Basics of modern computers

Dag Sverre Seljebotn

October 25, 2012

Donald Knuth:

Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. *Yet we should not pass up our opportunities in that critical 3%.*

The law of leaky abstractions

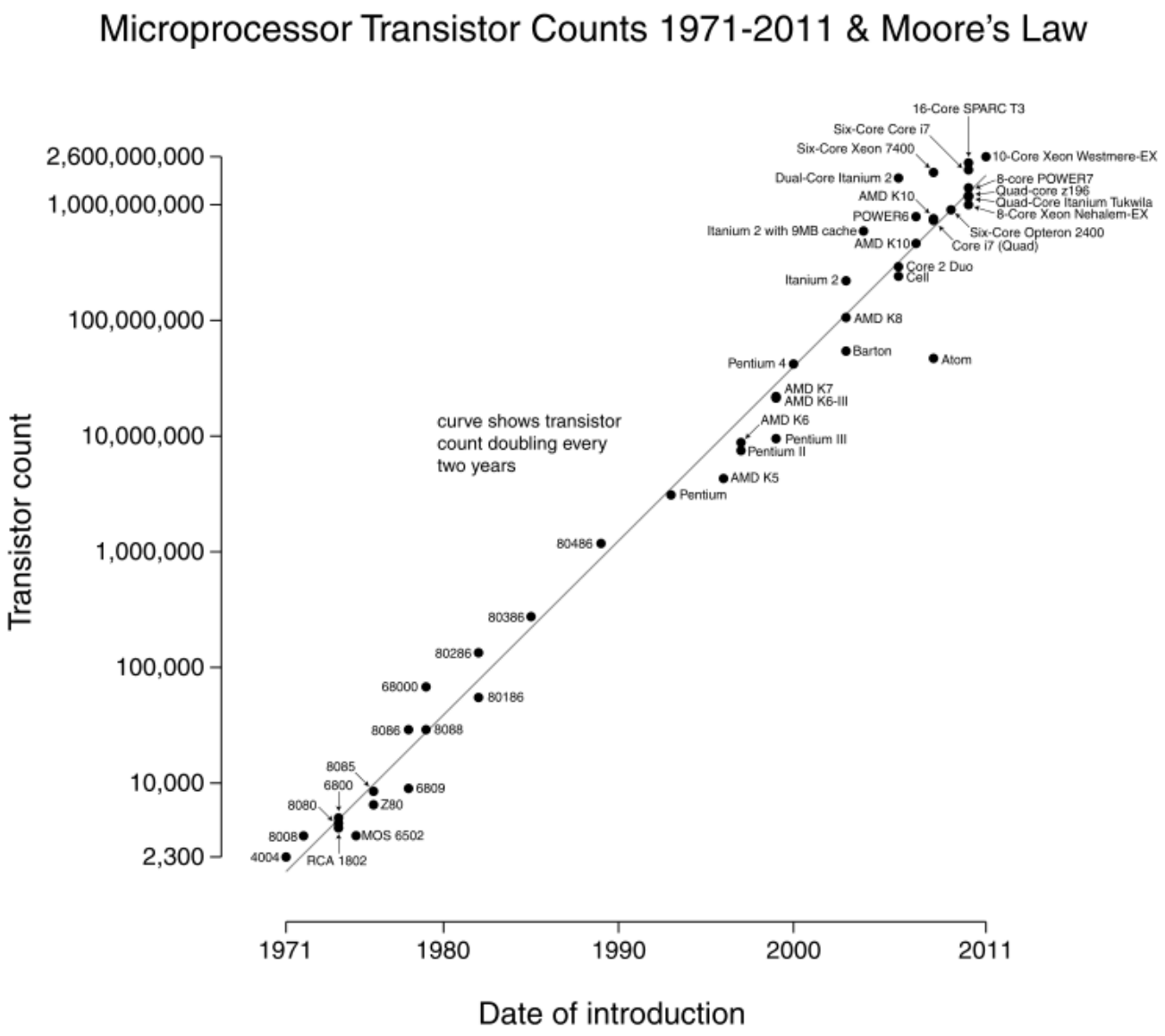
Joel Spolsky:

All non-trivial abstractions, to some degree, are leaky. *[...] Indeed, the abstractions we've created over the years do allow us to deal with new orders of complexity in software development that we didn't have to deal with ten or fifteen years ago [...]. Suddenly one day we need to figure out a problem where the abstraction leaked, and it takes 2 weeks.*

http://www.joelonsoftware.com/articles/LeakyAbstractions.html

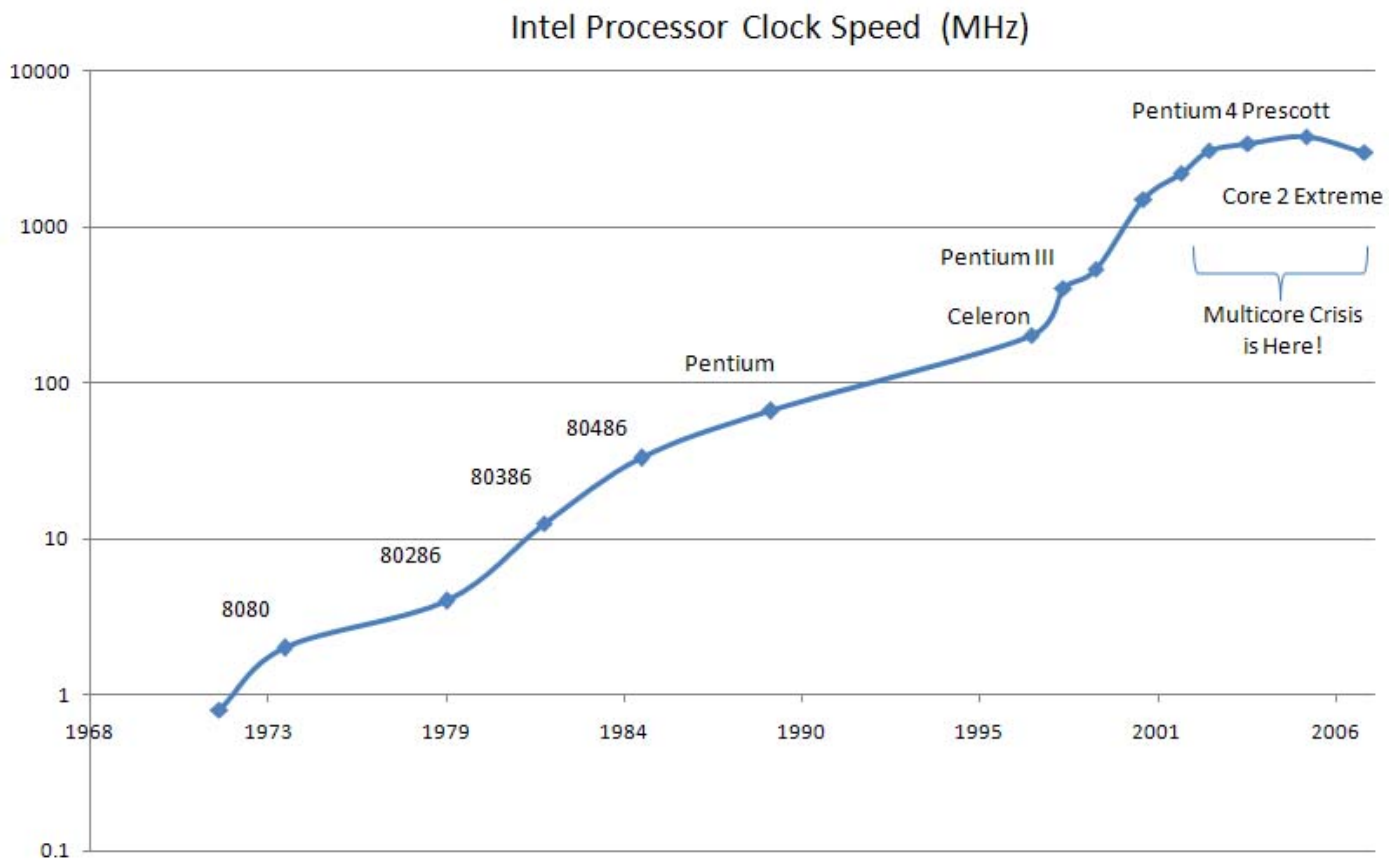
Past decades: Optimism

"Don't worry, better CPUs and compilers will take care of it"



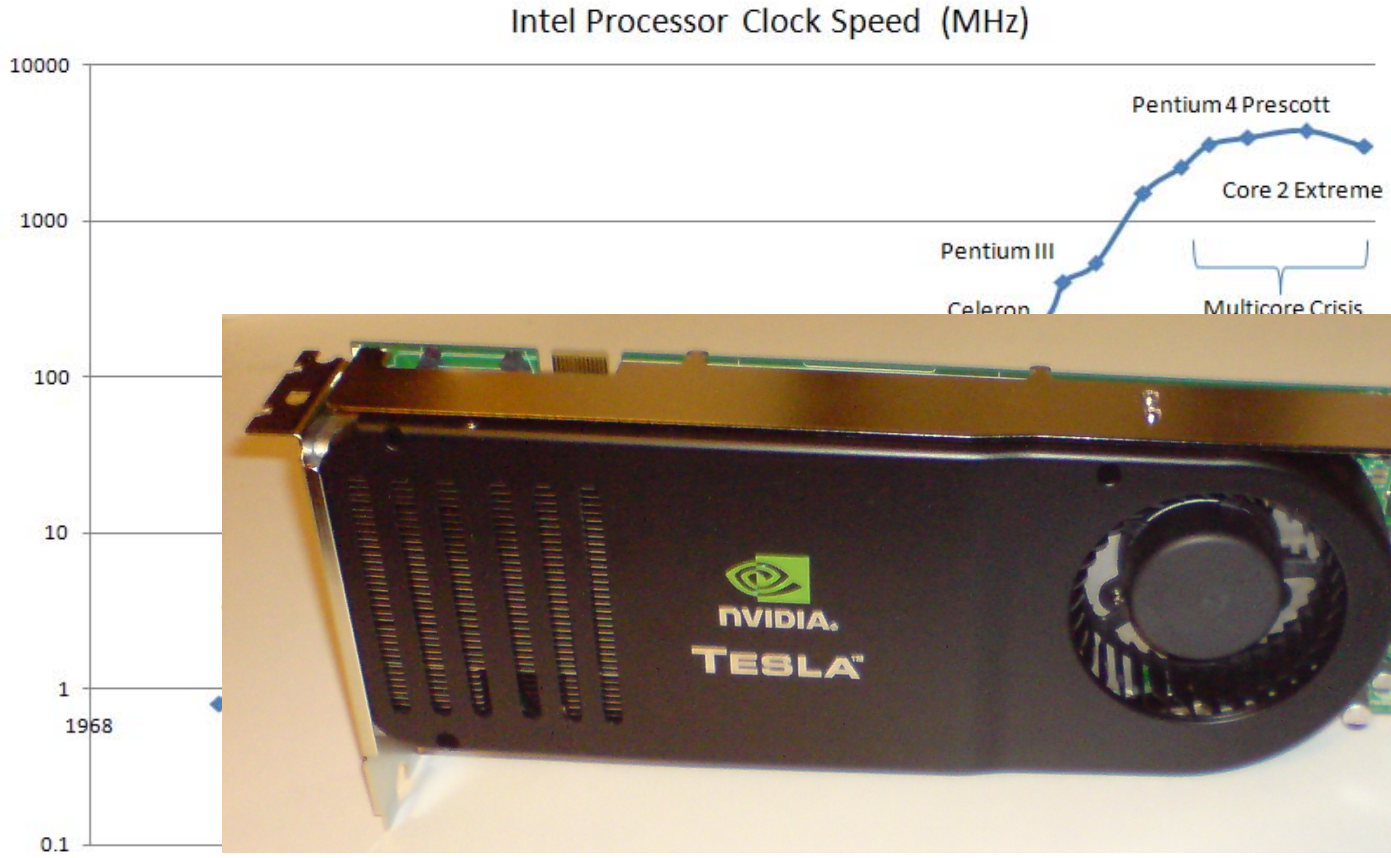
The reality

More complexity; algorithm must be adapted to hardware; compilers no magic bullet



The reality

More complexity; algorithm must be adapted to hardware; compilers no magic bullet



The basic abstractions for working with computers

Memory, encoding and pointers

Everything (data, program) is encoded in *memory bytes*, which (nowadays) are always 8 bits, allowing $2^8 = 256$ distinct values.

$$
\begin{aligned}
124_{10} &= 1 \cdot 10^2 + 2 \cdot 10^1 + 4 \cdot 10^0 \\
&= 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0
\end{aligned}
$$

Demo: `binary.c`

Memory, encoding and pointers

Everything (data, program) is encoded in *memory bytes*, which (nowadays) are always 8 bits, allowing $2^8 = 256$ distinct values.

$$
\begin{aligned}
124_{10} &= 1 \cdot 10^2 + 2 \cdot 10^1 + 4 \cdot 10^0 \\
&= 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 \\
&= 7 \cdot 16^1 + 12 \cdot 16^0 = \texttt{0x7c}
\end{aligned}
$$

Demo: `binary.c`

Pointers (as seen in C)

`int32_t x;`	Declare a 4-byte integer "x"

Pointers (as seen in C)

`int32_t x;`	Declare a 4-byte integer "x"
`x = 42;`	Memory of "x" now contains 42

Pointers (as seen in C)

`int32_t x;`	Declare a 4-byte integer "x"
`x = 42;`	Memory of "x" now contains 42
`int32_t *p;`	Declare an 8-byte **pointer to int** "p"

Pointers (as seen in C)

`int32_t x;`	Declare a 4-byte integer "x"
`x = 42;`	Memory of "x" now contains 42
`int32_t *p;`	Declare an 8-byte **pointer to int** "p"
`p = &x;`	Memory of "p" now contains 0x00007fffa63cc4b0

Pointers (as seen in C)

`int32_t x;`	Declare a 4-byte integer "x"
`x = 42;`	Memory of "x" now contains 42
`int32_t *p;`	Declare an 8-byte **pointer to int** "p"
`p = &x;`	Memory of "p" now contains 0x00007fffa63cc4b0
`*p = 10;`	Memory of "x" now contains 10

Pointers and arrays...

