IMM

Application Tuning

DTU

# Getting good performance from your application

Tuning techniques for serial programs on cache-based computer systems

# Overview

- Introduction
- Memory Hierarchy
- General Optimization Techniques
- 
- Compilers
- Analysis Tools
- Tuning Guide



# Introduction

# Introduction

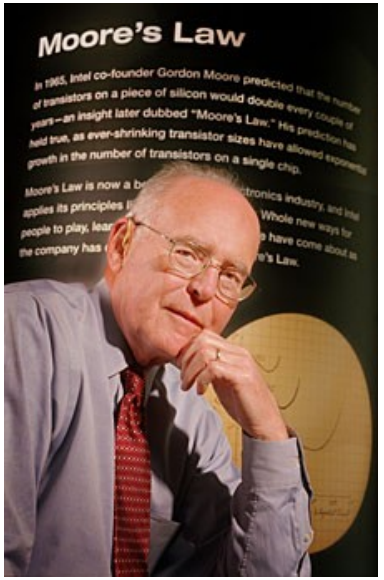
## Moore's Law

- Popular version:
  - *"CPU speed usually doubles every 18 months."*
- More correct version:
  - *"The number of transistors per integrated circuit will double every 18 months."*

# Introduction

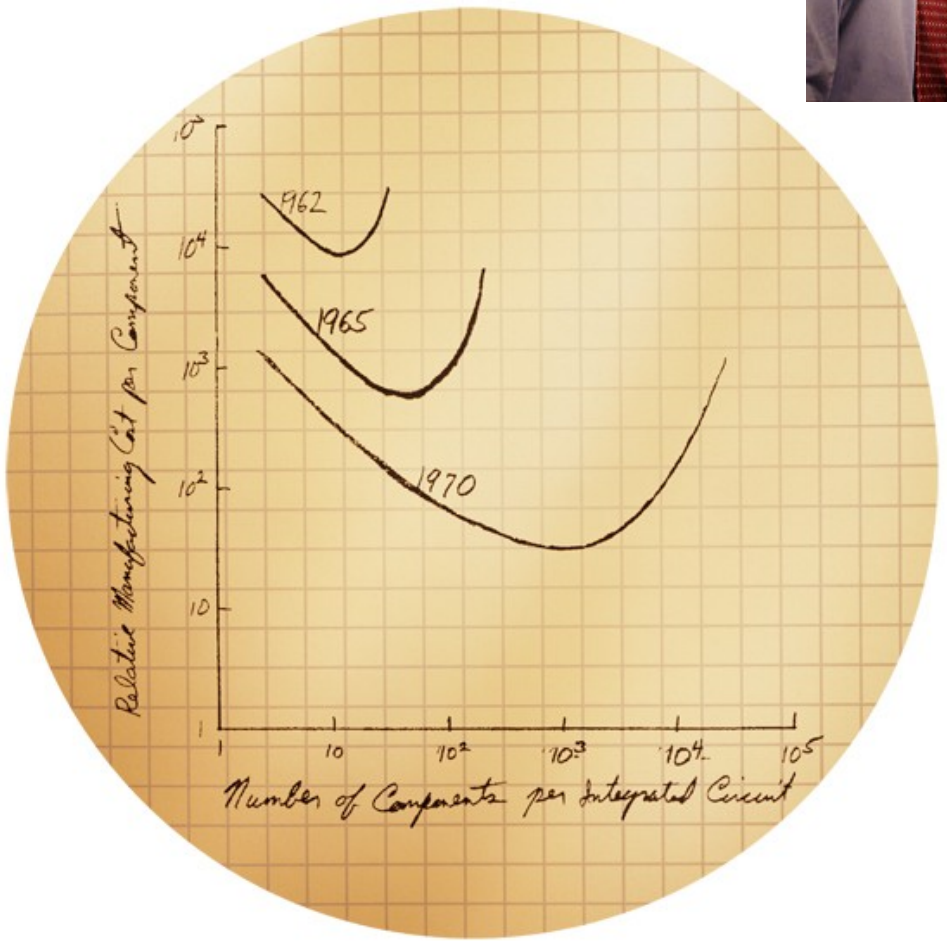


## Gordon Moore – co-founder of Intel



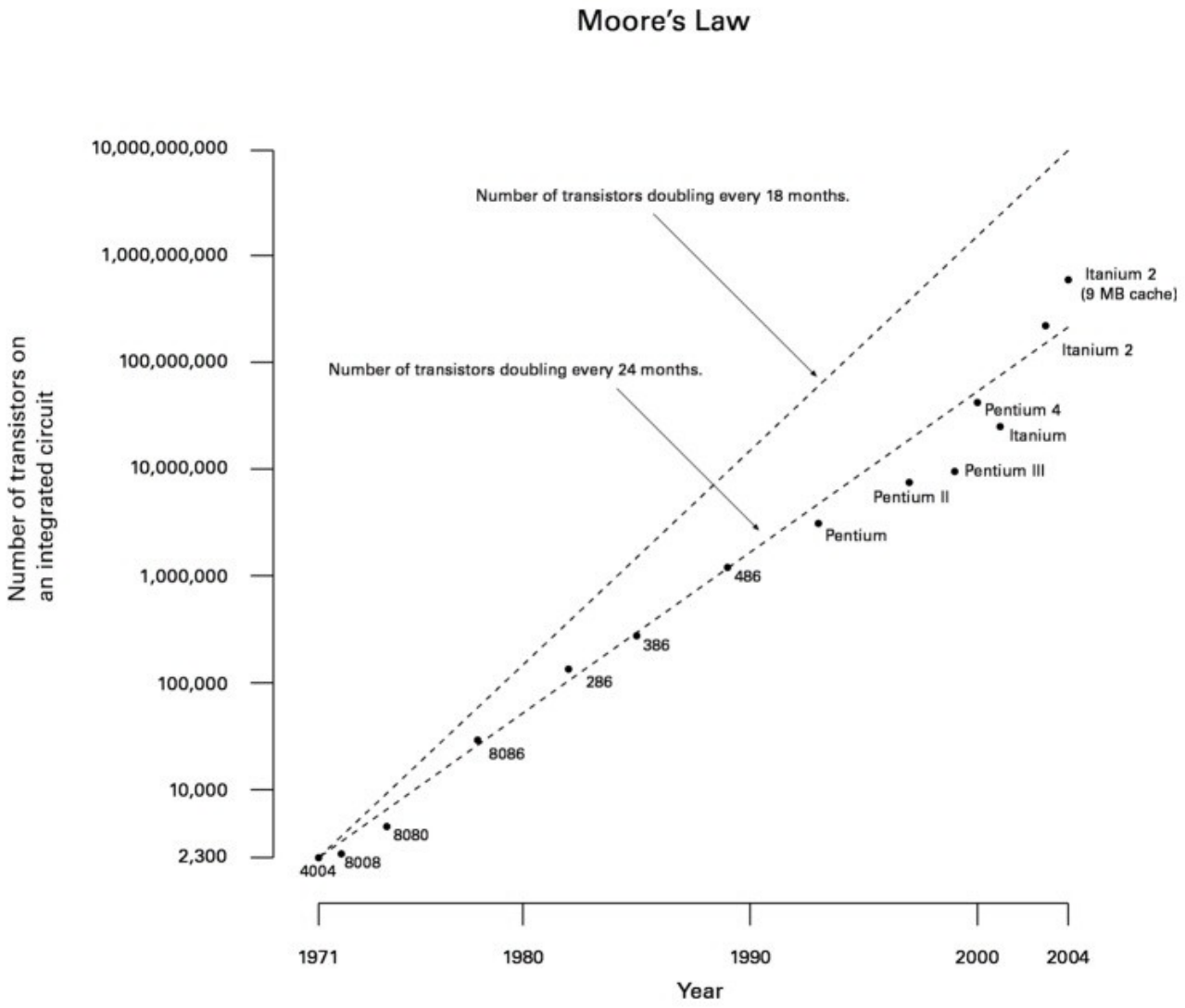

*"I never said 18 months. I said one year, and then two years ... Moore's Law has been the name given to everything that changes exponentially. ... If Gore invented the Internet, I invented the exponential."*

- Gordon Moore in an interview (2000)



