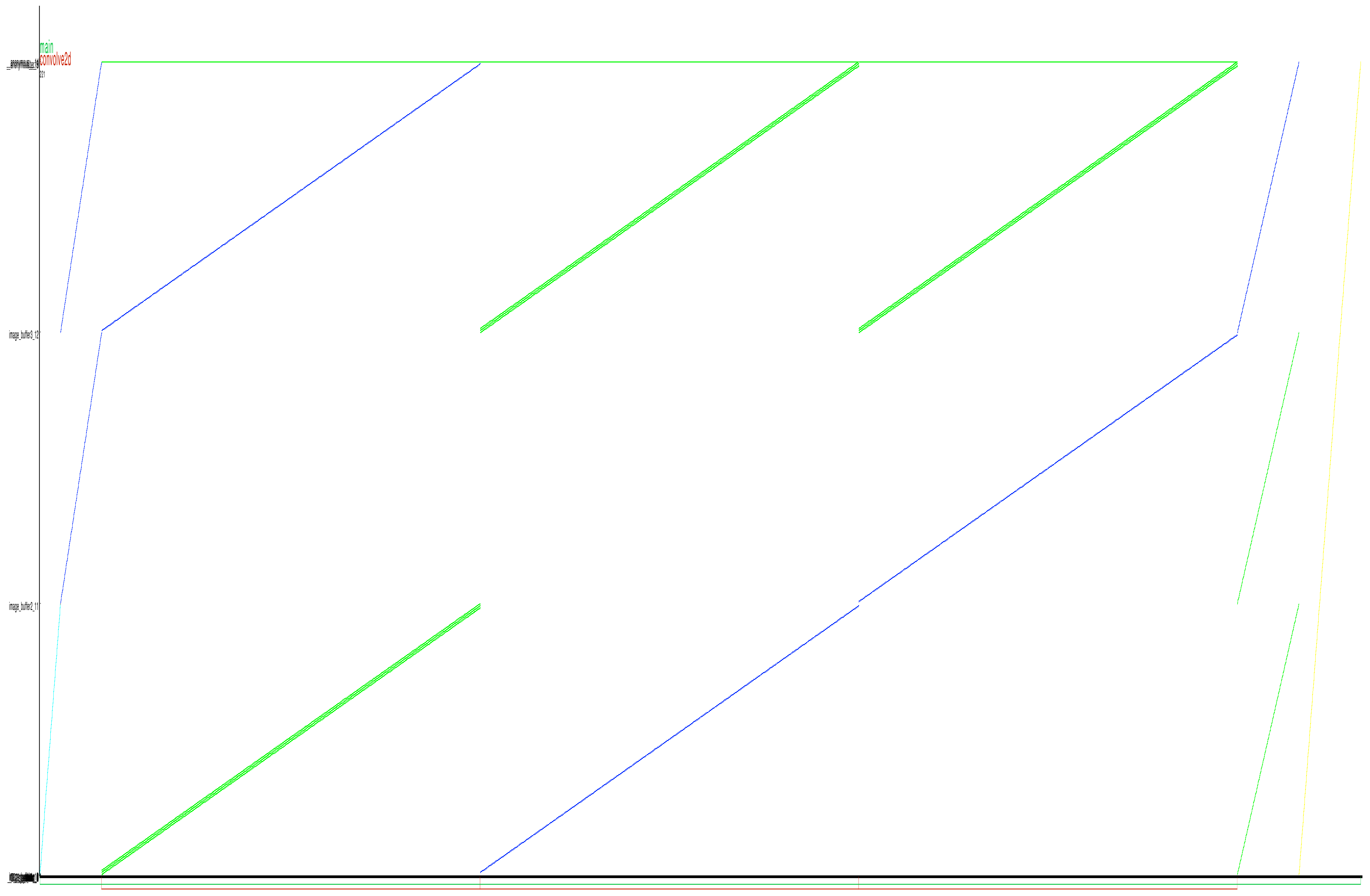


Coarse Grain Parallelism & Task Graph Extraction