`int32_t *p`	"p" is a pointer to integers

Pointers and arrays...

`int32_t *p`	"p" is a pointer to integers
`p = malloc(40);`	Allocate space for 10 integers ("array")
	"p" contains 0x00007fffa63cc4b0

Pointers and arrays...

`int32_t *p`	"p" is a pointer to integers
`p = malloc(40);`	Allocate space for 10 integers ("array")
	"p" contains 0x00007fffa63cc4b0
`int32_t *u;`	

Pointers and arrays...

`int32_t *p`	"p" is a pointer to integers
`p = malloc(40);`	Allocate space for 10 integers ("array")
	"p" contains 0x00007fffa63cc4b0
`int32_t *u;`	
`u = p + 1;`	"u" contains 0x00007fffa63cc4b4

Pointers and arrays...

`int32_t *p`	"p" is a pointer to integers
`p = malloc(40);`	Allocate space for 10 integers ("array")
	"p" contains 0x00007fffa63cc4b0
`int32_t *u;`	
`u = p + 1;`	"u" contains 0x00007fffa63cc4b4
`*p = 10;`	Assign to first element of array

Pointers and arrays...

`int32_t *p`	"p" is a pointer to integers
`p = malloc(40);`	Allocate space for 10 integers ("array")
	"p" contains 0x00007fffa63cc4b0
`int32_t *u;`	
`u = p + 1;`	"u" contains 0x00007fffa63cc4b4
`*p = 10;`	Assign to first element of array
`*u = 20;`	Assign to second element of array

Pointers and arrays...

`int32_t *p`	"p" is a pointer to integers
`p = malloc(40);`	Allocate space for 10 integers ("array")
	"p" contains 0x00007fffa63cc4b0
`int32_t *u;`	
`u = p + 1;`	"u" contains 0x00007fffa63cc4b4
`*p = 10;`	Assign to first element of array
`*u = 20;`	Assign to second element of array
`p[0] = 10;`	Same as `*p = 10`

Pointers and arrays...

`int32_t *p`	"p" is a pointer to integers
`p = malloc(40);`	Allocate space for 10 integers ("array")
	"p" contains 0x00007fffa63cc4b0
`int32_t *u;`	
`u = p + 1;`	"u" contains 0x00007fffa63cc4b4
`*p = 10;`	Assign to first element of array
`*u = 20;`	Assign to second element of array
`p[0] = 10;`	Same as `*p = 10`
`p[1] = 20;`	Same as `*(p + 1) = 20`

Pointers and arrays...

`int32_t *p`	"p" is a pointer to integers
`p = malloc(40);`	Allocate space for 10 integers ("array")
	"p" contains 0x00007fffa63cc4b0
`int32_t *u;`	
`u = p + 1;`	"u" contains 0x00007fffa63cc4b4
`*p = 10;`	Assign to first element of array
`*u = 20;`	Assign to second element of array
`p[0] = 10;`	Same as `*p = 10`
`p[1] = 20;`	Same as `*(p + 1) = 20`
`p[10] = 30;`	Trampling on unallocated memory!

Multi-dimensional arrays

```
int32_t array = malloc(400);
int32_t nrows = 5, ncols = 20;

/* Treat memory in row-major ordering (C-order) */
for (int32_t i = 0; i < nrows; i++) {
  for (int32_t j = 0; j < ncols; j++) {
    array[i * ncols + j] = some_function_of(i, j);
  }
}

/* Treat memory in column-major ordering (Fortran-order) */
for (int32_t i = 0; i < nrows; i++) {
  for (int32_t j = 0; j < ncols; j++) {
    array[i + j * nrows] = some_function_of(i, j);
  }
}
```

Other uses for pointers

```
double square(double x) {
   return x * x;
}
b = square(a);

/* using a pointer to store result instead: */

double square_me(double *x) {
    *x = *x * *x;
}
square(&a);
```

Simple model of CPUs

- **Registers:** "Hardware variables". E.g., RAX/EAX, RBX/EAX, XMM0, . . . ; about 32 registers on 64-bit AMD/Intel CPUs

Simple model of CPUs

- **Registers:** "Hardware variables". E.g., RAX/EAX, RBX/EAX, XMM0, . . . ; about 32 registers on 64-bit AMD/Intel CPUs
- **Instructions:** Stream of numbers telling the CPU what to do.

Simple model of CPUs

- **Registers:** "Hardware variables". E.g., RAX/EAX, RBX/EAX, XMM0, . . . ; about 32 registers on 64-bit AMD/Intel CPUs
- **Instructions:** Stream of numbers telling the CPU what to do.
  `mov eax, [rsp]` Load from memory

Simple model of CPUs

- **Registers:** "Hardware variables". E.g., RAX/EAX, RBX/EAX, XMM0, . . . ; about 32 registers on 64-bit AMD/Intel CPUs
- **Instructions:** Stream of numbers telling the CPU what to do.

`mov eax, [rsp]`	Load from memory
`imul eax, eax`	Do something with register

Simple model of CPUs

- **Registers:** "Hardware variables". E.g., RAX/EAX, RBX/EAX, XMM0, . . . ; about 32 registers on 64-bit AMD/Intel CPUs
- **Instructions:** Stream of numbers telling the CPU what to do.

`mov eax, [rsp]`	Load from memory
`imul eax, eax`	Do something with register
`mov [rsp], eax`	Store to memory

Simple model of CPUs

- **Registers:** "Hardware variables". E.g., RAX/EAX, RBX/EAX, XMM0, . . . ; about 32 registers on 64-bit AMD/Intel CPUs
- **Instructions:** Stream of numbers telling the CPU what to do.

`mov eax, [rsp]`	Load from memory
`imul eax, eax`	Do something with register
`mov [rsp], eax`	Store to memory

Example: Disassemble `add_and_square.o`

Ways of using memory

Conventionally we divide memory into:

Ways of using memory

Conventionally we divide memory into:

- **Static**: Your program, global variables, constants

Ways of using memory

Conventionally we divide memory into:

- **Static**: Your program, global variables, constants
- **Heap**: Dynamically allocated/deallocated blocks
  - C: `malloc()`, `free()` functions
  - C++: `new`, `delete` keywords
  - Fortran: `allocate`, `deallocate` keywords
  - Modern languages: `new` + Garbage Collection

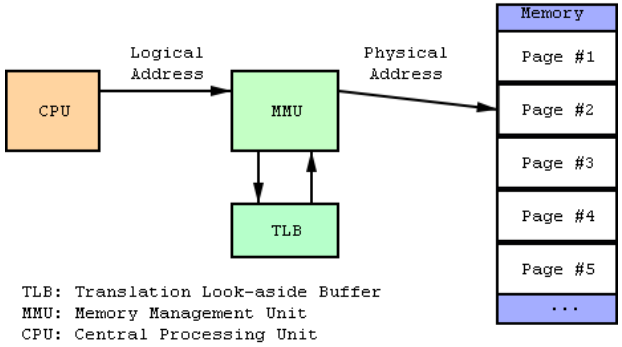
Ways of using memory

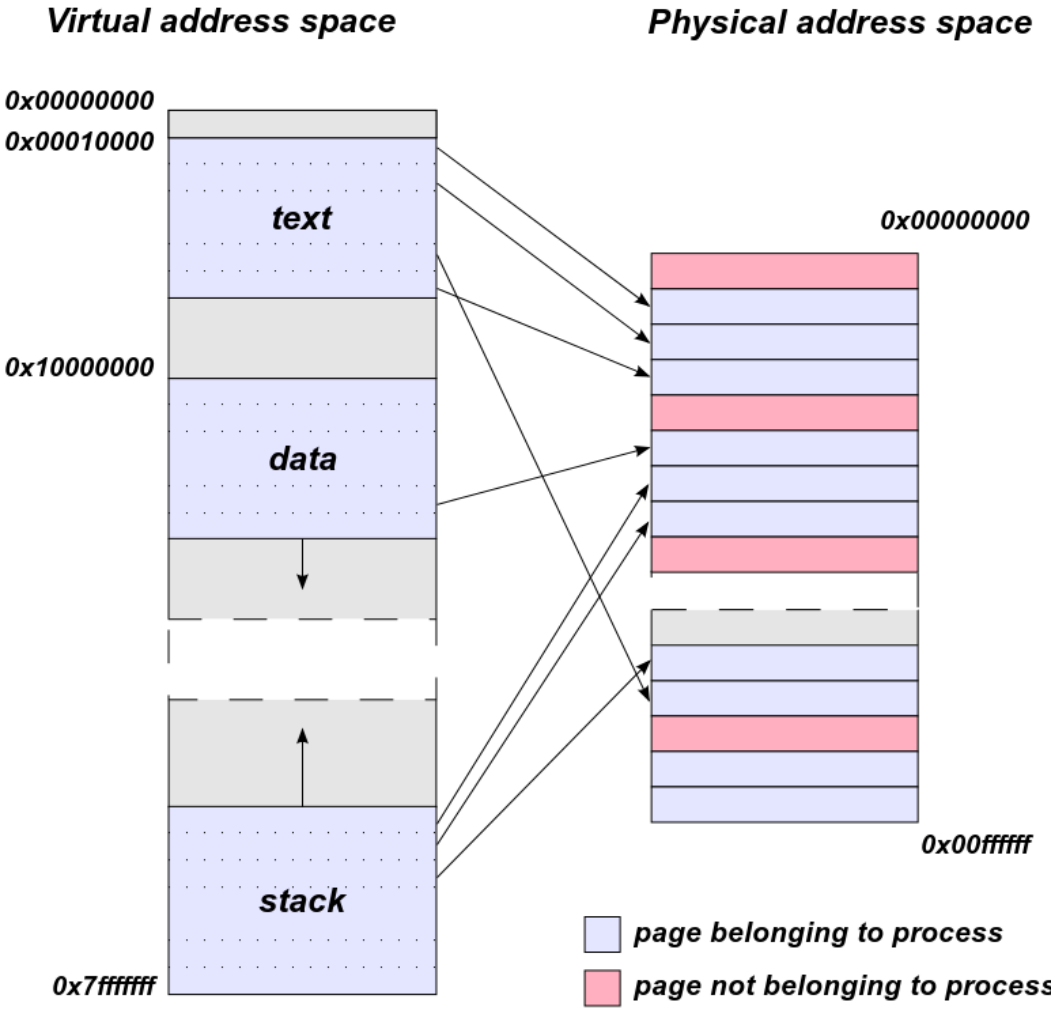
Conventionally we divide memory into:

- **Static**: Your program, global variables, constants
- **Heap**: Dynamically allocated/deallocated blocks
  - C: `malloc()`, `free()` functions
  - C++: `new`, `delete` keywords
  - Fortran: `allocate`, `deallocate` keywords
  - Modern languages: `new` + Garbage Collection
- **Stack**: Allocated at program startup
  - Implicitly used through program flow

Demo program: `recursion.c`

Virtual memory





Illustrations from Wikipedia

The memory bandwidth wall

Benchmarking

- **Problem:** What speed is your CPU *really* running at?
  1. Power saving
  2. "Turbo mode"

Benchmarking

- **Problem:** What speed is your CPU *really* running at?
  1. Power saving
  2. "Turbo mode"

  Solution: Disable these features

Benchmarking

- **Problem:** What speed is your CPU *really* running at?
  1. Power saving
  2. "Turbo mode"

  Solution: Disable these features

- **Problem:** Interruptions from the OS

Benchmarking

- **Problem:** What speed is your CPU *really* running at?
  1. Power saving
  2. "Turbo mode"

  Solution: Disable these features

- **Problem:** Interruptions from the OS

  Solution: Take many samples of *wall time*, then use *minimum*. For each sample, may have to repeat and average.

Benchmarking

- **Problem:** What speed is your CPU *really* running at?
  1. Power saving
  2. "Turbo mode"

  Solution: Disable these features

- **Problem:** Interruptions from the OS

  Solution: Take many samples of *wall time*, then use *minimum*. For each sample, may have to repeat and average.

- **Problem:** Does benchmark include memory bus etc. or just CPU?