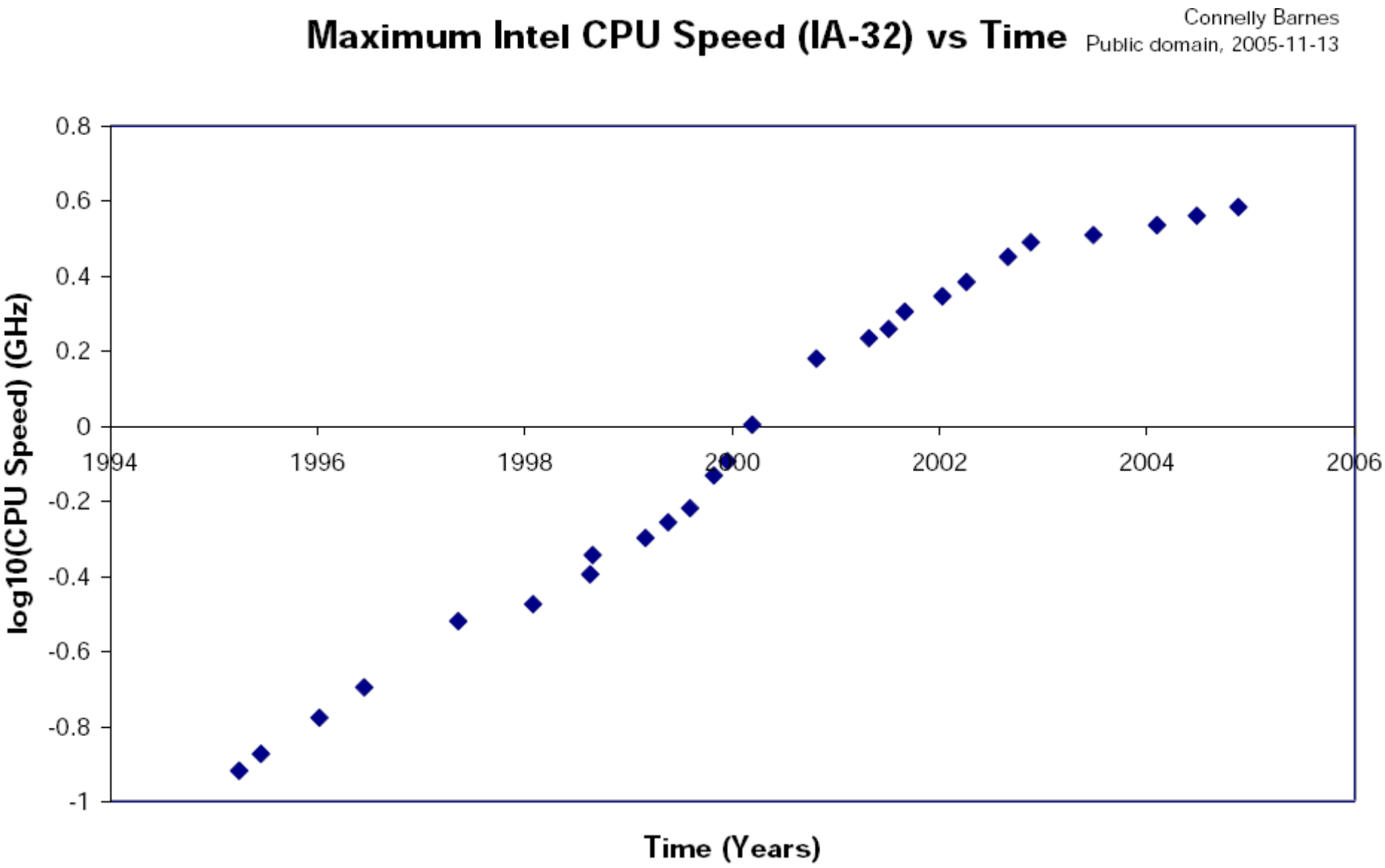
# Introduction





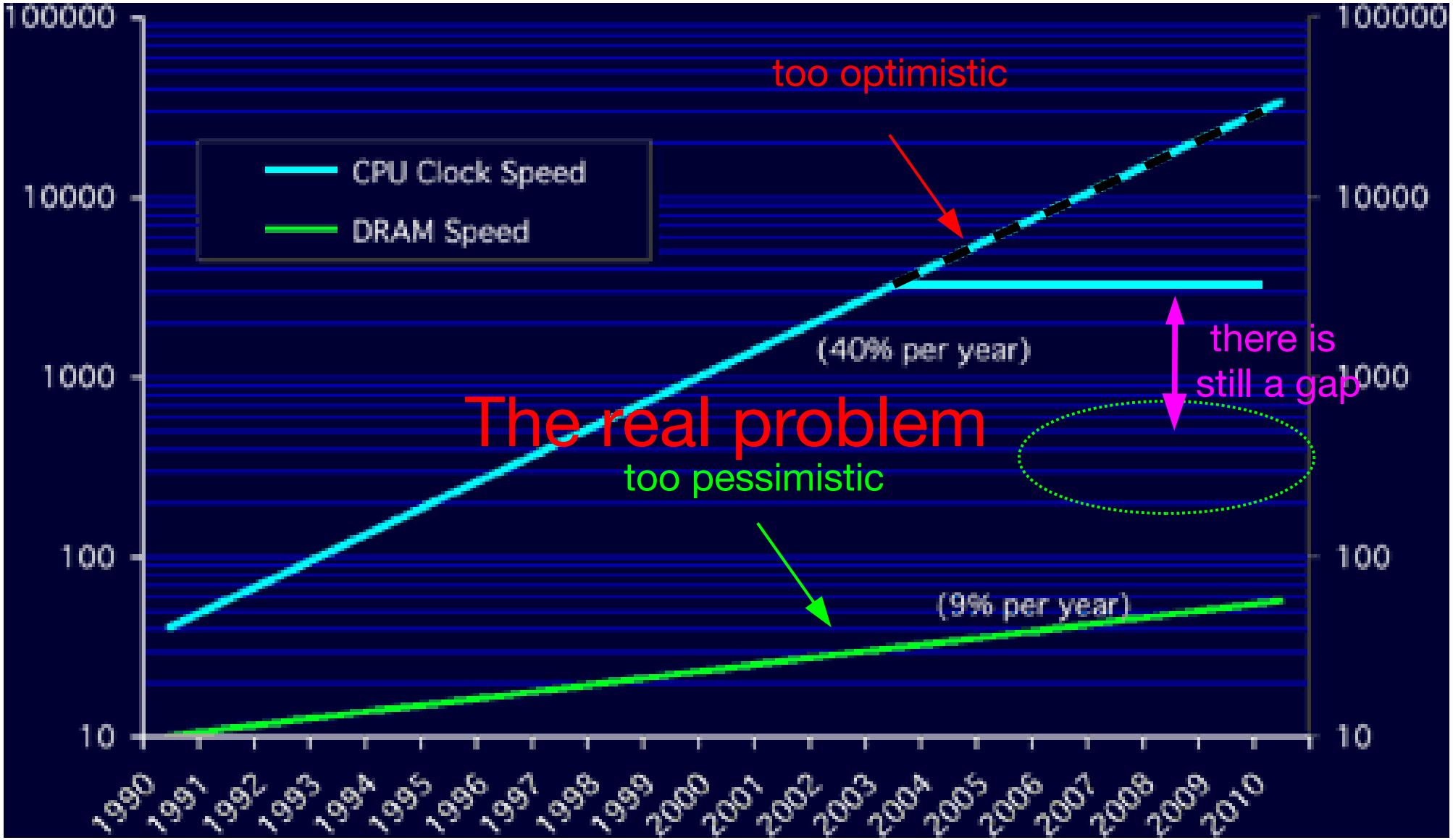
# Introduction





# Introduction





# Introduction

- CPU speed usually doubles every 18-24 months (not true any longer!).
- Development on the memory side is much slower (~ 6 years!).
- Memory speeds catch up – but also have to serve more cores!
- Something you should have in mind when designing your program!



# Introduction



- DDR2 specs (1.8V)

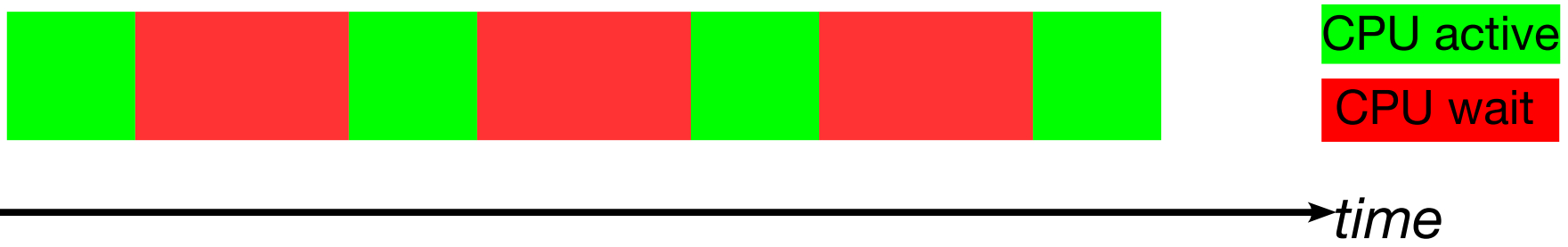
| Standard name | Memory clock | Cycle time | I/O Bus clock | Data transfers per second |
|---|---|---|---|---|
| DDR2-400 | 100 MHz | 10 ns | 200 MHz | 400 Million |
| DDR2-533 | 133 MHz | 7.5 ns | 266 MHz | 533 Million |
| DDR2-667 | 166 MHz | 6 ns | 333 MHz | 667 Million |
| DDR2-800 | 200 MHz | 5 ns | 400 MHz | 800 Million |
| DDR2-1066 | 266 MHz | 3.75 ns | 533 MHz | 1066 Million |

- DDR3 specs (1.5V)

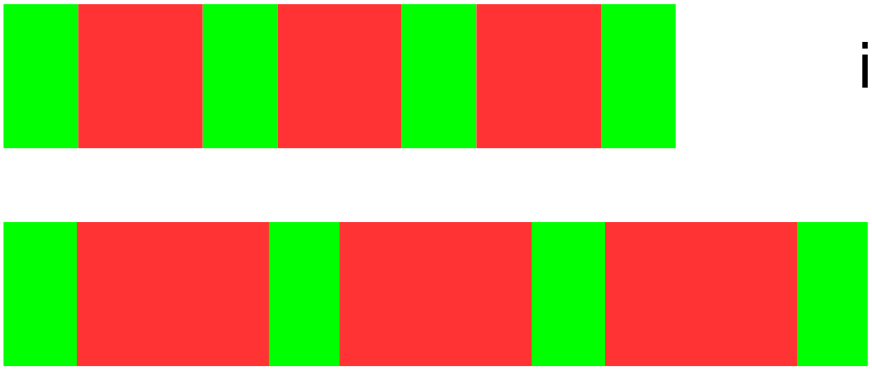
| Standard name | Memory clock | Cycle time | I/O Bus clock | Data transfers per second |
|---|---|---|---|---|
| DDR3-800 | 100 MHz | 10 ns | 400 MHz | 800 Million |
| DDR3-1066 | 133 MHz | 7.5 ns | 533 MHz | 1066 Million |
| DDR3-1333 | 166 MHz | 6 ns | 667 MHz | 1333 Million |
| DDR3-1600 | 200 MHz | 5 ns | 800 MHz | 1600 Million |

# Motivation for Application Tuning

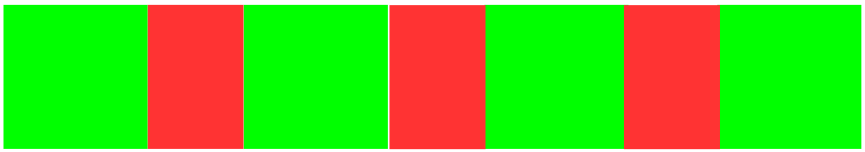
time flow in a computational task (simplified picture):




new hardware:




code tuning (old hardware):