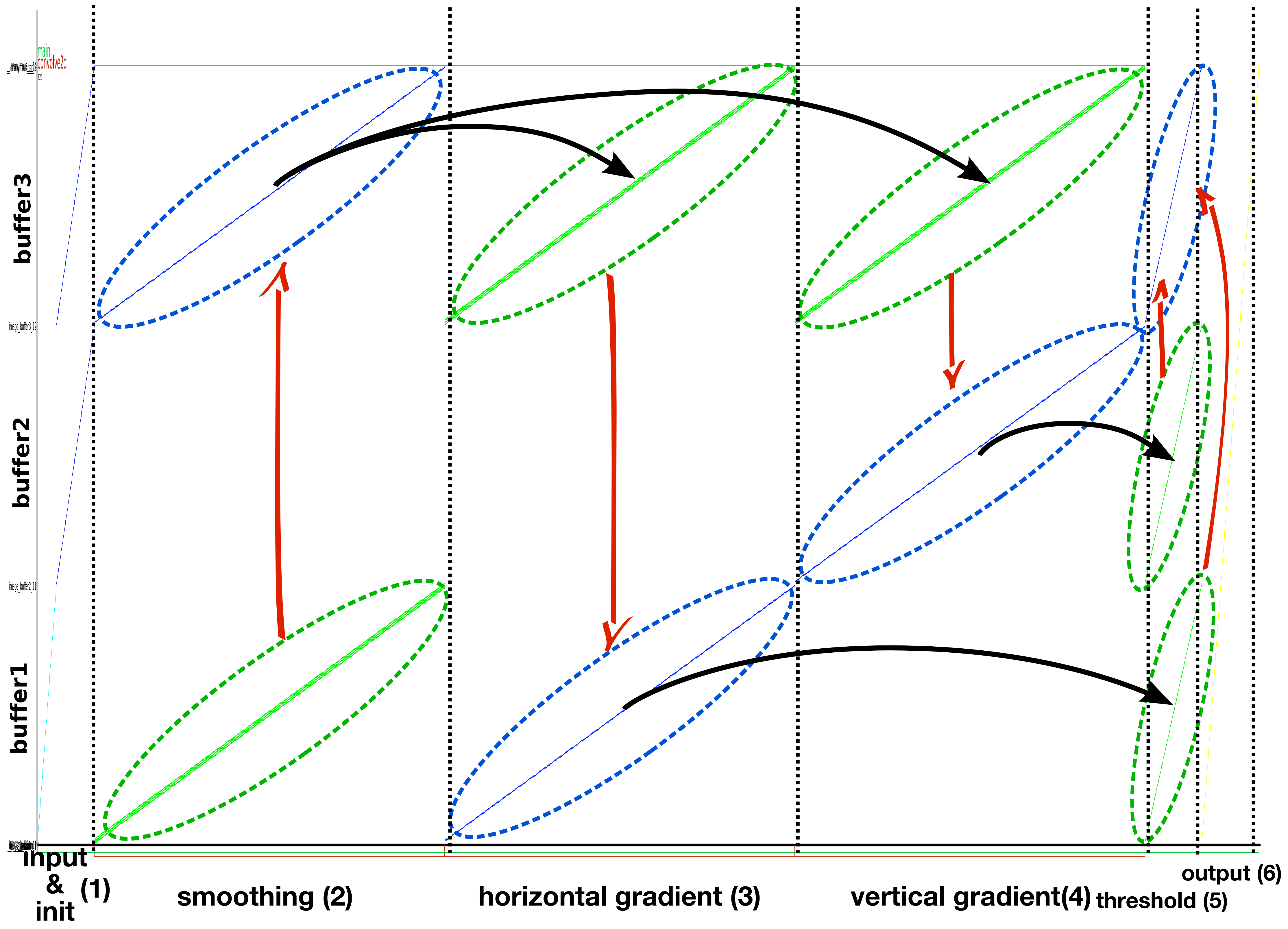


Profiling Infrastructure