Benchmarking

- **Problem:** What speed is your CPU *really* running at?
  1. Power saving
  2. "Turbo mode"

  Solution: Disable these features

- **Problem:** Interruptions from the OS

  Solution: Take many samples of *wall time*, then use *minimum*. For each sample, may have to repeat and average.

- **Problem:** Does benchmark include memory bus etc. or just CPU?
  Solution: Benchmark using all CPU cores and with realistically sized data sets

Benchmarking

- **Problem:** What speed is your CPU *really* running at?
  1. Power saving
  2. "Turbo mode"

  Solution: Disable these features

- **Problem:** Interruptions from the OS

  Solution: Take many samples of *wall time*, then use *minimum*. For each sample, may have to repeat and average.

- **Problem:** Does benchmark include memory bus etc. or just CPU?
  Solution: Benchmark using all CPU cores and with realistically sized data sets

- **Problem:** The compiler optimizing away the test code

Experiments exploring memory bus, cache, cache lines:

1. Memory bandwidth

Experiments exploring memory bus, cache, cache lines:

1. Memory bandwidth
2. Memory latency

Experiments exploring memory bus, cache, cache lines:

1. Memory bandwidth
2. Memory latency
3. Matrix transpose

Conclusion of experiments

Assuming an out-of-cache situation,

Conclusion of experiments

Assuming an out-of-cache situation,

- What matters today is *moving the data*. "Arithmetic is free".

Conclusion of experiments

Assuming an out-of-cache situation,

- What matters today is *moving the data*. "Arithmetic is free".
- Latency vs. bandwidth
  - Latency: Going "somewhere new" is 250x to 500x slower than arithmetic
  - Bandwidth: Reading "somewhere expected" is 30x to 100x slower than arithmetic

Conclusion of experiments

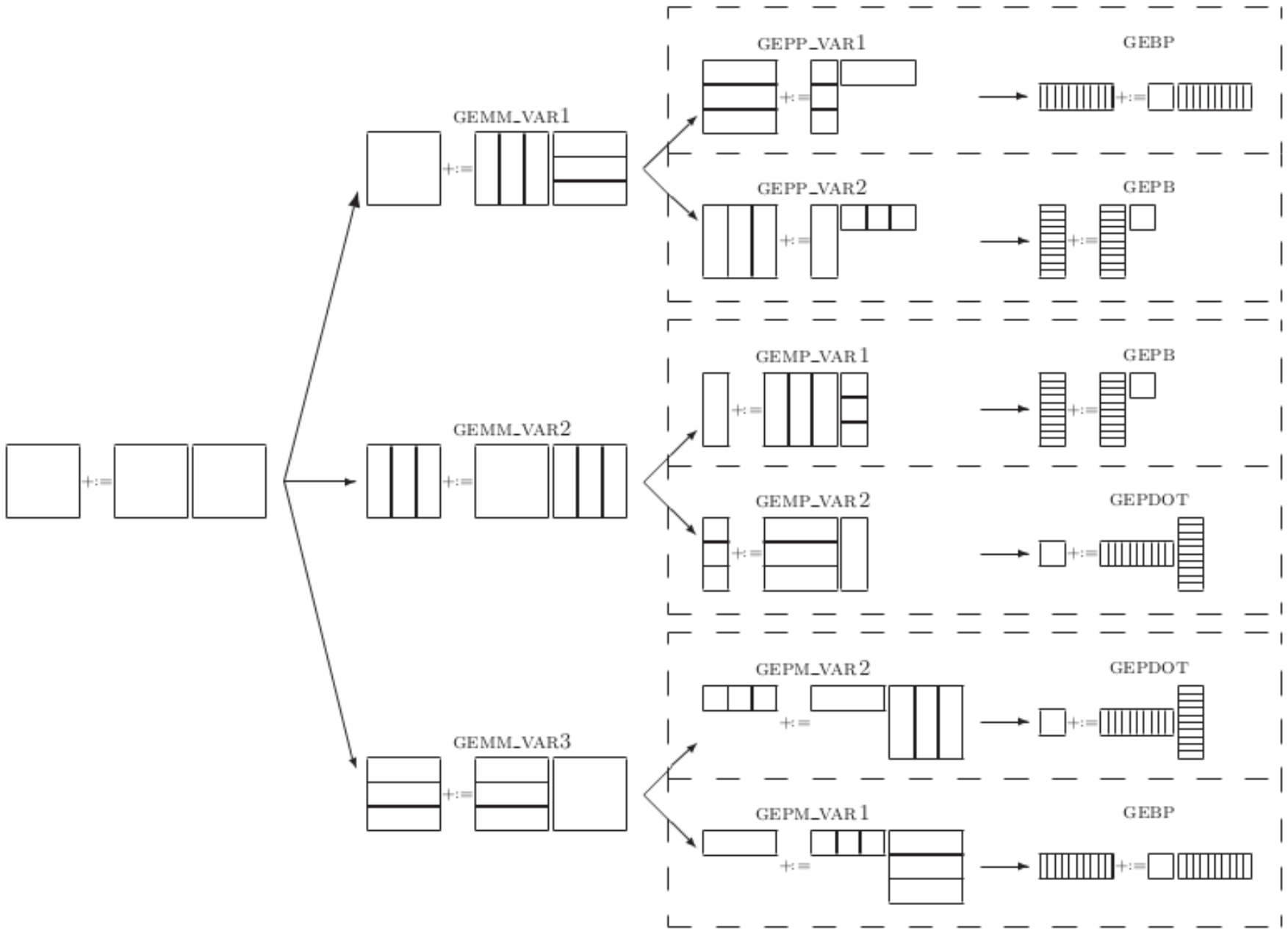
Assuming an out-of-cache situation,

- What matters today is *moving the data*. "Arithmetic is free".
- Latency vs. bandwidth
  - Latency: Going "somewhere new" is 250x to 500x slower than arithmetic
  - Bandwidth: Reading "somewhere expected" is 30x to 100x slower than arithmetic
- Very important to lay-out arrays the right way.
  - Cache-lines!

“Numbers every programmer should know”

L1 cache reference	0.5	ns		
Branch mispredict	5	ns		
L2 cache reference	7	ns		
Mutex lock/unlock	25	ns		
Main memory reference	100	ns		
Compress 1K bytes with Zippy	3,000	ns		
Send 1K bytes over 1 Gbps network	10,000	ns	0.01	ms
Read 4K randomly from SSD*	150,000	ns	0.15	ms
Read 1 MB sequentially from memory	250,000	ns	0.25	ms
Round trip within same datacenter	500,000	ns	0.5	ms
Read 1 MB sequentially from SSD*	1,000,000	ns	1	ms
Disk seek	10,000,000	ns	10	ms
Read 1 MB sequentially from disk	20,000,000	ns	20	ms
Send packet CA->Netherlands->CA	150,000,000	ns	150	ms

Matrix multiplication (Goto & van de Geijn, 2008)



Parallel computing and cluster hardware

Two models

- Threads sharing address space (OpenMP-style)

Two models

- Threads sharing address space (OpenMP-style)
  - Sounds convenient, but dangerous for performance (*abstraction leaking*)

Two models

- Threads sharing address space (OpenMP-style)
  - Sounds convenient, but dangerous for performance (*abstraction leaking*)
  - Non-trivial code is easy to mess up very badly: Dead-locks, races

Two models

- Threads sharing address space (OpenMP-style)
  - Sounds convenient, but dangerous for performance (*abstraction leaking*)
  - Non-trivial code is easy to mess up very badly: Dead-locks, races
  - One solution is to use ZeroMQ

Two models

- Threads sharing address space (OpenMP-style)
  - Sounds convenient, but dangerous for performance (*abstraction leaking*)
  - Non-trivial code is easy to mess up very badly: Dead-locks, races
  - One solution is to use ZeroMQ
- Processes passing messages (MPI-style)

Two models

- Threads sharing address space (OpenMP-style)
  - Sounds convenient, but dangerous for performance (*abstraction leaking*)
  - Non-trivial code is easy to mess up very badly: Dead-locks, races
  - One solution is to use ZeroMQ
- Processes passing messages (MPI-style)
  - Explicit, fewer hidden dangers

Two models

- Threads sharing address space (OpenMP-style)
  - Sounds convenient, but dangerous for performance (*abstraction leaking*)
  - Non-trivial code is easy to mess up very badly: Dead-locks, races
  - One solution is to use ZeroMQ
- Processes passing messages (MPI-style)
  - Explicit, fewer hidden dangers
  - Need to think about communication patterns

Two models

- Threads sharing address space (OpenMP-style)
  - Sounds convenient, but dangerous for performance (*abstraction leaking*)
  - Non-trivial code is easy to mess up very badly: Dead-locks, races
  - One solution is to use ZeroMQ
- Processes passing messages (MPI-style)
  - Explicit, fewer hidden dangers
  - Need to think about communication patterns
  - Need to duplicate shared resources; memory scarce on supercomputers...

Example setup typical of today

- Each CPU core runs 1-2 threads sharing L1 and L2 cache

Example setup typical of today

- Each CPU core runs 1-2 threads sharing L1 and L2 cache
- Each CPU die has 6-8 cores, sharing L3 cache, memory bus, RAM chips

Example setup typical of today

- Each CPU core runs 1-2 threads sharing L1 and L2 cache
- Each CPU die has 6-8 cores, sharing L3 cache, memory bus, RAM chips
- Each physical box ("node") contains 4 CPU dies, connected with a fast on-board network

Example setup typical of today

- Each CPU core runs 1-2 threads sharing L1 and L2 cache
- Each CPU die has 6-8 cores, sharing L3 cache, memory bus, RAM chips
- Each physical box ("node") contains 4 CPU dies, connected with a fast on-board network
- A cluster contains hundreds of boxes ("nodes"), connected with cables

Example setup typical of today

- Each CPU core runs 1-2 threads sharing L1 and L2 cache
- Each CPU die has 6-8 cores, sharing L3 cache, memory bus, RAM chips
- Each physical box ("node") contains 4 CPU dies, connected with a fast on-board network
- A cluster contains hundreds of boxes ("nodes"), connected with cables

Typical tradeoff: One process per die containing one thread per core.

Example setup typical of today

- Each CPU core runs 1-2 threads sharing L1 and L2 cache
- Each CPU die has 6-8 cores, sharing L3 cache, memory bus, RAM chips
- Each physical box ("node") contains 4 CPU dies, connected with a fast on-board network
- A cluster contains hundreds of boxes ("nodes"), connected with cables

Typical tradeoff: One process per die containing one thread per core.

- Fine-grained parallelism

Example setup typical of today

- Each CPU core runs 1-2 threads sharing L1 and L2 cache
- Each CPU die has 6-8 cores, sharing L3 cache, memory bus, RAM chips
- Each physical box ("node") contains 4 CPU dies, connected with a fast on-board network
- A cluster contains hundreds of boxes ("nodes"), connected with cables

Typical tradeoff: One process per die containing one thread per core.

- Fine-grained parallelism
- Coarse-grained parallelism through MPI

Example setup typical of today

- Each CPU core runs 1-2 threads sharing L1 and L2 cache
- Each CPU die has 6-8 cores, sharing L3 cache, memory bus, RAM chips
- Each physical box ("node") contains 4 CPU dies, connected with a fast on-board network
- A cluster contains hundreds of boxes ("nodes"), connected with cables

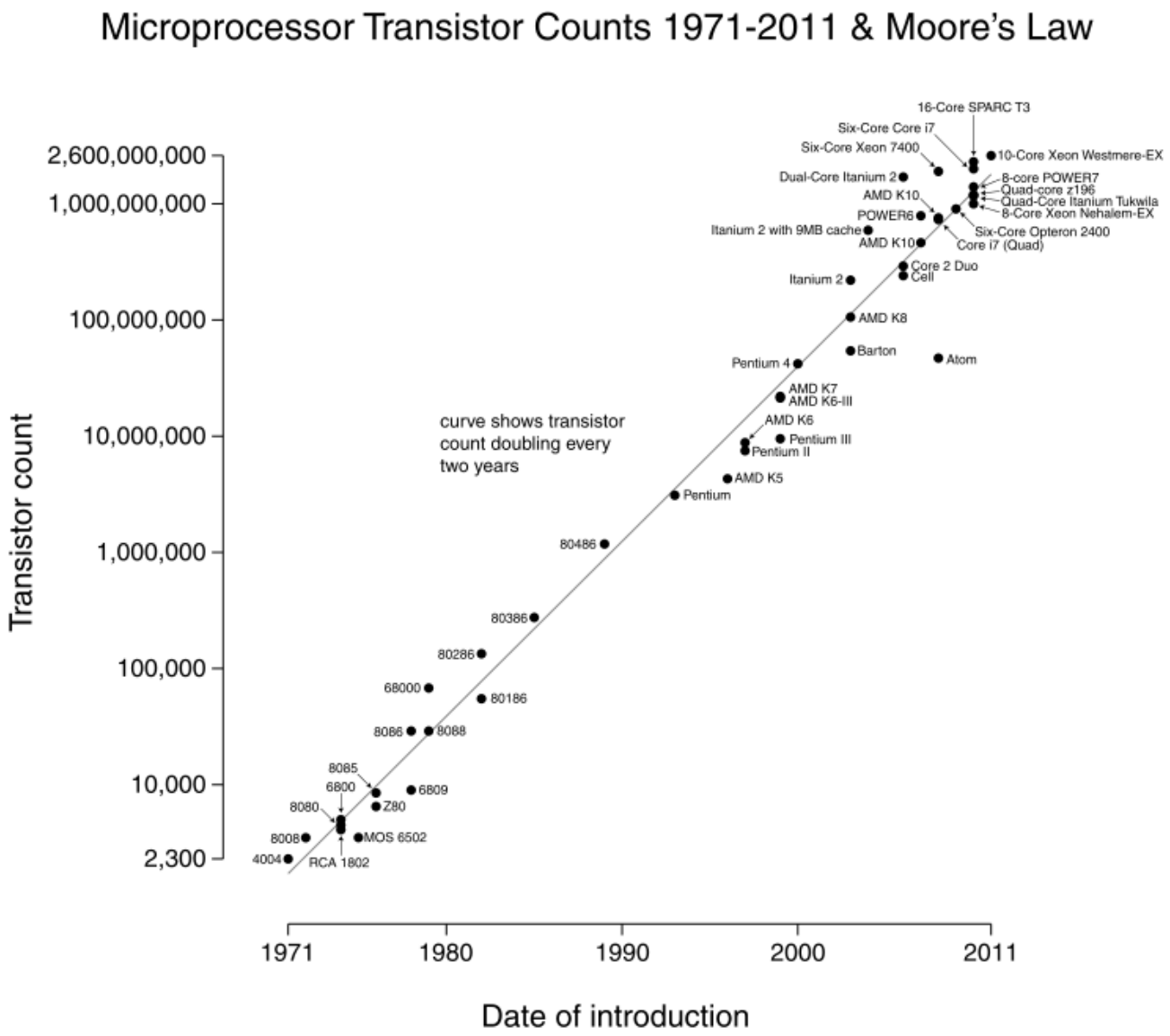
Typical tradeoff: One process per die containing one thread per core.