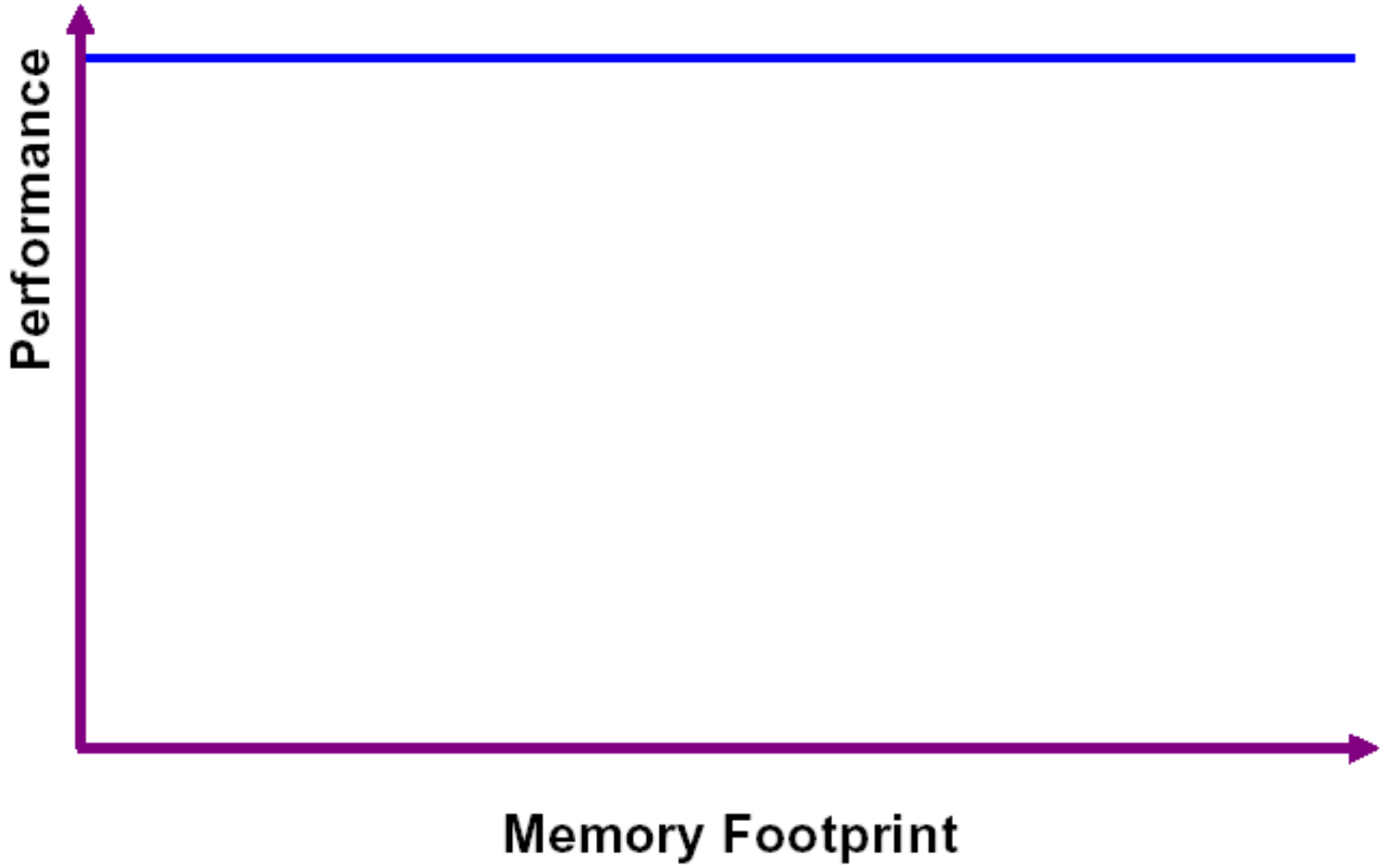


# The Memory Hierarchy

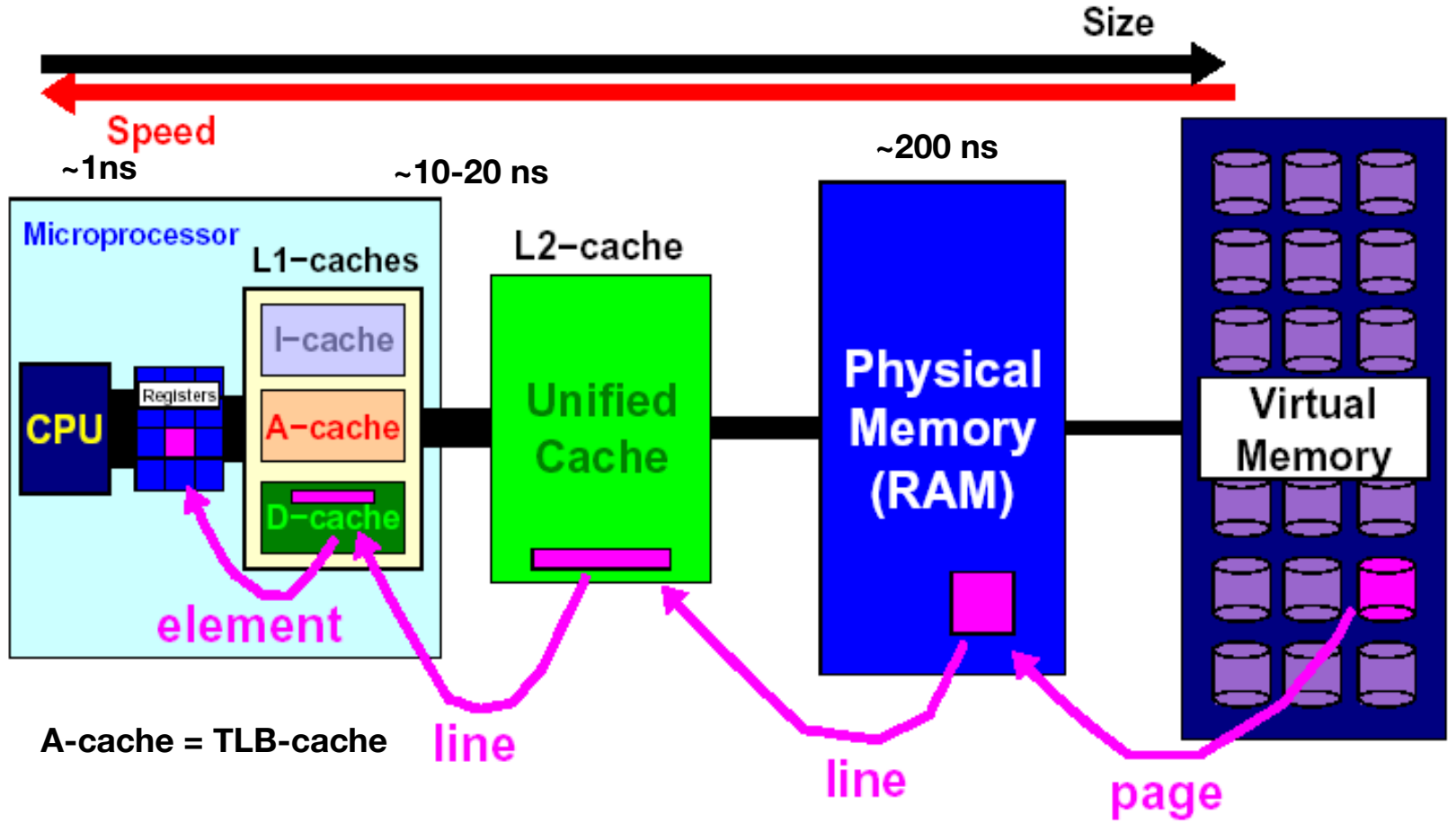
# The Memory Hierarchy



*Intuitive Performance Graph:*



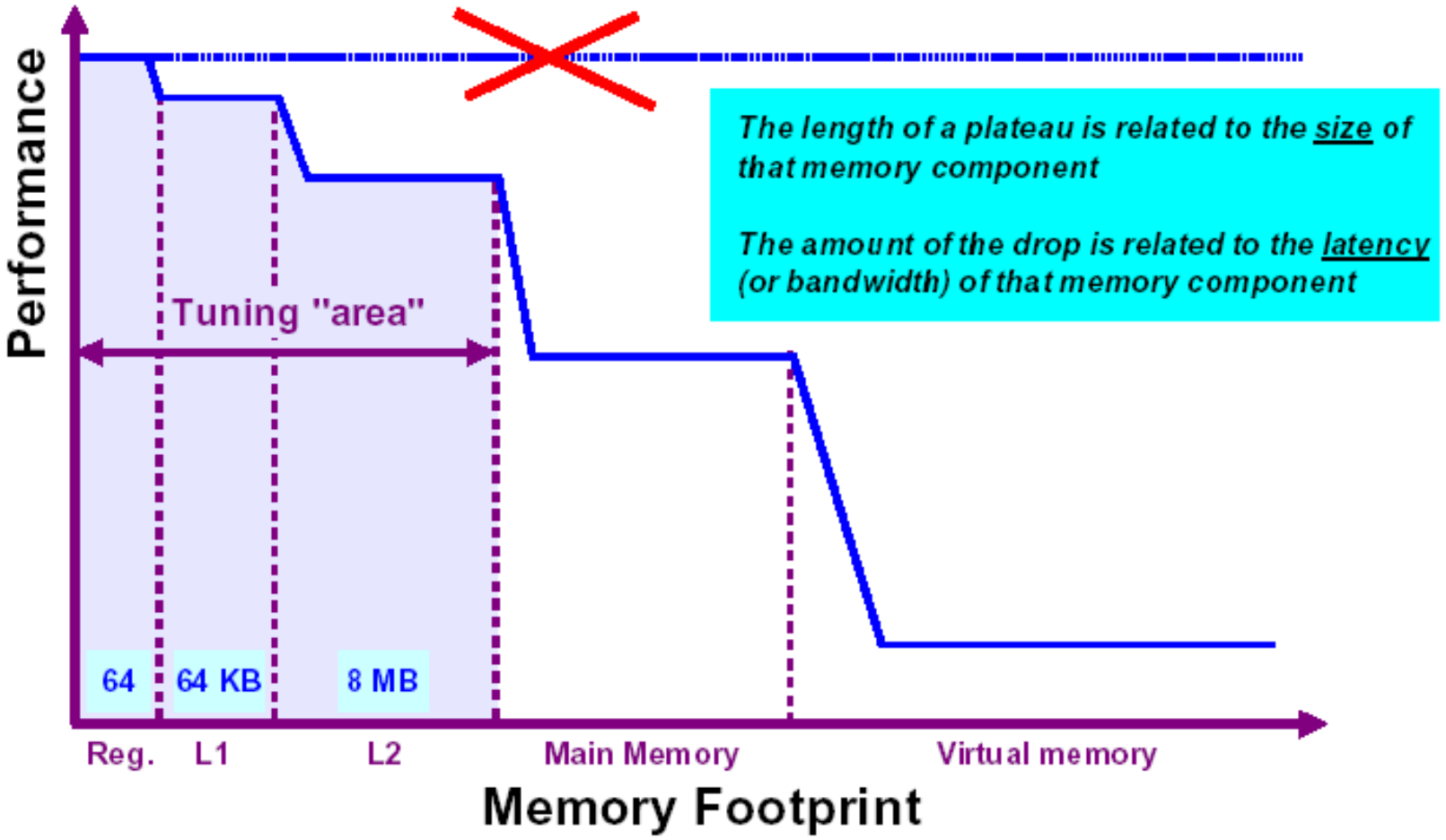
# The Memory Hierarchy




***Memory Optimization:***
***Keep frequently used data close to the processor***

# The Memory Hierarchy

*Performance is not uniform:*



# The Memory Hierarchy

- Memory plays a crucial role in performance
- Not accessing memory in the right way will degrade performance on **all** computer systems
- The extent of degradation will depend on the system
- Knowledge about the relevant memory characteristics helps to write code that minimizes those problems