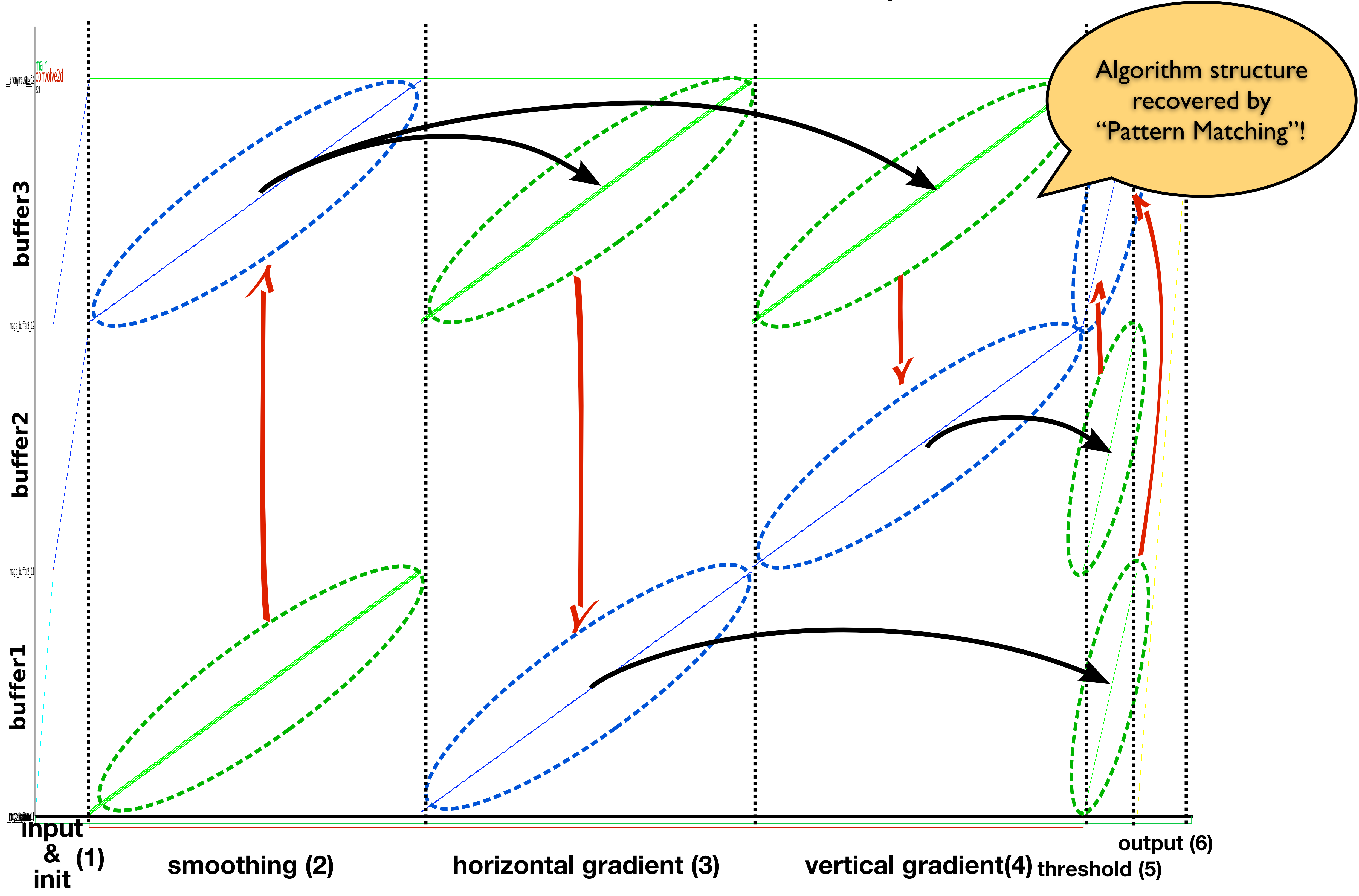
Coarse Grain Parallelism & Task Graph Extraction