- Fine-grained parallelism
- Coarse-grained parallelism through MPI
- More re-use of resource data than

A more complicated picture of CPUs

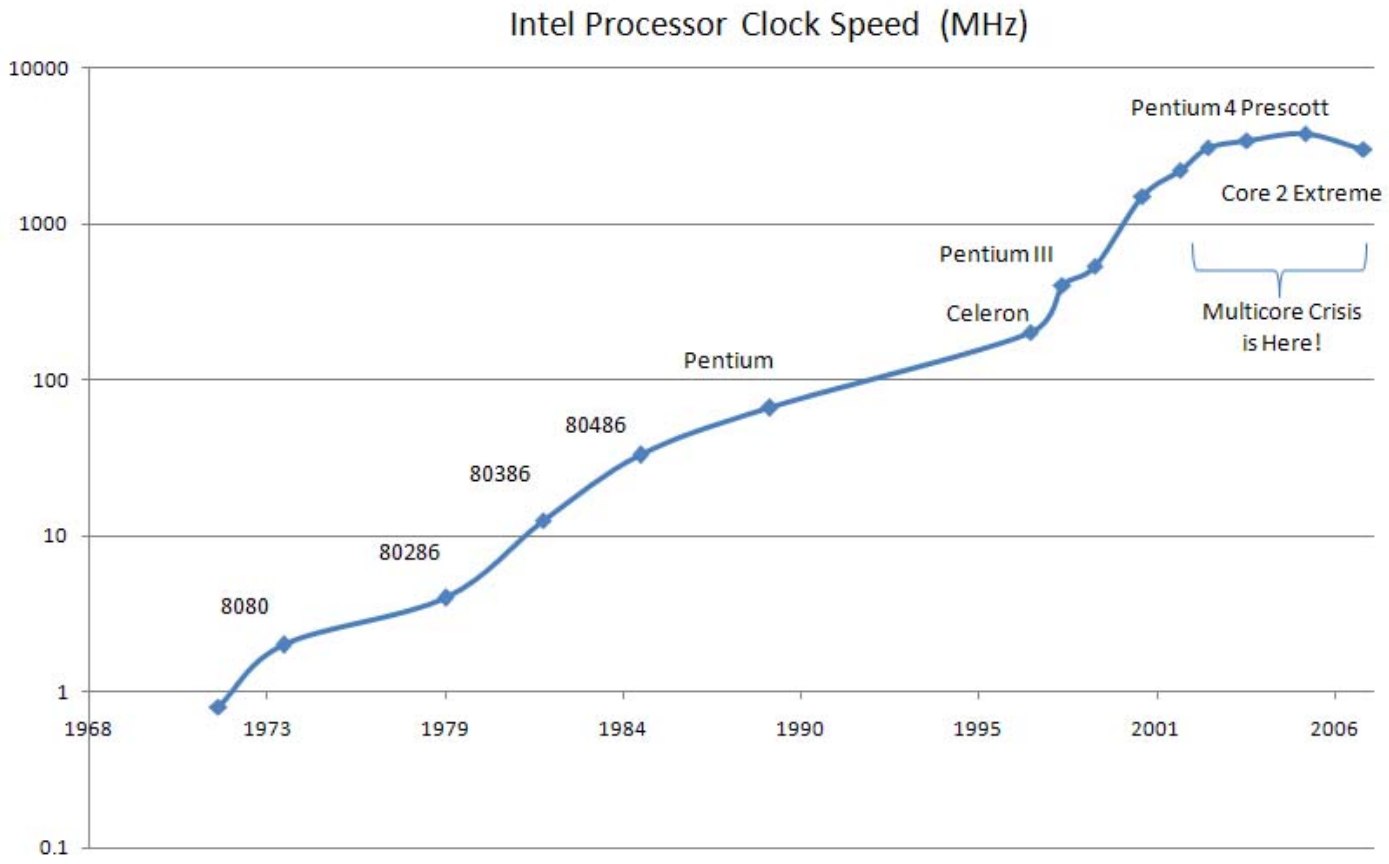
(+ some GPU)

CPUs getting more complicated



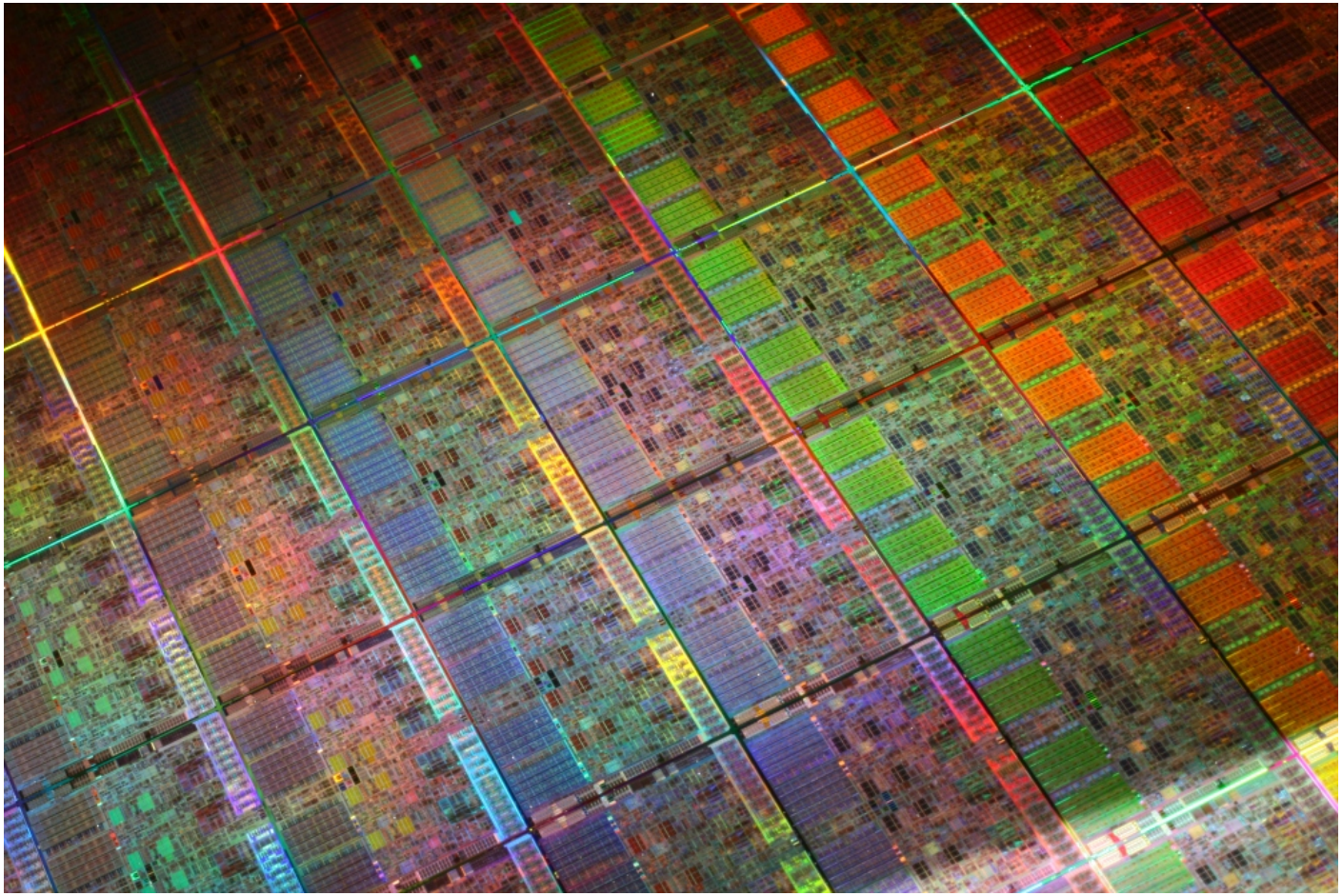
CPUs getting more complicated

But:

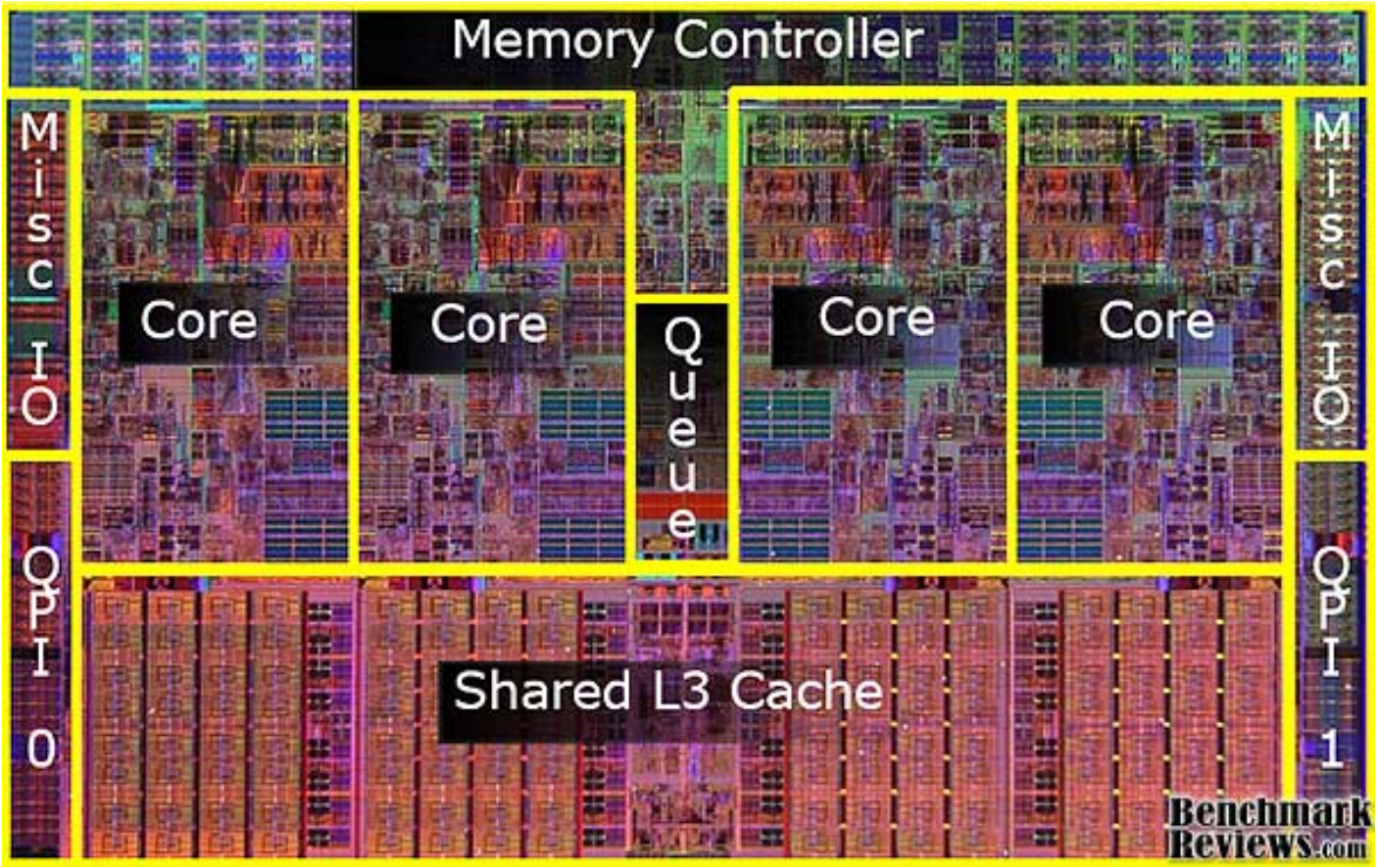


CPUs today

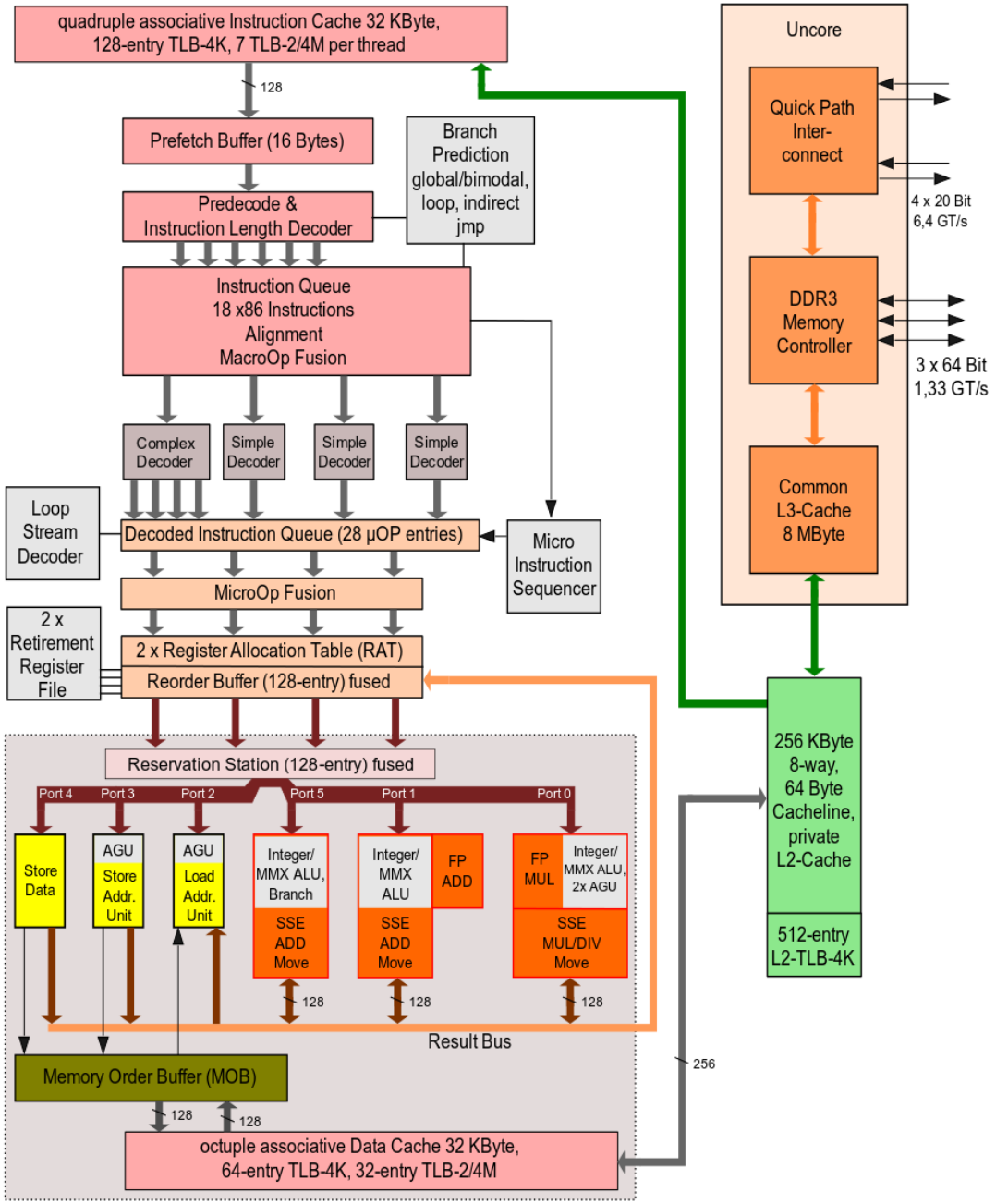
CPUs today



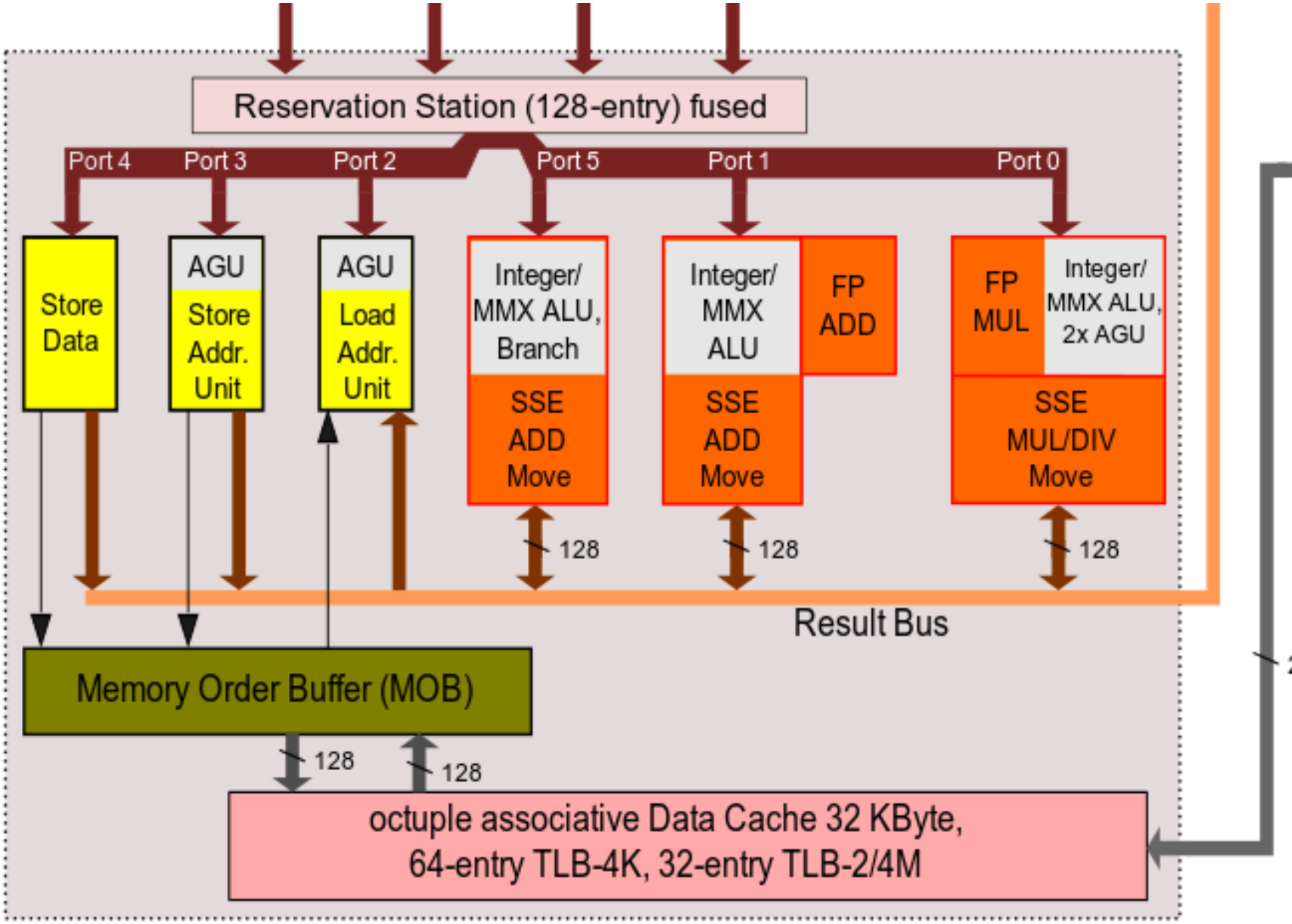
CPUs today


Memory Controller
Misc IO
Core
Core
Queue
Core
Core
Misc IO
QPI 0
Shared L3 Cache
QPI 1
Benchmark Reviews.com

Intel Nehalem microarchitecture



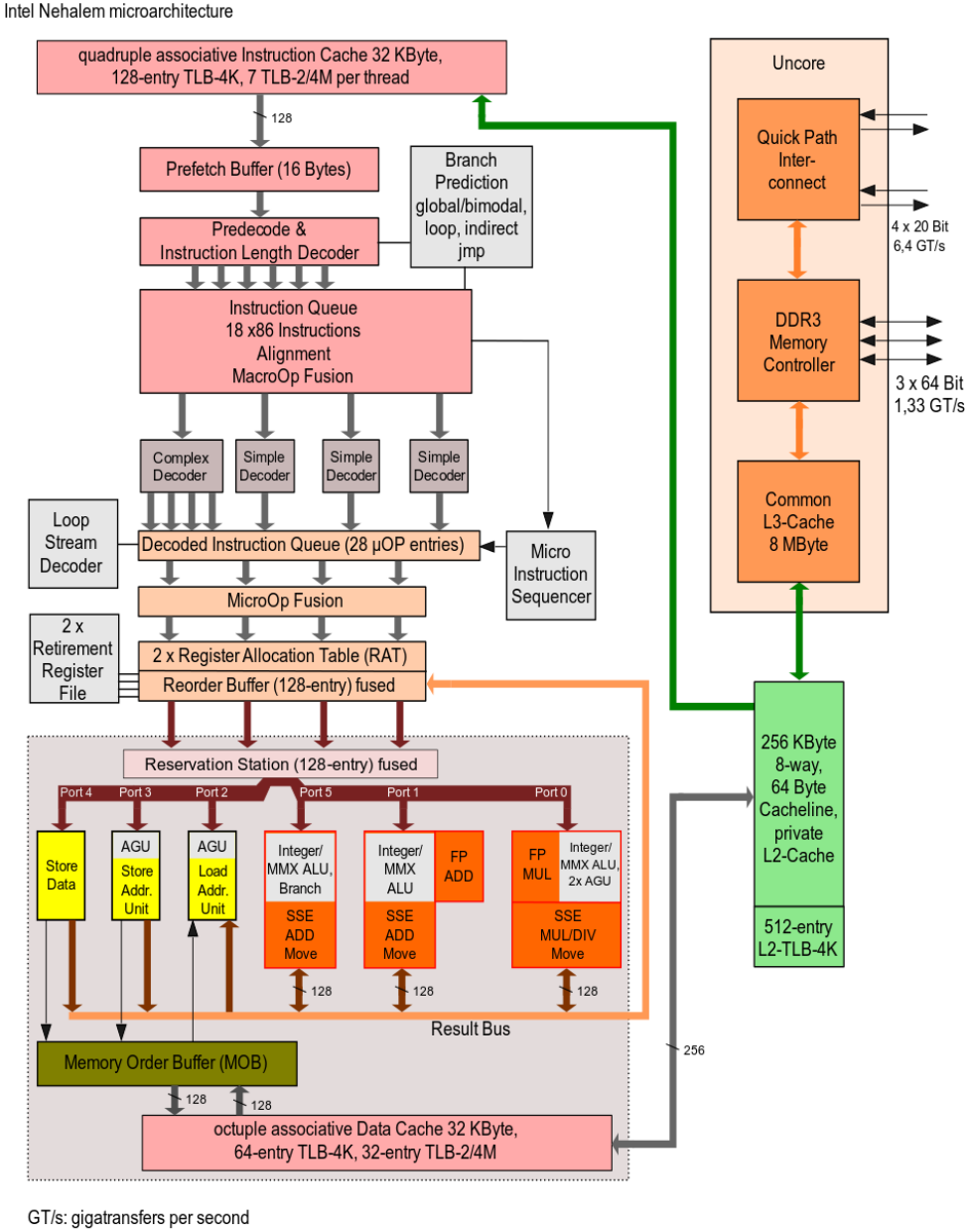
GT/s: gigatransfers per second



GT/s: gigatransfers per second

Out-of-order execution & register renaming

```
mov   eax, [rsp]
imul  eax, eax
mov   [rsp], eax
mov   eax, [rsp+0x4]
imul  eax, eax
mov   [rsp+0x4], eax
```