# But what if ...

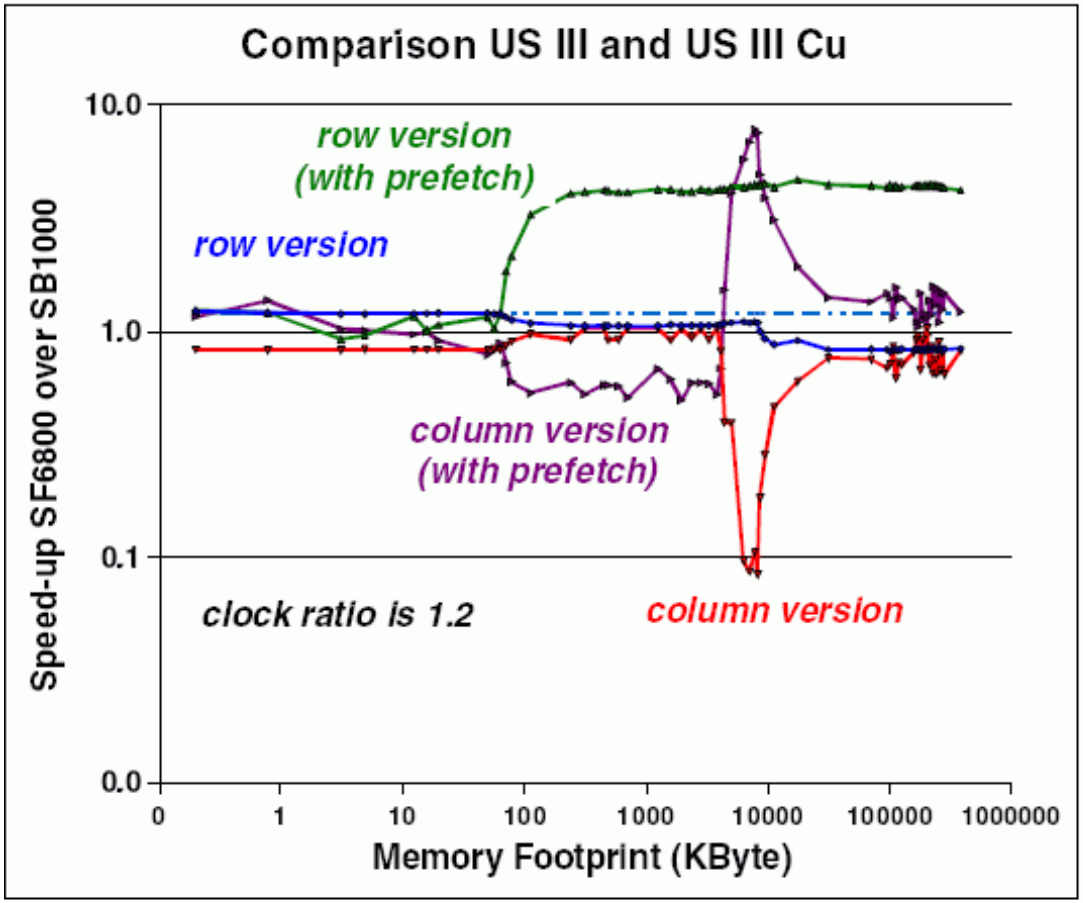
... I have an application that runs twice as fast on my double speed CPU?

Answer:

Your problem size probably fits into the L1 and L2 caches!

# What is performance?

- Matrix summation in two ways
- Compare two versions of the US-III chip:




- ***Compare the same version on the two different systems***
- ***Very often we do not see the clock ratio ....***
- ***It is either higher or lower***
- ***The column version takes advantage of the larger TLB capacity of the US III Cu processor***

# Caches – and all that ...



How do those caches work?

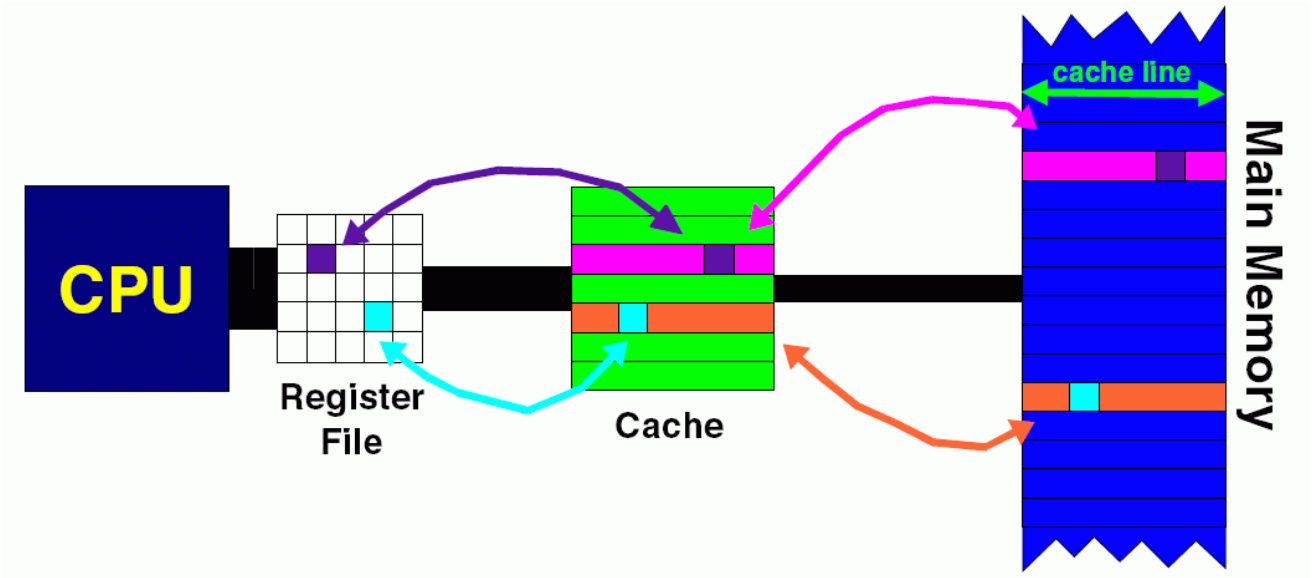
# Caches

- Cache memory or cache for short (from French: cacher – to hide): fast buffers that help to hide the memory latency
- One distinguishes between
  - data cache
  - instruction cache
  - address cache (also called TLB – Translation Lookaside Buffer) – mapping between virtual and physical addresses



# Cache Lines

- To get good performance, optimal use of the caches is crucial
- The unit of transfer is a "*cache line*":
  - linear structure of fixed length (bytes)
  - fixed starting address in memory



# Cache Organisation

## Direct Mapped:

- Each memory address maps onto exactly one line in cache
- simple and efficient
- built-in replacement policy
- easy to scale to larger sizes
- downside: no control by usage – danger of replacing data that will be needed again soon

# Cache Organisation



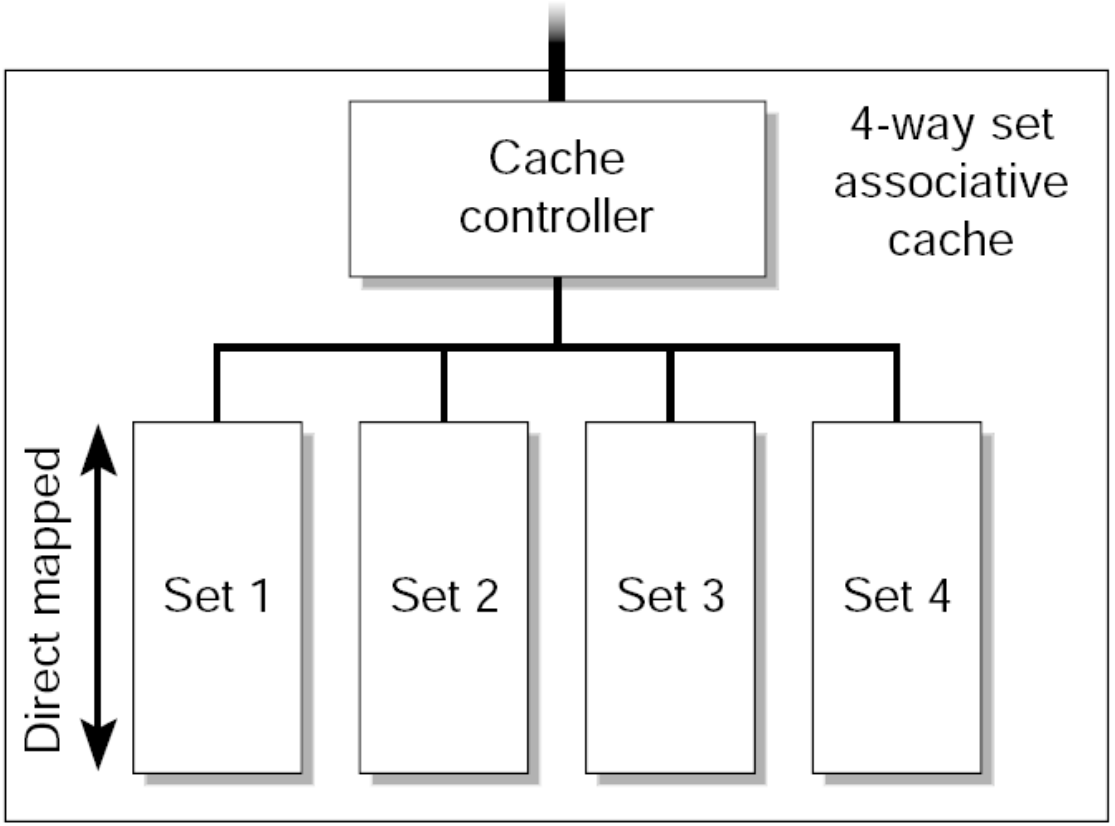
## Fully Associative:

- Every memory address can be mapped anywhere in cache
- Need to track usage of cache lines
- Requires a replacement policy, e.g. *least recent used* (LRU), *least frequent used* (LFU), random, etc
- Doesn't scale well to large sizes
- Costly design

# Cache Organisation

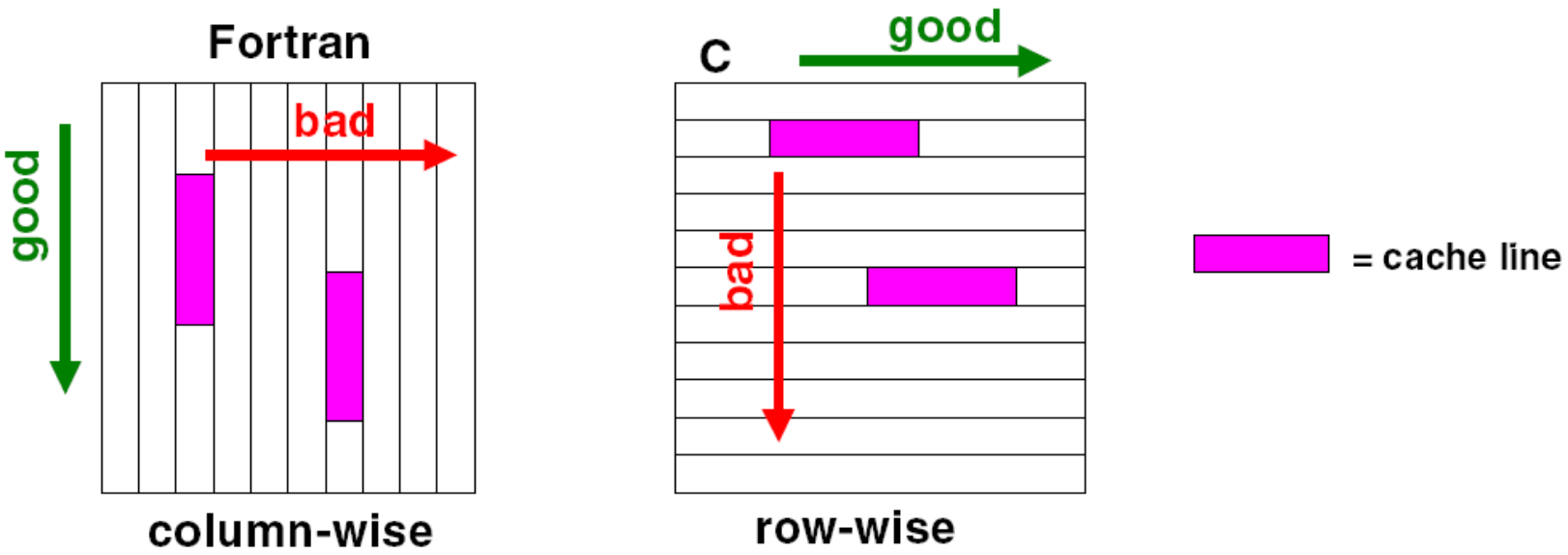
## N-way Set Associative:

- Sets of direct mapped caches:



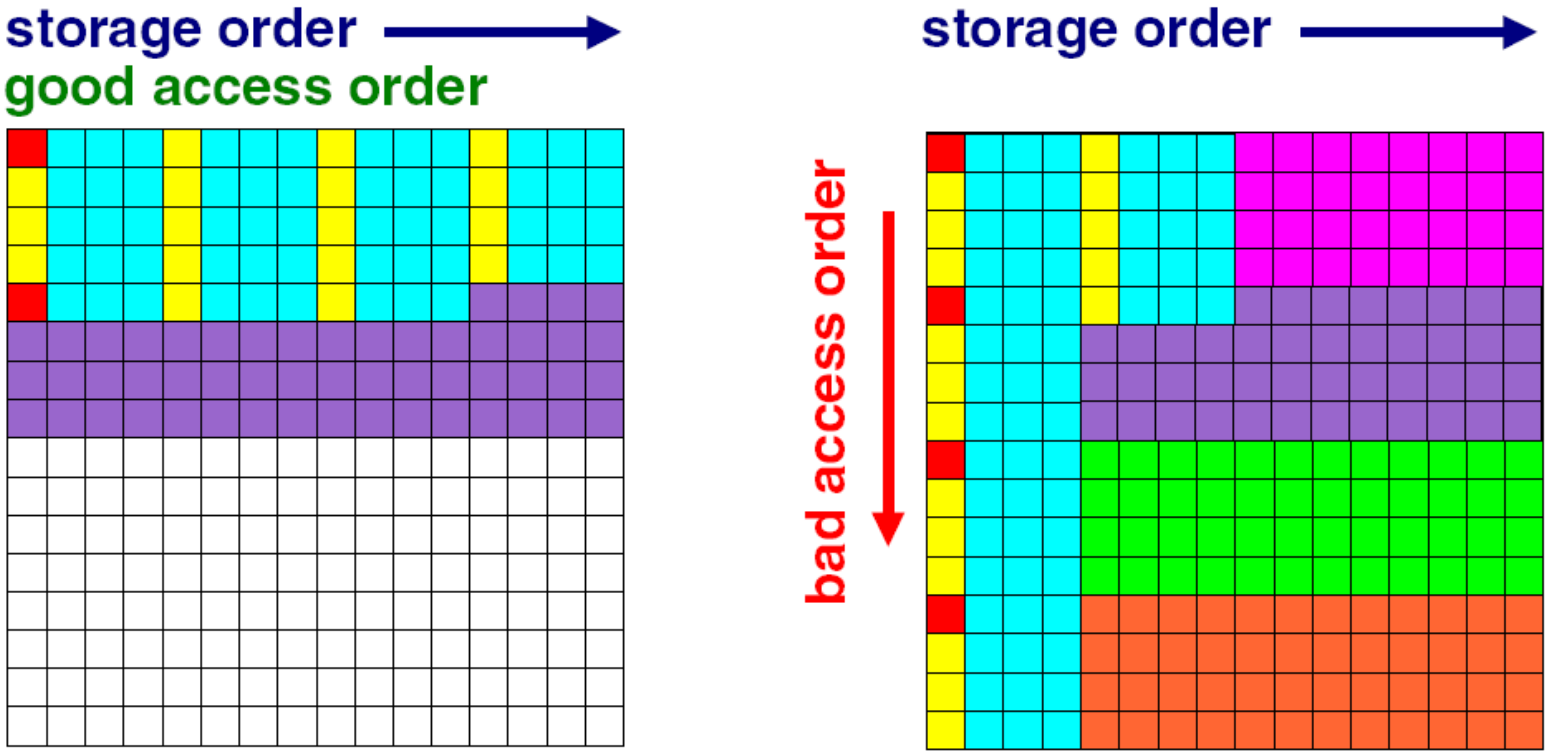
# Memory access

- Memory has a 1-dimensional linear structure
- Access to multi-dimensional arrays depends on how data is stored




Bad memory access has a huge impact on performance!!!

# Bad memory access – C example




= TLB miss
= D-cache miss
= Cached elements
= Virtual memory page

- If the entire matrix fits in the cache, the access pattern *hardly* matters.
- For large (out-of-cache) matrices, the access pattern **does** matter – both data cache and TLB misses

# About cache misses

Some simple rules:

- You cannot avoid cache misses – there are part of the nature of cache-based systems
- But you should try to minimize them to get good performance

# Cache Line Utilization

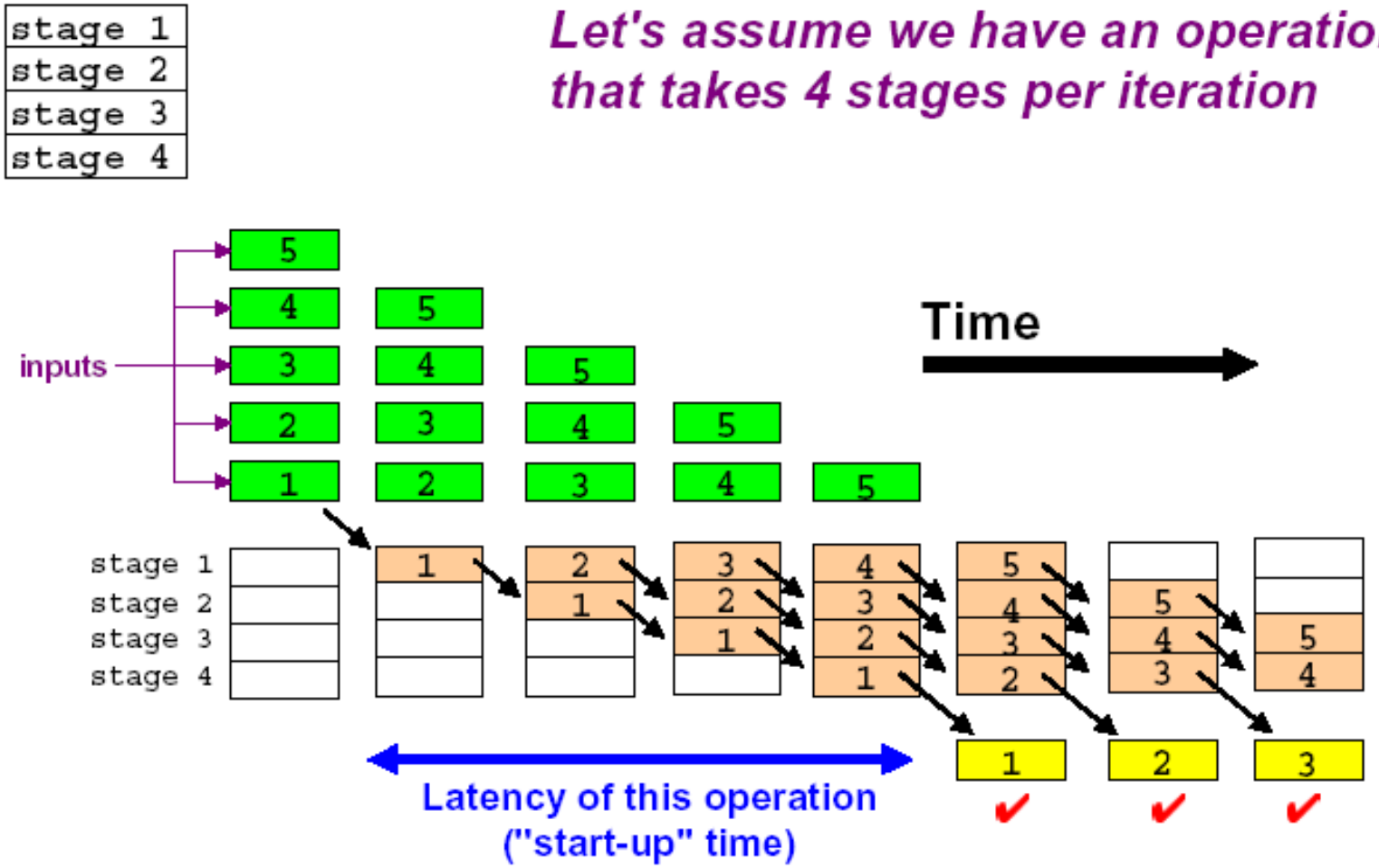


Two key rules: Maximize ...