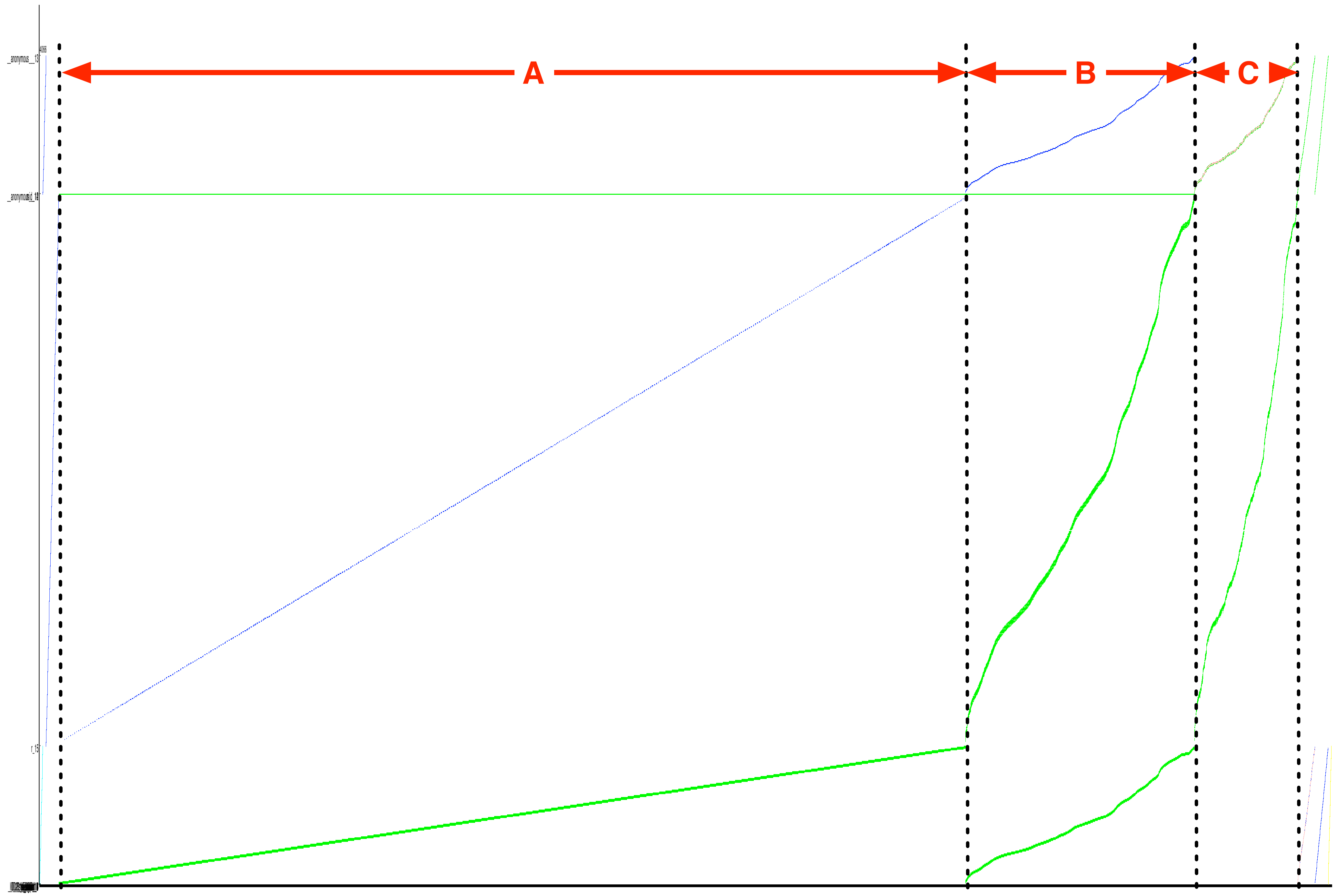
Coarse Grain Parallelism & Task Graph Extraction