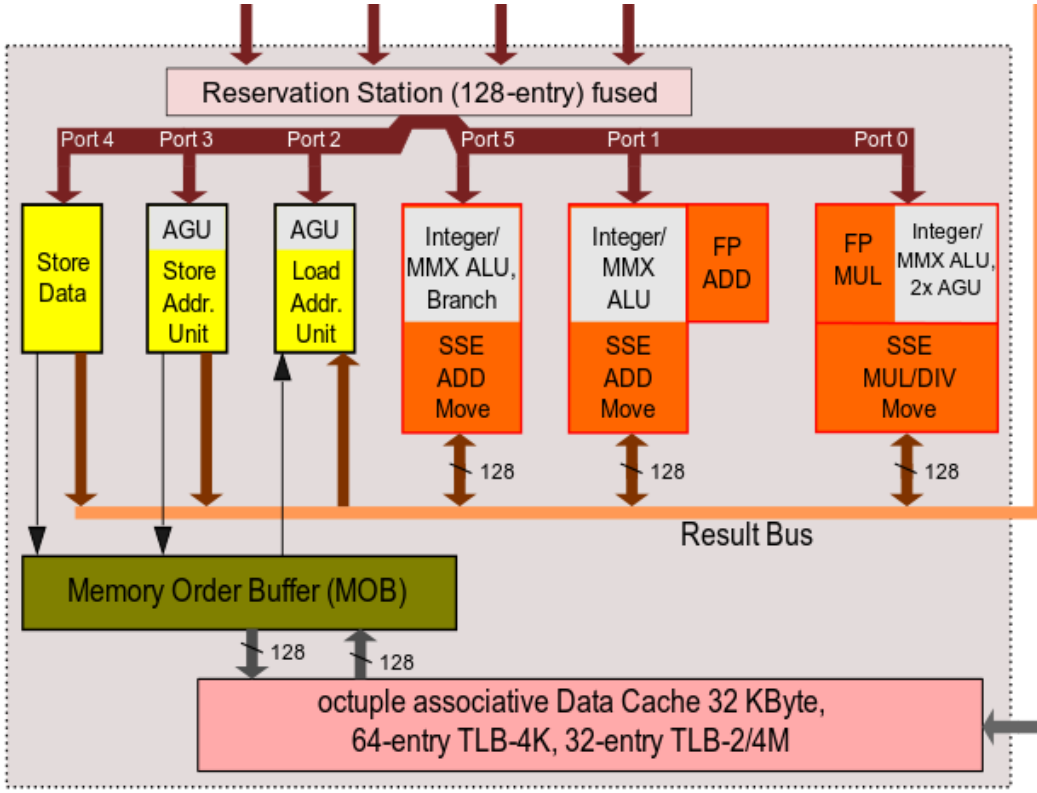
Pipelining

Not so good:

```
for (i = 2; i < n; ++i) {
   out[i] = a[i] * out[i - 2] + b[i] * out[i - 1];
}
```

Good:

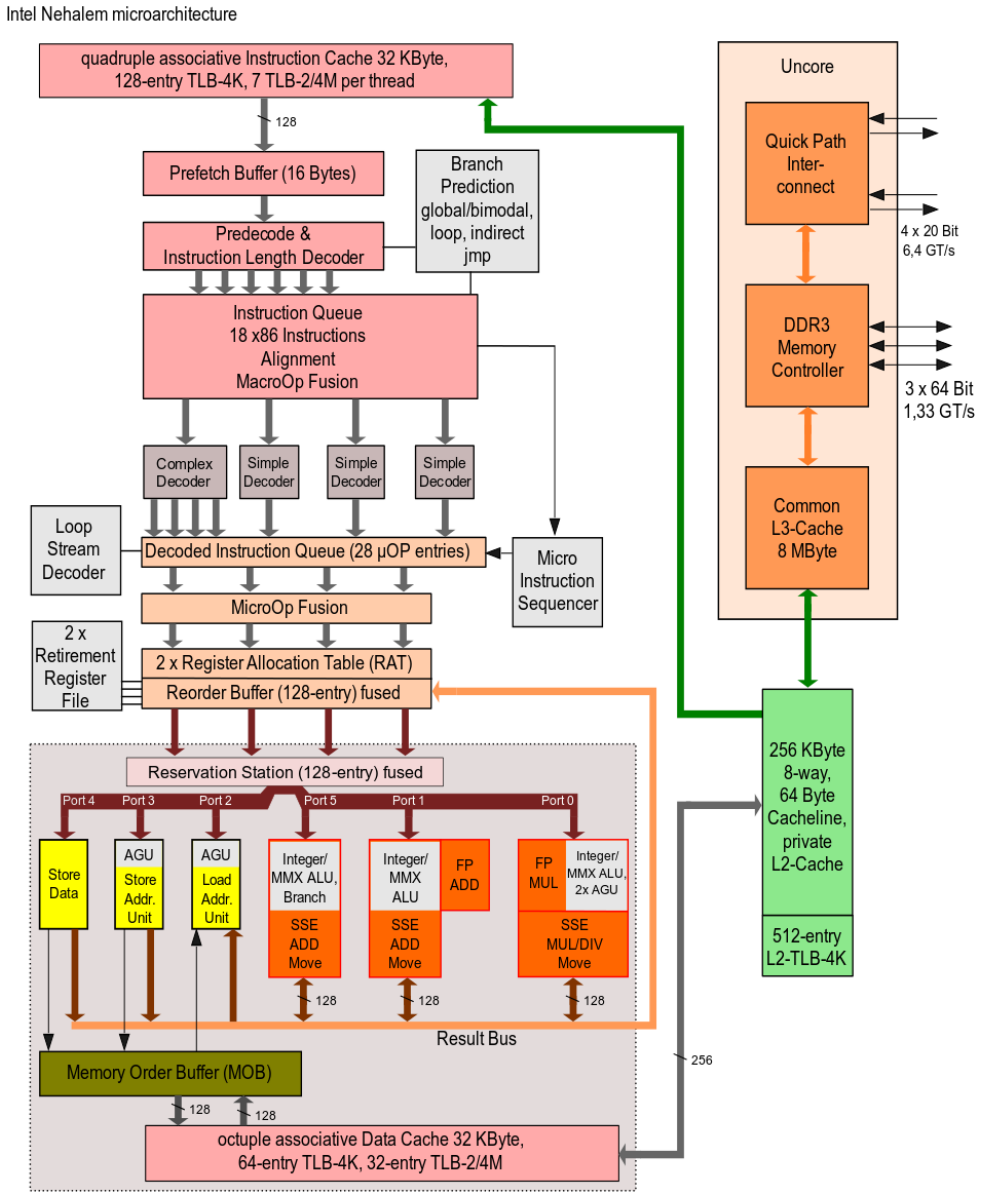
```
for (i = 0; i < n; ++i) {
   out[i] = a[i] + b[i];
}
```



GT/s: gigatransfers per second

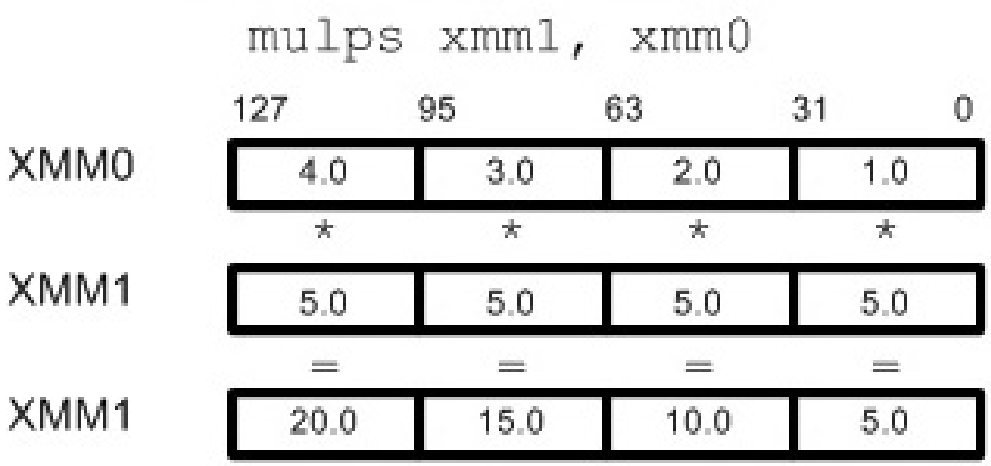
Branch prediction & speculative execution

```
x = f(y);
if (x < 100) {
   x *= 2;
} else {
  x -= 100;
}
```



Vector instructions

- SSE: 128-bit registers (4 $\times$ SP or 2 $\times$ DP)
- AVX: 256-bit registers (8 $\times$ SP or 4 $\times$ DP)



Experiment: `sse_transpose.c`

CPUs have become complicated...what about GPUs?

CPUs have become complicated...what about GPUs?

- Out-of-order execution: **No**
- Register renaming: **No**
- Branch prediction: **No**
- Speculative execution: **No**
- Cache: **Yes, but manual** (optional CPU-style cache available recently)
- Pipelining: **Yes**
- Vector registers: **Sort of...**

SM
Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (32,768 x 32-bit)
CUDA Core
Dispatch Port
Operand Collector
FP Unit
INT Unit
Result Queue
Core
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache
Tex
Texture Cache
PolyMorph Engine
Vertex Fetch
Tessellator
Viewport Transform
Attribute Setup
Stream Output

(slides taken out)

Non-Uniform Memory Architecture

```
dagss@owl3 $ numactl --hardware
available: 8 nodes (0-7)
node 0 size: 24216 MB
node 0 free: 22685 MB
node 1 size: 24240 MB
node 1 free: 24111 MB
node 2 size: 24240 MB
node 2 free: 24158 MB
<...>
node distances:
node   0   1   2   3   4   5   6   7
  0:  10  16  16  22  16  22  16  22
  1:  16  10  16  22  22  16  22  16
  2:  16  16  10  16  16  16  16  22
  3:  22  22  16  10  16  16  22  16
  4:  16  22  16  16  10  16  16  16
  5:  22  16  16  16  16  10  22  22
  6:  16  22  16  22  16  22  10  16
  7:  22  16  22  16  16  22  16  10
```

Non-Uniform Memory Architecture: AMD

```
dagss@owl3 $ bin/numabench 6
== Write (6 threads per node, GiB/s) ==
   5.9     4.9     2.2     1.8     2.1     1.8     4.0     3.6
   4.9     5.9     2.1     1.8     1.8     2.0     1.9     2.2
   2.1     2.2     5.9     4.9     2.2     2.2     2.0     1.8
   2.1     2.1     4.9     5.9     4.0     4.0     1.8     2.1
   3.7     3.6     2.2     3.9     5.9     4.9     2.2     3.9
   3.5     3.7     2.2     2.2     4.8     6.0     2.1     3.8
   2.2     1.9     3.7     3.5     2.2     1.8     5.9     4.9
   1.8     2.2     1.8     2.1     2.1     1.8     4.9     5.9
== Read (6 threads per node, GiB/s) ==
  11.1     4.7     2.1     2.0     2.4     2.5     2.1     1.8
   4.7    11.2     2.1     2.1     2.5     2.4     1.8     2.1
   2.1     2.1    11.2     4.8     2.1     2.2     2.4     1.7
   1.8     1.8     4.8    11.1     4.1     2.1     2.5     2.0
   1.9     1.8     2.1     4.1    11.2     4.4     2.1     2.1
   1.8     1.9     2.2     2.7     3.2    11.2     1.8     1.8
   2.6     1.8     1.9     1.7     2.1     2.1    11.1     4.4
   2.5     2.1     1.7     2.0     2.6     2.3     4.7    11.1
```

Non-Uniform Memory Architecture: Intel

```
dagss@mimosa $ bin/numabench 8
== Write (8 threads per node, GiB/s) ==
  10.1     7.2     5.4     5.1     4.6     5.4     5.3     7.3
   7.2    10.2     7.7     5.4     5.6     4.6     5.2     5.4
   5.4     7.6    10.0     7.2     5.4     5.2     4.5     5.5
   5.1     5.4     7.2    10.0     7.3     5.4     5.4     4.5
   4.5     5.4     5.3     7.2    10.0     7.2     5.4     5.1
   5.5     4.5     5.2     5.4     7.2    10.0     7.6     5.4
   5.3     5.2     4.5     5.5     5.4     7.6    10.0     7.2
   7.3     5.3     5.4     4.5     5.1     5.4     7.2    10.0
== Read (8 threads per node, GiB/s) ==
  22.0     7.6     8.2     7.4     7.1     7.8     8.0     8.5
   7.7    21.9     8.6     8.2     7.3     7.0     7.5     8.0
   8.1     9.0    21.8     7.6     8.0     7.4     7.0     7.4
   7.3     8.1     7.6    22.0     9.4     8.0     7.7     6.9
   7.0     7.7     7.9     9.2    21.9     7.6     8.2     7.3
   7.4     6.9     7.4     7.9     7.6    21.8     8.9     8.2
   8.0     7.5     7.0     7.3     8.1     8.6    21.8     7.6
   9.0     8.0     7.7     6.9     7.3     8.1     7.6    22.0
```