- Spatial locality ⇒ Use all data in one cache line
  - depends on storage layout
  - depends on access patterns
    - stride = 1 is good
    - random access is really bad
- Temporal locality ⇒ Re-use data in a cache line
  - depends on algorithm used

# Some Terminology

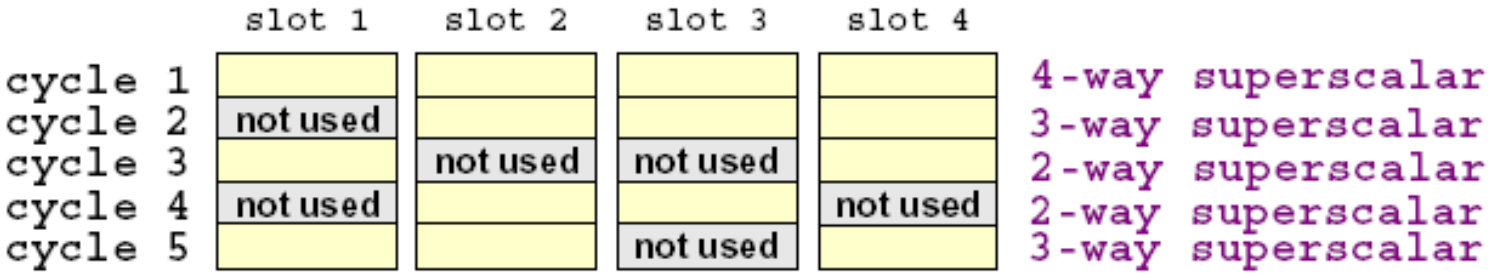
# Terminology: Pipelining




Rule of thumb: keep the pipeline filled for good performance!

# Terminology: Superscalar

- *N-way superscalar:*
  - *Execute N instructions at the same time*
- *This is also called Instruction Level Parallelism (ILP)*




- *The hardware has to support this, but it is up to the software to take advantage of it*
- *Often there are restrictions which instructions can be "bundled"*
- *These are documented in the Architecture Reference Manual for the microprocessor*



# Latency and Bandwidth

Latency:

- the time it takes from the initiation of an action till you have the first result
- unit: time

Bandwidth:

- how many
  - actions can be carried out,
  - results can be obtained,

  within a given time
- unit: #/time



# General Optimization Techniques

# Optimization Techniques - Overview

- Most optimization techniques are “loop based”
- Loop based optimizations:
  - Interchange
  - Fission and Fusion
  - Unrolling
  - Blocking
-



# Optimization Techniques - Overview

- Designing your data structures the “right way” can also be important
- Other techniques:
  - De-vectorization
  - Stripmining

# Loop based optimizations

# Coding style: array indexing

- To apply safe transformations, the compilers have to analyze data dependencies in a loop
- Explicit expressions will help the compilers to do a good job in loop optimization

**Good**

```
for(i=0; i<m; i++)
  for (j=0; j<n; j++)
    .. a[i][j] ..
```

**Reasonable**

```
for(i=0; i<m; i++)
  for (j=0; j<n; j++)
    .. a[i*n+j] ..
```

**Bad**

```
k = 0;
for(i=0; i<m; i++)
  for (j=0; j<n; j++)
    .. a[k++] ..
```

**Caution**

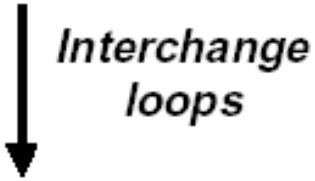
```
for(i=0; i<m; i++)
  for (j=0; j<n; j++)
    .. a[indx[i][j]] ..
```



# Loop Interchange



```
DO I = 1, M
   DO J = 1, N
      A(I,J)=B(I,J)+C(I,J)
   END DO
END DO
```




```
DO J = 1, N
   DO I = I, M
      A(I,J)=B(I,J)+C(I,J)
   END DO
END DO
```

- The matrices are accessed over the second dimension first
- This is the wrong order in Fortran
- A loop interchange solves the problem
- In C, the situation is reversed:
  - row access is okay
  - column access is bad

# Loop Fission



```
for (j=0; j<n; j++)
{
   c[j] = exp(j/n);
   for (i=0; i<m; i++)
      a[i][j]=b[i][j]+d[i]*e[j];
}
```

- *Access on arrays 'a' and 'b' is bad*
- *We can not simply interchange the loops*
- *Fission/splitting is the solution*

*Fission*

*This loop can now also be vectorized*

```
for (j=0; j<n; j++)
   c[j] = exp(j/n);
```

*New loop created*

*Interchange loops for better performance*

```
for (j=0; j<n; j++)
   for (i=0; i<m; i++)
      a[i][j]=b[i][j]+d[i]*e[j];
```

# Loop Fusion



```
for (i=0; i<n; i++)
    a[i] = 2 * b[i];
```

```
for (i=0; i<n; i++)
    c[i] = a[i] + d[i];
```

- *Assume that 'n' is large*
- *In the second loop, a[i] will no longer be in the cache*
- *Fusing the loops will ensure a[i] is still in the cache when needed*

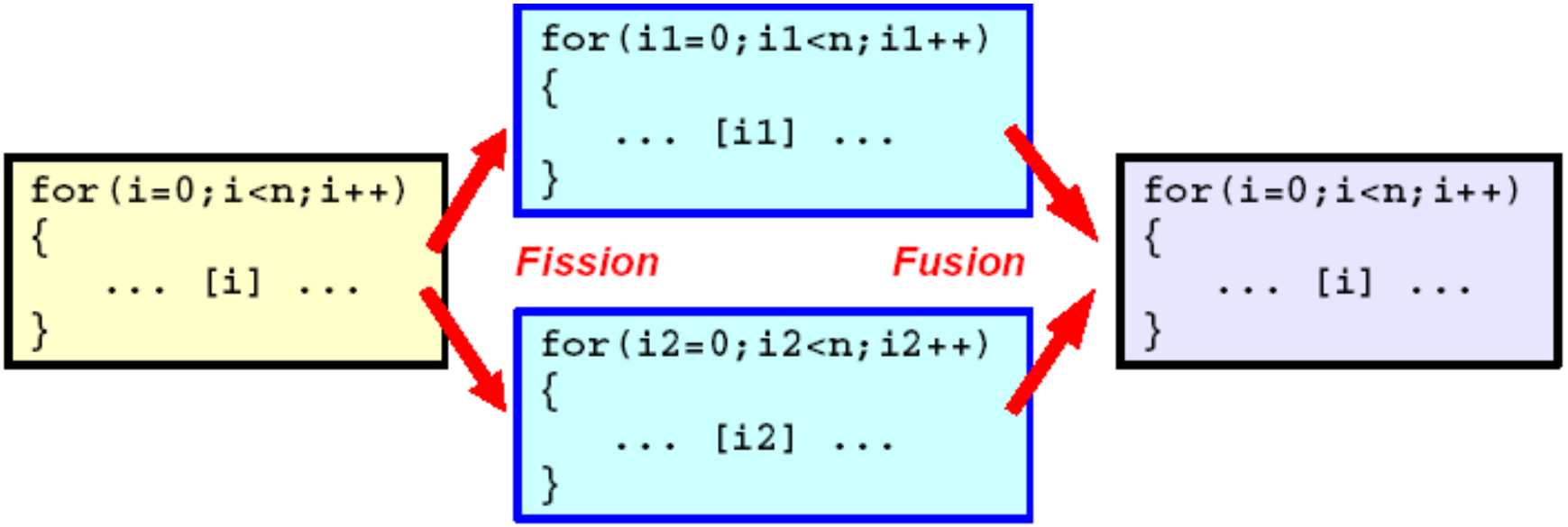
*Fusion*

*Note that it is possible to apply fusion to loops with (slightly) different boundaries*

*In such a case, some iterations will have to be 'peeled' off*

```
for (i=0; i<n; i++)
{
    a[i] = 2 * b[i];
    c[i] = a[i] + d[i];
}
```

# Fission and Fusion – Summary




## *Fission*

- ✔ *Reduce register pressure*
- ✔ *Enable loop interchange*
- ✔ *Isolate dependencies*
- ✔ *Increase opportunities for optimization (e.g. vectorization of intrinsics)*

## *Fusion*

- ✔ *Reduce cache reloads*
- ✔ *Increase Instruction Level Parallelism (ILP)*
- ✔ *Reduce loop overhead*



# Inner Loop Unrolling



*Through unrolling, the loop overhead ('book keeping') is reduced*

```
for (i=0; i<n; i++)
   a[i] = b[i] + c[i];
```

*Loop is unrolled with a factor of 4*

```
for (i=0; i<n; i+=4)
{
   a[i  ] = b[i  ] + c[i  ];
   a[i+1] = b[i+1] + c[i+1];
   a[i+2] = b[i+2] + c[i+2];
   a[i+3] = b[i+3] + c[i+3];
}
<clean-up loop>
```

```
Loads      : 2
Stores     : 1
FP Adds    : 1
I=I+1
Test I < N ?
Branch
Addr. incr: 3
```