Coarse Grain Parallelism & Task Graph Extraction

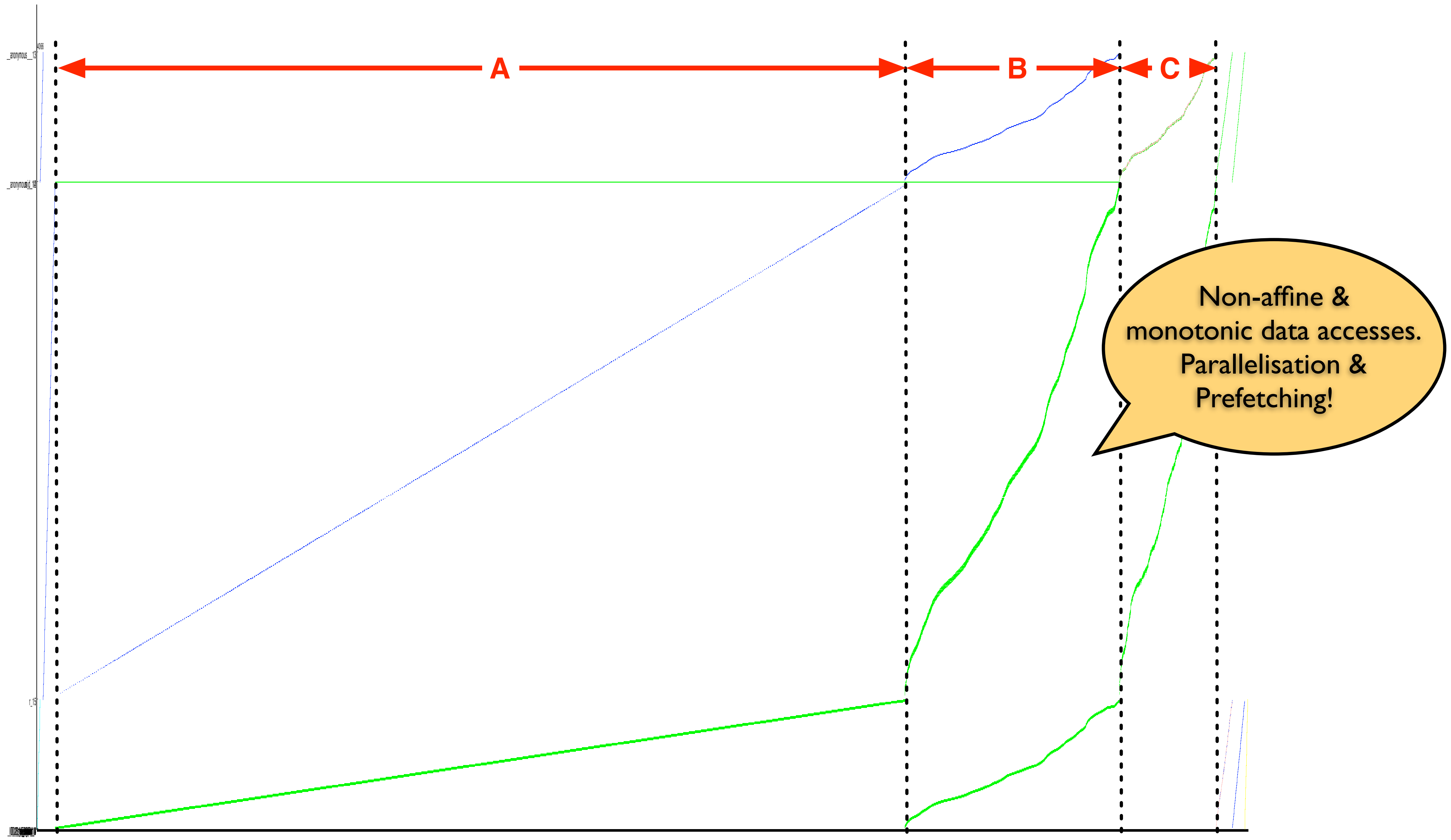


Coarse Grain Parallelism & Task Graph Extraction



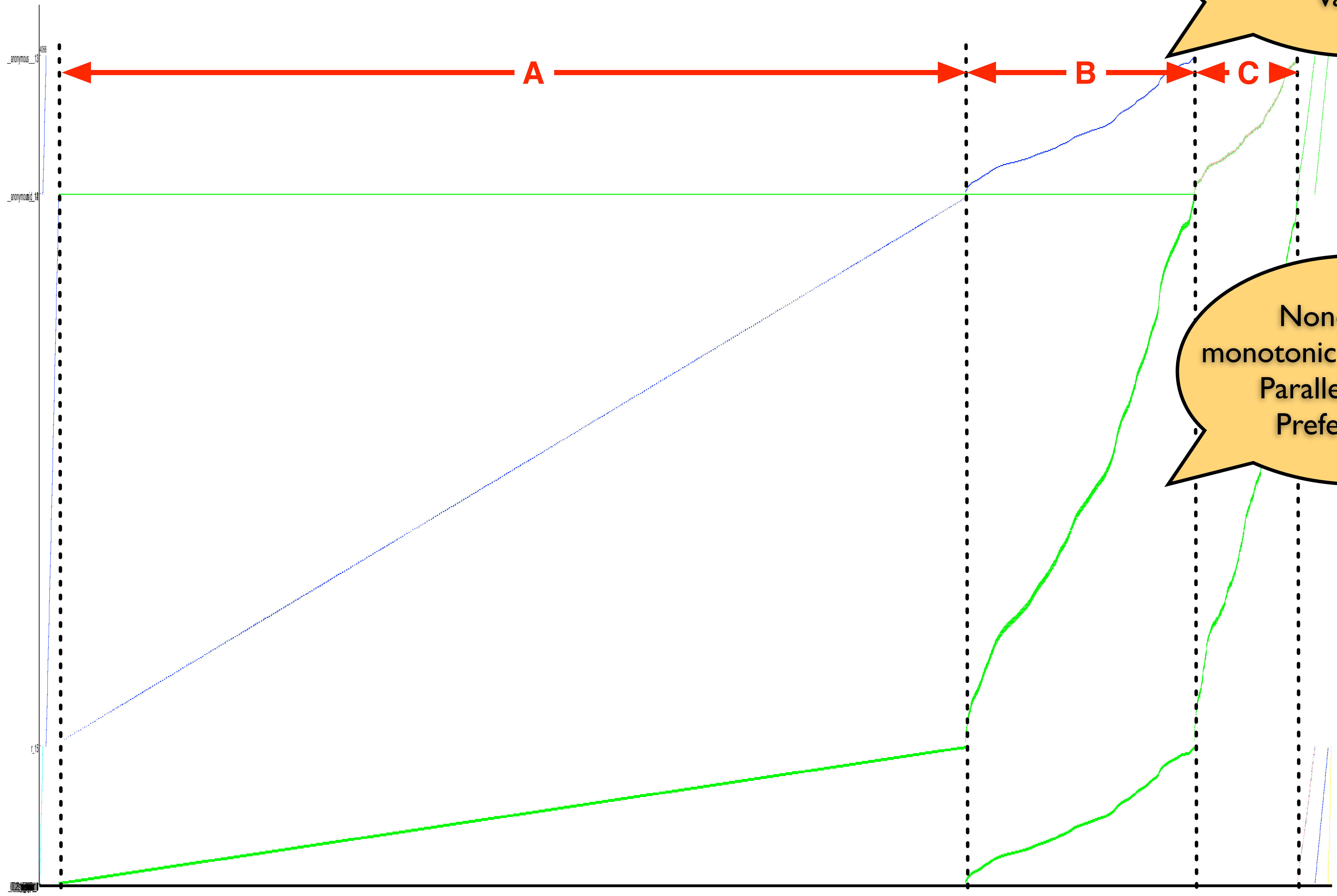
MiBench Susan

Coarse Grain Parallelism & Task Graph Extraction



MiBench Susan

Coarse Grain Parallelism & Task Graph Extr



Static sequence of phases A,B,C for all data inputs. Relative durations vary!

Non-affine & monotonic data accesses. Parallelisation & Prefetching!

MiBench Susan

Coarse Grain Parallelism & Task Graph Extraction

- Exploit semi-static behaviour for different data sets
    - No random behaviour on macroscopic scale, but patterns e.g. A(BC|D)*F
- Application “Scenarios”
    - Different modes of execution, e.g. progressive vs interlaced encoding
    - Trace different phases back to earliest decision point, usually header information or parameters
- Information about behaviour far into the future available, but we don’t use it!

Conclusion

- Beyond loop level parallelisation and static analysis!!!
    - Detailed qualitative and quantitative data
    - Combine static and profiling based parallelisation
    - Interactive, “Pattern matching”, iterative, statistical (evidence for hypothesis?) approaches. How to deal with “uncertainty”?
- Combine task graph extraction with loop parallelisation
    - Balancing of tasks, improve certain critical kernels