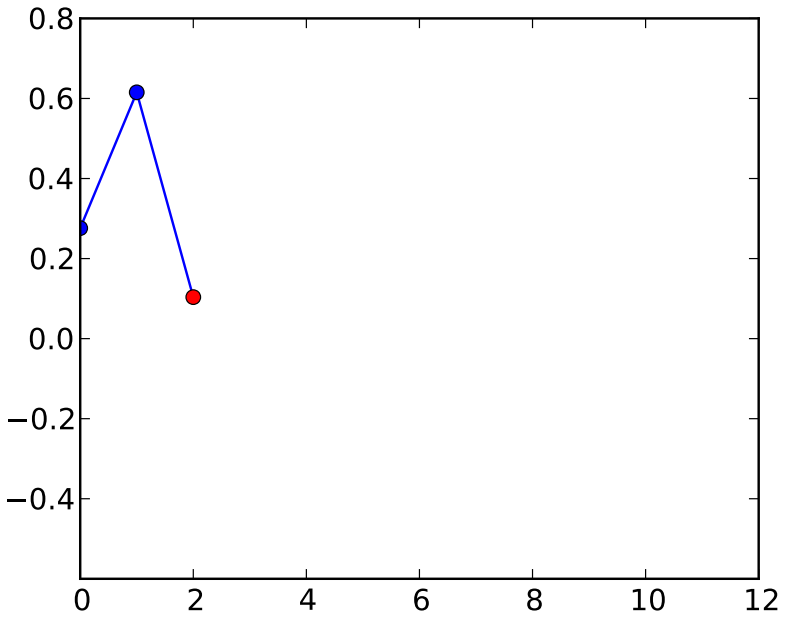
Example

Legendre transform:

$$q_i = \sum_{k=k_{\text{start}}}^{k_{\text{stop}}} \Lambda_{i,k} a_k$$

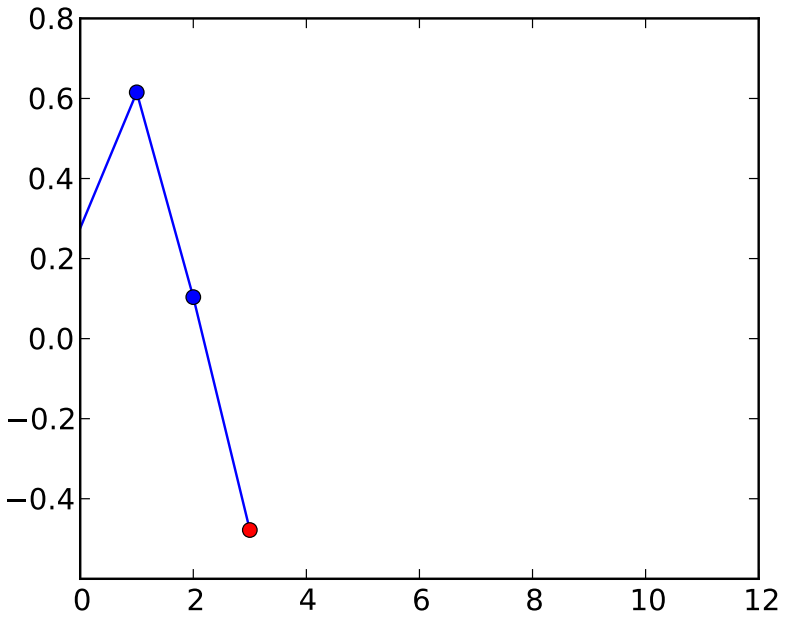
The Legendre functions by recurrence relation

$$\Lambda_{i,k+1} = (x_i^2 + \alpha_k)\beta_k \Lambda_{i,k} + \gamma_k \Lambda_{i,k-1}$$



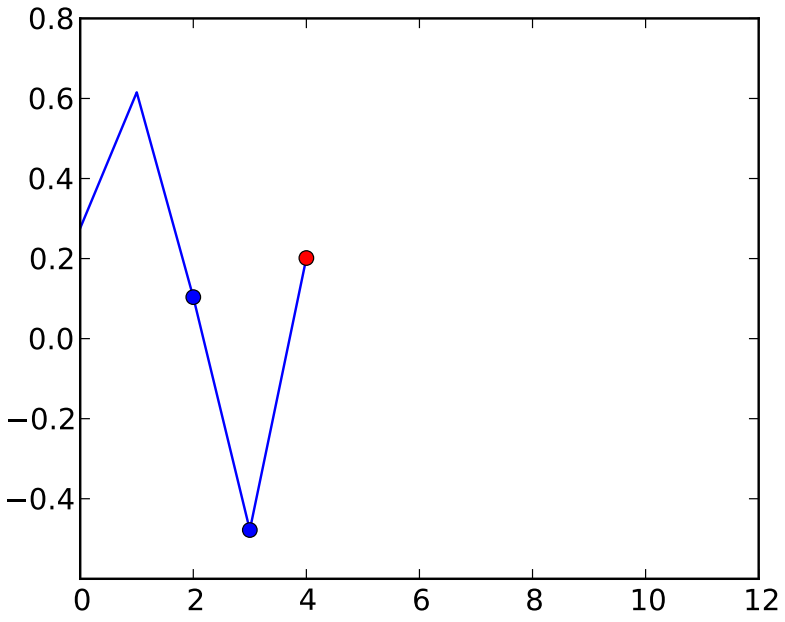
The Legendre functions by recurrence relation

$$\Lambda_{i,k+1} = (x_i^2 + \alpha_k)\beta_k \Lambda_{i,k} + \gamma_k \Lambda_{i,k-1}$$



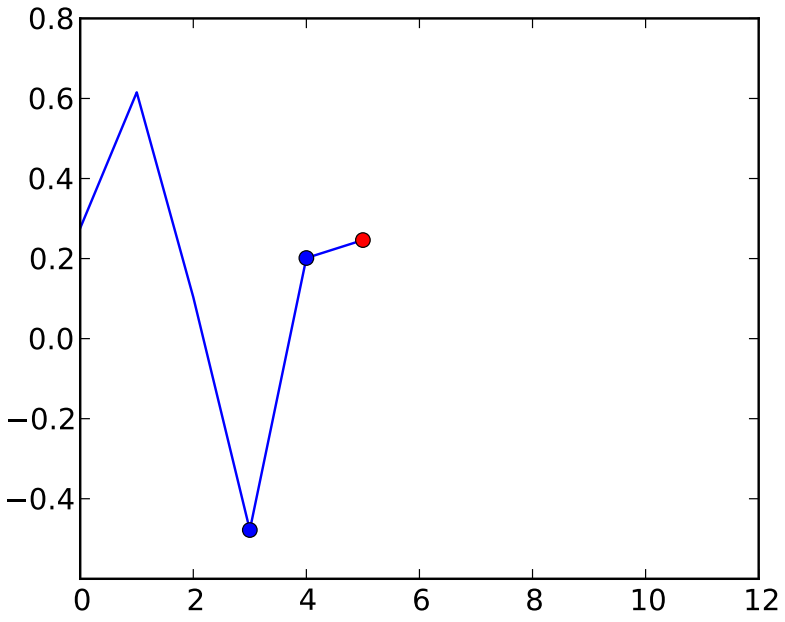
The Legendre functions by recurrence relation

$$\Lambda_{i,k+1} = (x_i^2 + \alpha_k)\beta_k \Lambda_{i,k} + \gamma_k \Lambda_{i,k-1}$$



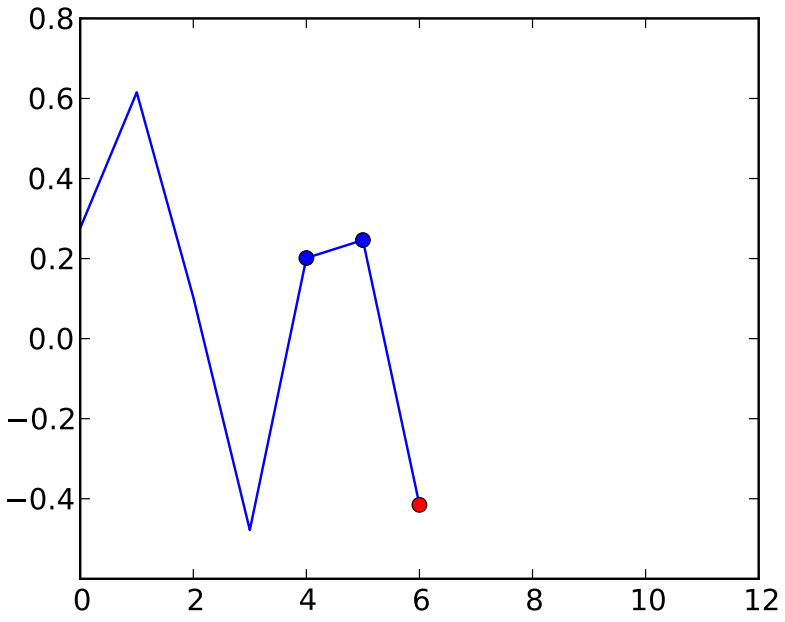
The Legendre functions by recurrence relation

$$\Lambda_{i,k+1} = (x_i^2 + \alpha_k)\beta_k \Lambda_{i,k} + \gamma_k \Lambda_{i,k-1}$$



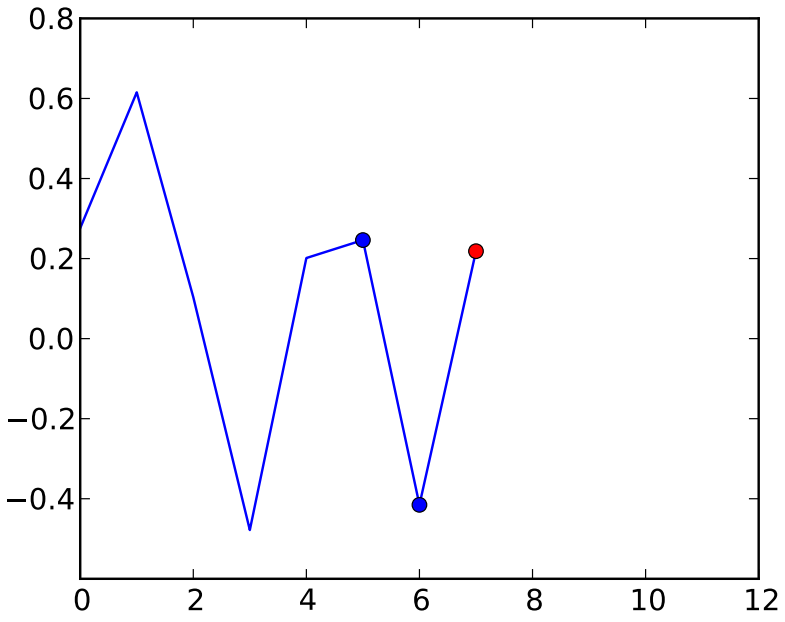
The Legendre functions by recurrence relation

$$\Lambda_{i,k+1} = (x_i^2 + \alpha_k)\beta_k \Lambda_{i,k} + \gamma_k \Lambda_{i,k-1}$$



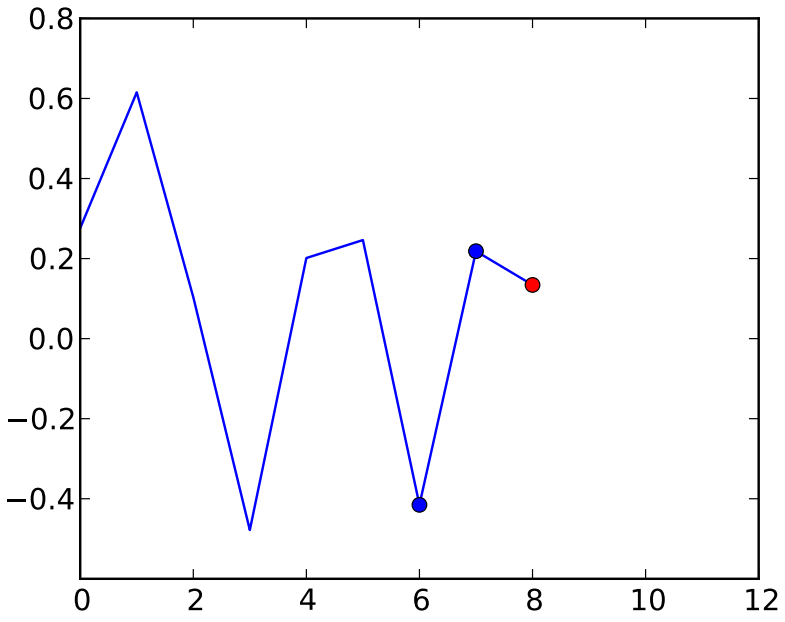
The Legendre functions by recurrence relation

$$\Lambda_{i,k+1} = (x_i^2 + \alpha_k)\beta_k \Lambda_{i,k} + \gamma_k \Lambda_{i,k-1}$$



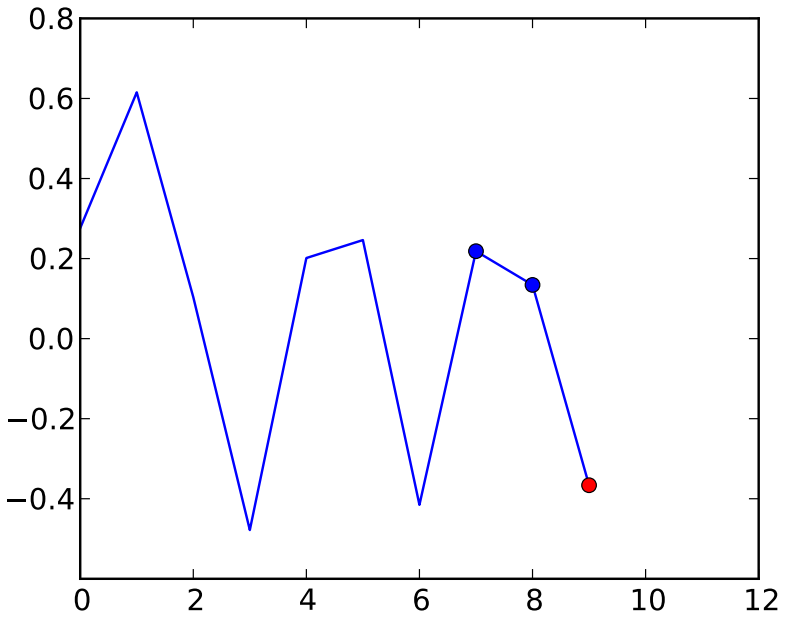
The Legendre functions by recurrence relation

$$\Lambda_{i,k+1} = (x_i^2 + \alpha_k)\beta_k \Lambda_{i,k} + \gamma_k \Lambda_{i,k-1}$$