*Work: 4*
*Overhead: 6*

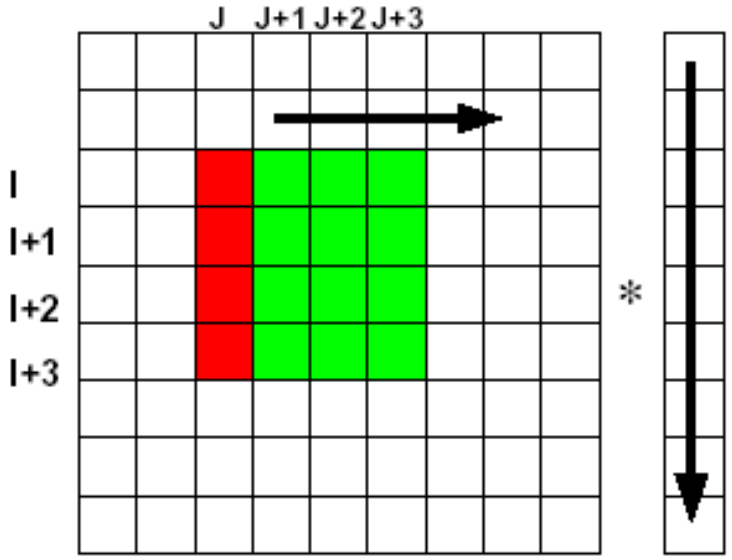
```
Loads      : 8
Stores     : 4
FP Adds    : 4
I=I+4
Test I < N ?
Branch
Addr. incr: 3
```

*Work: 16*
*Overhead: 6*

*Note: the amount of addressing needed in reality is less*

# Outer Loop Unrolling 1






```
for (i=0; i<m; i++)
  for(j=0; j<n; j++)
  {
    a[i] += b[i][j] * c[j];
  }
```

```
for (i=0; i<m; i+=4)
  for(j=0; j<n; j++)
  {
    a[i  ] += b[i  ][j] * c[j];
    a[i+1] += b[i+1][j] * c[j];
    a[i+2] += b[i+2][j] * c[j];
    a[i+3] += b[i+3][j] * c[j];
  }
<clean-up loop>
```

- ***Advantage:***
  - ***c[j] is re-used 3 more times (temporal locality)***
- ***Deeper unrolling, say 8, requires more fp registers (17 instead of 9), but improves re-use of c[j]***

# Outer Loop Unrolling 2



```
for (i=0; i<m; i++)
  for(j=0; j<n; j++)
    a[i] += b[i][j] * c[j];
```

*Outer loop unrolling*

```
for (i=0; i<m-m%4; i+=4)
{
  for(j=0; j<n; j++)
    a[i  ] += b[i  ][j] * c[j];
  for(j=0; j<n; j++)
    a[i+1] += b[i+1][j] * c[j];
  for(j=0; j<n; j++)
    a[i+2] += b[i+2][j] * c[j];
  for(j=0; j<n; j++)
    a[i+3] += b[i+3][j] * c[j];
}
for (i=m-m%4; i<m; i++)
  for(j=0; j<n; j++)
    a[i] += b[i][j] * c[j];
```

*clean-up loop*

*Unroll and Jam*

```
for (i=0; i<m-m%4; i+=4)
  for(j=0; j<n; j++)
  {
    a[i  ] += b[i  ][j] * c[j];
    a[i+1] += b[i+1][j] * c[j];
    a[i+2] += b[i+2][j] * c[j];
    a[i+3] += b[i+3][j] * c[j];
  }
for (i=m-m%4; i<m; i++)
  for(j=0; j<n; j++)
    a[i] += b[i][j] * c[j];
```

*clean-up loop*

*Jam the loops together again*

# Loop unrolling – structure



```
for (i=0; i<n; i++)
{
    ... [i] ...
}
```

```
DO I = 1, N
    ... (I) ...
END DO
```

*Loop unroll factor is "unroll"*

```
for(i=0;i<n-n%unroll;i+=unroll)
{
    ...[i]   ...
    ...[i+1]...
    ...[i+2]...


    ...[i+unroll-1]...
}
```

*Unrolled Loop*

```
DO I = 1, N-mod(N,unroll), unroll
    ...(I)   ...
    ...(I+1)...
    ...(I+2)...

    ...(I+unroll-1)...
END DO
```

*Cleanup Loop*

```
for(i=n-n%unroll;i<n;i++)
{
    ...[i]...
}
```

```
DO I = N-mod(N,unroll)+1,N
    ... (I) ...
END DO
```

# Loop Unrolling – Summary

- More than one iteration per loop pass
- Inner loop unrolling:
  - reduce loop overhead
  - better instruction scheduling
- Outer loop unrolling:
  - improve cache line usage (spatial locality)
  - re-use data (temporal locality)
- Disadvantages:
  - more registers needed
  - clean-up code required



# Loop Unrolling – Compilers

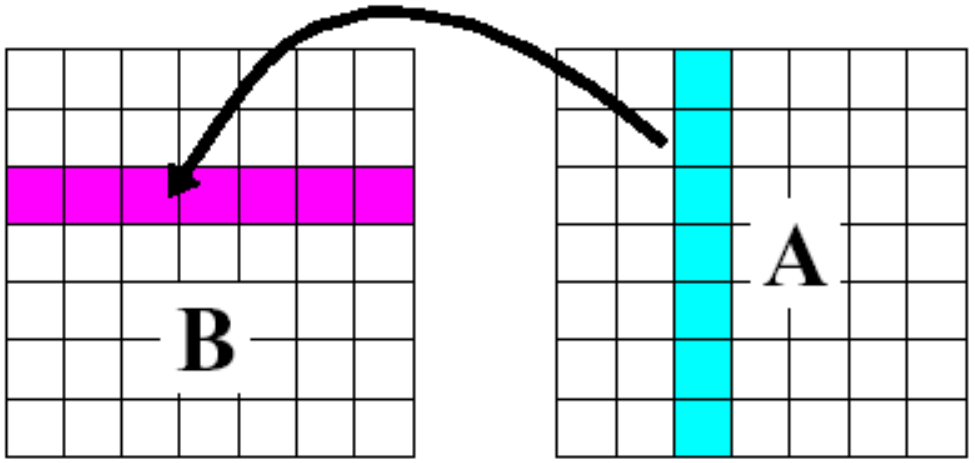


- Compilers do usually a good job in loop unrolling
- there are options to control the unroll depth
  - Sun: -xunroll=n (1: no unroll, 2..n: unroll n times)
  - gcc: -funroll-loops --params max-unroll-times=n
  - Intel: -unroll[n] (0: disable loop unrolling)

# Loop Blocking – 1

## *Transposing a matrix*

```
for (j=0; j<n; j++)
   for (i=0; i<n; i++)
      b[j][i] = a[i][j];
```




- *Loop interchange will not help here:*
  - *Role of 'a' and 'b' will only be interchanged*
- *Change of programming language won't help either*
- *Unrolling the i-loop can be beneficial, but requires more registers and doesn't address TLB-misses*
- *Loop blocking achieves good memory performance, without the need for additional registers*

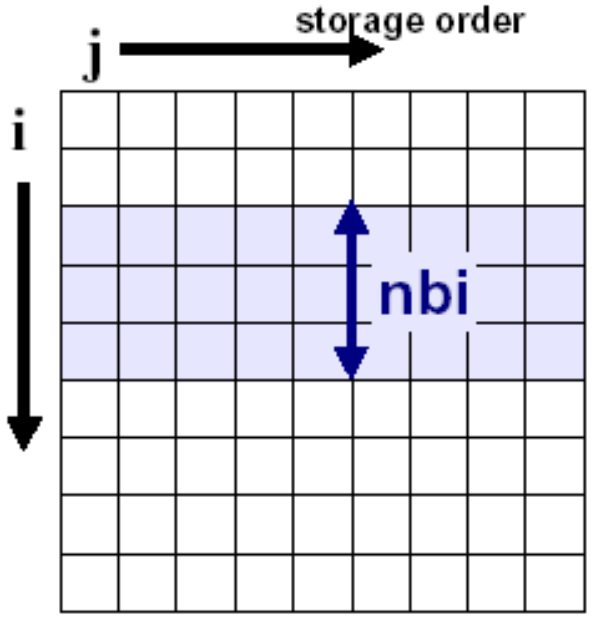


# Loop Blocking – 2

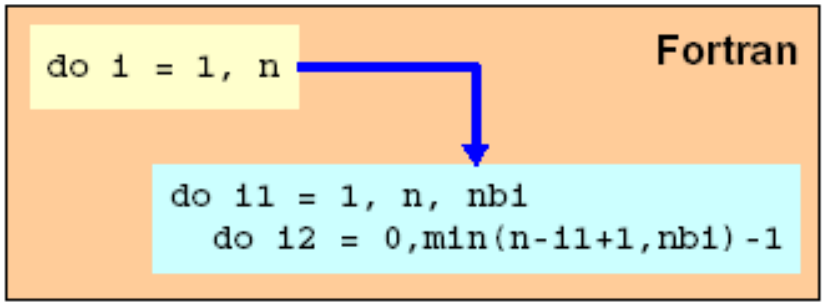


***Blocking and interchanging the I-loop***

```
for(i1=0; i1<n; i1+=nbi)
  for (j=0; j<n; j++)
    for (i2=0;i2<MIN(n-i1,nbi);i2++)
      b[j][i1+i2] = a[i1+i2][j];
```




- *Parameter 'nbi' is the blocking size*
- *Should be chosen as large as possible*
- *Actual value depends on the cache to block for:*
  - *L1-cache*
  - *L2-cache*
  - *TLB*
  - *....*



# Loop Blocking – Summary

- Powerful technique to improve:
  - memory access (spatial locality)
  - data re-use (temporal locality)
- Preserves portability – but blocking size depends on:
  - cache type/level/capacity
  - data requirements



# Loop Blocking – Summary

Recommendations:

- choose blocking size as large as possible
- leave space for other data
- parameterize cache characteristics, especially size

# Optimization Benefits

| Optimization | Instruction | Memory | Sun Compiler |
|---|---|---|---|
| Loop Interchange | + | ++ | yes |
| Loop Fission | + | ++ | yes |
| Loop Fusion | + | ++ | yes |
| Inner Loop Unrolling | ++ | - | yes |
| Outer Loop Unrolling | + | ++ | yes |
| Loop Blocking | - | ++ | yes |

Keep the last column in mind when playing around with your code – the compiler might already have done what you intended to do!!!