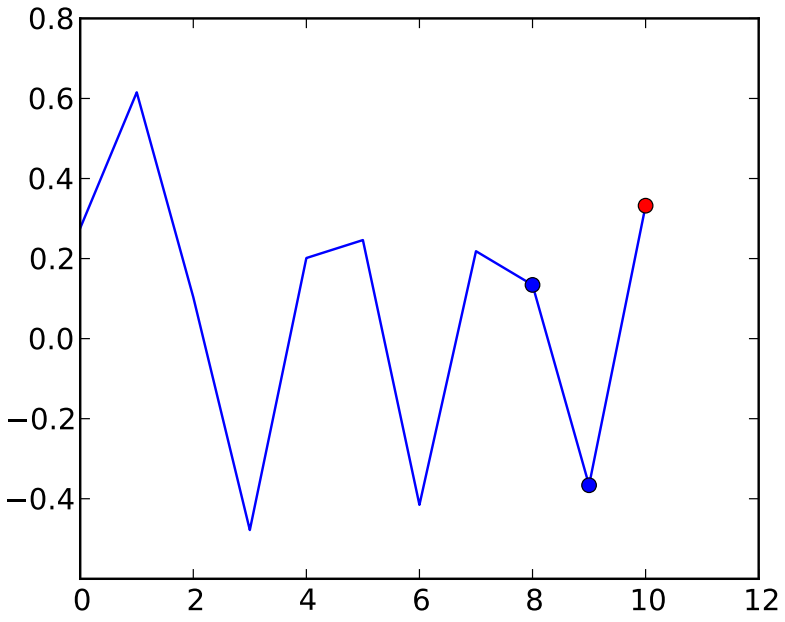
The Legendre functions by recurrence relation

$$\Lambda_{i,k+1} = (x_i^2 + \alpha_k)\beta_k \Lambda_{i,k} + \gamma_k \Lambda_{i,k-1}$$

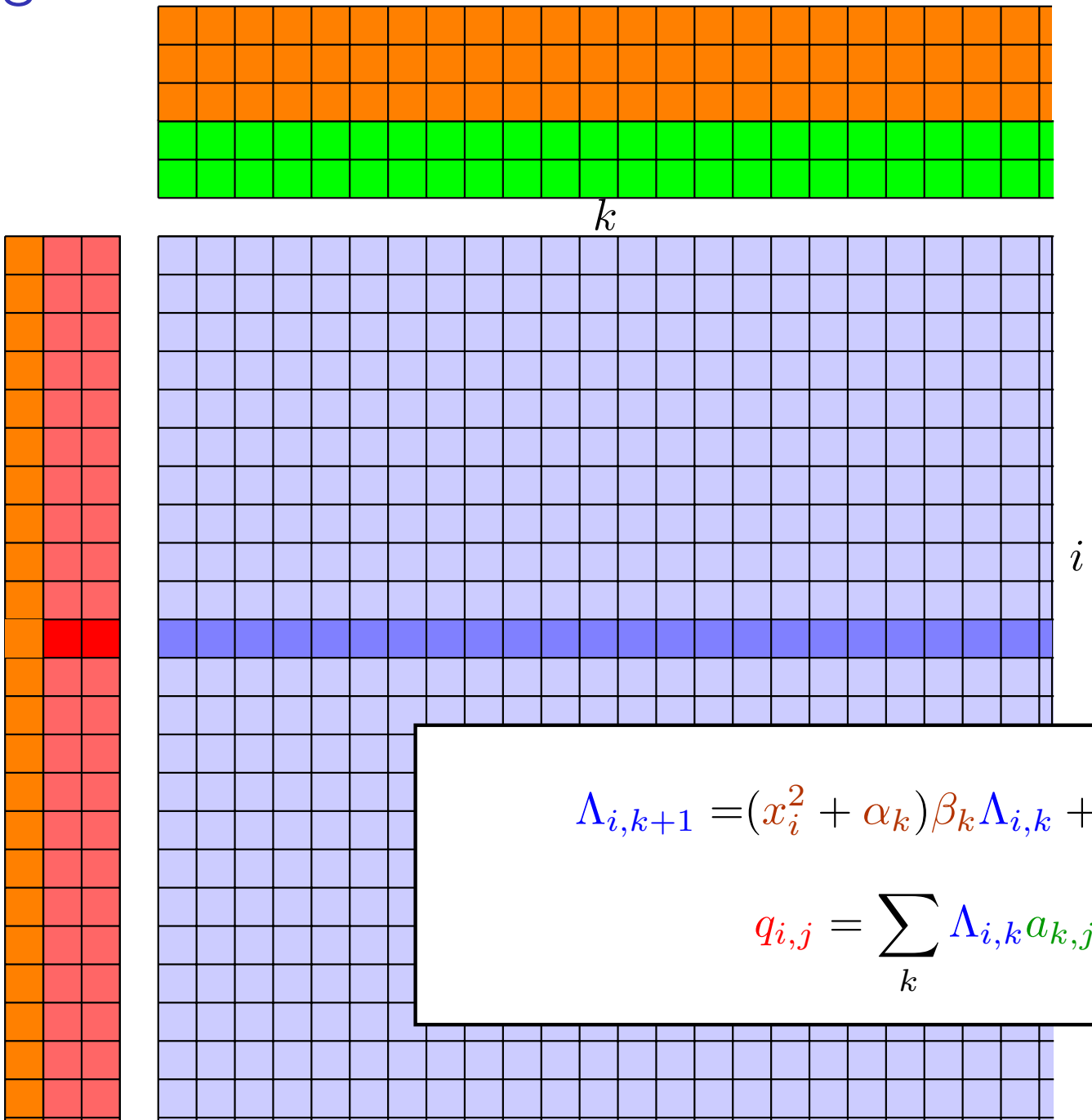


The Legendre functions by recurrence relation

$$\Lambda_{i,k+1} = (x_i^2 + \alpha_k)\beta_k \Lambda_{i,k} + \gamma_k \Lambda_{i,k-1}$$



Legendre transform



$$\Lambda_{i,k+1} = (x_i^2 + \alpha_k)\beta_k \Lambda_{i,k} + \gamma_k \Lambda_{i,k-1}$$

$$q_{i,j} = \sum_k \Lambda_{i,k} a_{k,j}$$

The SSE implementation

$$\Lambda_{i,k+1} = (x_i^2 + \alpha_k)\beta_k \Lambda_{i,k} + \gamma_k \Lambda_{i,k-1} \qquad q_{i,j} = \sum_k \Lambda_{i,k} a_{k,j}$$

4.28 GFLOP/s (46%)

$$\begin{bmatrix} \alpha_{17} \\ \alpha_{17} \end{bmatrix} \begin{bmatrix} \beta_{17} \\ \beta_{17} \end{bmatrix} \begin{bmatrix} \gamma_{17} \\ \gamma_{17} \end{bmatrix} \qquad \begin{bmatrix} a_{17,0} \\ a_{17,0} \end{bmatrix} \begin{bmatrix} a_{17,1} \\ a_{17,1} \end{bmatrix}$$

$$\begin{bmatrix} q_{12,0} & q_{12,1} \\ q_{13,0} & q_{13,1} \\ q_{14,0} & q_{14,1} \\ q_{15,0} & q_{15,1} \\ q_{16,0} & q_{16,1} \\ q_{17,0} & q_{17,1} \end{bmatrix} \mathrel{+}= \begin{bmatrix} \ddots & \vdots & \vdots & \vdots & \iddots \\ \cdots & \Lambda_{12,15} & \Lambda_{12,16} & \Lambda_{12,17} & \cdots \\ \cdots & \Lambda_{13,15} & \Lambda_{13,16} & \Lambda_{13,17} & \cdots \\ \cdots & \Lambda_{14,15} & \Lambda_{14,16} & \Lambda_{14,17} & \cdots \\ \cdots & \Lambda_{15,15} & \Lambda_{15,16} & \Lambda_{15,17} & \cdots \\ \cdots & \Lambda_{16,15} & \Lambda_{16,16} & \Lambda_{16,17} & \cdots \\ \cdots & \Lambda_{17,15} & \Lambda_{17,16} & \Lambda_{17,17} & \cdots \\ \iddots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} x_{12}^2 \\ x_{13}^2 \\ x_{14}^2 \\ x_{15}^2 \\ x_{16}^2 \\ x_{17}^2 \end{bmatrix}$$

The SSE implementation

$$\Lambda_{i,k+1} = (x_i^2 + \alpha_k)\beta_k \Lambda_{i,k} + \gamma_k \Lambda_{i,k-1} \qquad q_{i,j} = \sum_k \Lambda_{i,k} a_{k,j}$$

5.69 GFLOP/s (62%)

$$\begin{bmatrix} \alpha_{17} \\ \alpha_{17} \end{bmatrix} \begin{bmatrix} \beta_{17} \\ \beta_{17} \end{bmatrix} \begin{bmatrix} \gamma_{17} \\ \gamma_{17} \end{bmatrix} \qquad \begin{bmatrix} a_{17,0} \\ a_{17,0} \end{bmatrix} \begin{bmatrix} a_{17,1} \\ a_{17,1} \end{bmatrix}$$

$$\begin{bmatrix} q_{12,0} & q_{12,1} \\ q_{13,0} & q_{13,1} \\ q_{14,0} & q_{14,1} \\ q_{15,0} & q_{15,1} \\ q_{16,0} & q_{16,1} \\ q_{17,0} & q_{17,1} \end{bmatrix} \mathrel{+}= \begin{bmatrix} \ddots & \vdots & \vdots & \vdots & \iddots \\ \cdots & \Lambda_{12,15} & \Lambda_{12,16} & \Lambda_{12,17} & \cdots \\ \cdots & \Lambda_{13,15} & \Lambda_{13,16} & \Lambda_{13,17} & \cdots \\ \cdots & \Lambda_{14,15} & \Lambda_{14,16} & \Lambda_{14,17} & \cdots \\ \cdots & \Lambda_{15,15} & \Lambda_{15,16} & \Lambda_{15,17} & \cdots \\ \cdots & \Lambda_{16,15} & \Lambda_{16,16} & \Lambda_{16,17} & \cdots \\ \cdots & \Lambda_{17,15} & \Lambda_{17,16} & \Lambda_{17,17} & \cdots \\ \iddots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} x_{12}^2 \\ x_{13}^2 \\ x_{14}^2 \\ x_{15}^2 \\ x_{16}^2 \\ x_{17}^2 \end{bmatrix}$$

The SSE implementation

$$\Lambda_{i,k+1} = (x_i^2 + \alpha_k)\beta_k \Lambda_{i,k} + \gamma_k \Lambda_{i,k-1} \qquad q_{i,j} = \sum_k \Lambda_{i,k} a_{k,j}$$

6.46 GFLOP/s (70%)

$$\begin{bmatrix} \alpha_{17} \\ \alpha_{17} \end{bmatrix} \begin{bmatrix} \beta_{17} \\ \beta_{17} \end{bmatrix} \begin{bmatrix} \gamma_{17} \\ \gamma_{17} \end{bmatrix} \qquad \begin{bmatrix} a_{17,0} \\ a_{17,0} \end{bmatrix} \begin{bmatrix} a_{17,1} \\ a_{17,1} \end{bmatrix}$$

$$\begin{bmatrix} q_{12,0} & q_{12,1} \\ q_{13,0} & q_{13,1} \\ q_{14,0} & q_{14,1} \\ q_{15,0} & q_{15,1} \\ q_{16,0} & q_{16,1} \\ q_{17,0} & q_{17,1} \end{bmatrix} \mathrel{+}= \begin{bmatrix} \ddots & \vdots & \vdots & \vdots & \iddots \\ \cdots & \Lambda_{12,15} & \Lambda_{12,16} & \Lambda_{12,17} & \cdots \\ \cdots & \Lambda_{13,15} & \Lambda_{13,16} & \Lambda_{13,17} & \cdots \\ \cdots & \Lambda_{14,15} & \Lambda_{14,16} & \Lambda_{14,17} & \cdots \\ \cdots & \Lambda_{15,15} & \Lambda_{15,16} & \Lambda_{15,17} & \cdots \\ \cdots & \Lambda_{16,15} & \Lambda_{16,16} & \Lambda_{16,17} & \cdots \\ \cdots & \Lambda_{17,15} & \Lambda_{17,16} & \Lambda_{17,17} & \cdots \\ \iddots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{bmatrix} x_{12}^2 \\ x_{13}^2 \\ x_{14}^2 \\ x_{15}^2 \\ x_{16}^2 \\ x_{17}^2 \end{bmatrix}$$