# Tricked by the compiler

Fortran code example: long and bulky loop

```
DO I1=1,NAT(IMT)
  IA1=IM1+(I1-1)*3
  DO I2=1,NAT(JMT)
    IA2=IM2+(I2-1)*3
        ... statements removed ...
    DX(1)=XNOW(IMT,IA1+1)-XNOW(JMT,IA2+1)
    DX(2)=XNOW(IMT,IA1+2)-XNOW(JMT,IA2+2)
    DX(3)=XNOW(IMT,IA1+3)-XNOW(JMT,IA2+3)
    CX(1)=CM1(1)-CM2(1)
    CX(2)=CM1(2)-CM2(2)
    CX(3)=CM1(3)-CM2(3)
        ... statements removed ...
  ENDDO
ENDDO
```

Independent of loop indices!!!

Moved the 3 lines above the DO loops – no improvement!
Compiler had done this already!



# Only the programmer knows ...



```
subroutine do_calc(...)

real(8),dimension(N,M,O,P)

!---- data initialization
r = 0.0d0; s = 0.0d0; t =

select case(calc_type)
   case(most_of_the_time)
      ...
      r(i,j,k,l) = r(i,j,k,
      s(i,j,k,l) = s(i,j,k,
      ...

   case(rare_event)
      ...
      r(i,j,k,l) = r(i,j,k,
      s(i,j,k,l) = s(i,j,k,
      t(i,j,k,l) = t(i,j,k,
      ...
end select
```

```
subroutine do_calc(...)

real(8),dimension(N,M,O,P):: r,s,t

!---- data initialization
r = 0.0d0; s = 0.0d0

select case(calc_type)
   case(most_of_the_time)
      ...
      r(i,j,k,l) = r(i,j,k,l) + ...
      s(i,j,k,l) = s(i,j,k,l) + ...
      ...

   case(rare_event)
      t = 0.0d0
      ...
      r(i,j,k,l) = r(i,j,k,l) + ...
      s(i,j,k,l) = s(i,j,k,l) + ...
      t(i,j,k,l) = t(i,j,k,l) + ...
      ...
end select
```

# Data structure design



Access your data in the right way

# Data structure design

- "Good advice" from a HPC tutorial: Use data structures to avoid too many index calculations
- Example: particle simulation in 3D with x, y, z coordinates and some other information about each particle, e.g. distance to i+1
  - `x[i], y[i], z[i], distance[i], someinfo[][i]`
  - turn this into a data structure ...



# Data structure design

- Particle data structure

```
typedef struct particle {
    double x, y, z;
    char someinfo[NBYTES];
    double distance;
} particle_t;
```

- Is this a good idea?



# Data structure design

- Answer: It depends ...
  - ... on problem size
  - ... how you access the data
  - ... cache, CPU, etc.
- Example: program with 2 functions/routines
  - calc() - accesses all parts of particle_t
  - re-use() - accesses particle.distance only
  - usage ratio of both functions is 1:1



# Data structure design

Comparison – two versions:

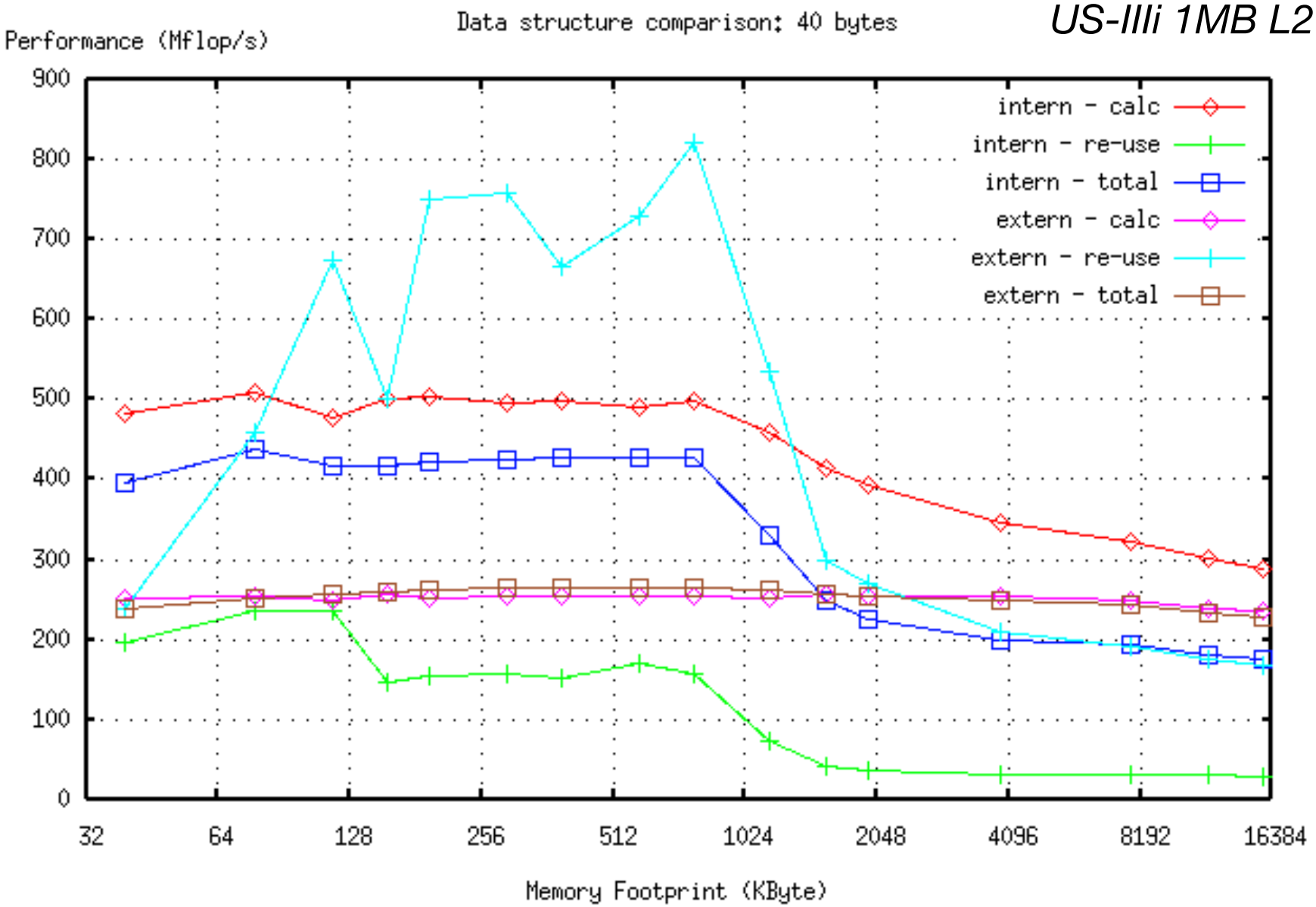
- *intern* – use data structure from above
- *extern* – distance peeled off the data structure as an external vector
- different problem sizes:
  - number of particles
  - size of data structure
- 2 different L2 cache sizes



# Data structure design



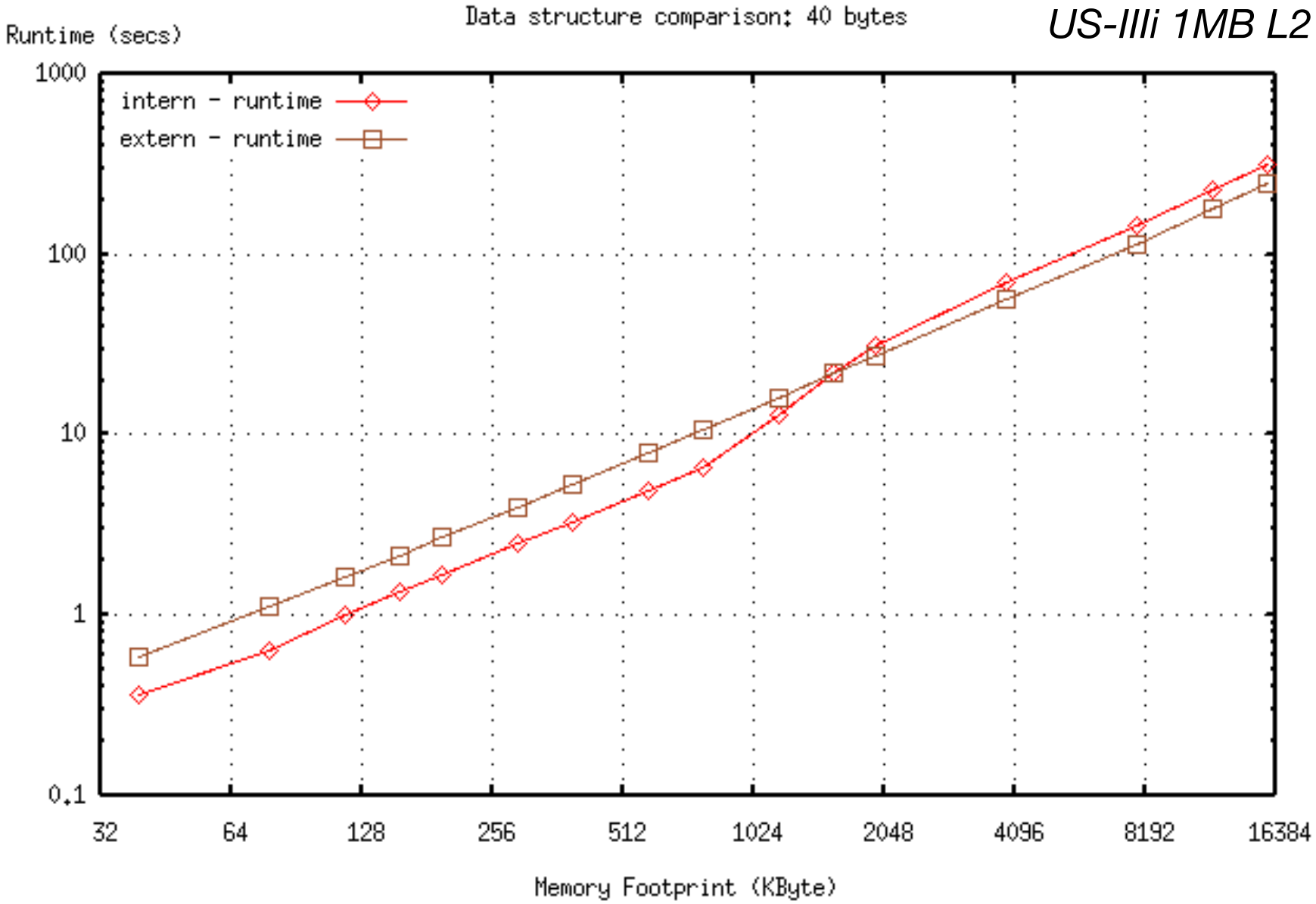
*US-IIIi 1MB L2 cache*



# Data structure design



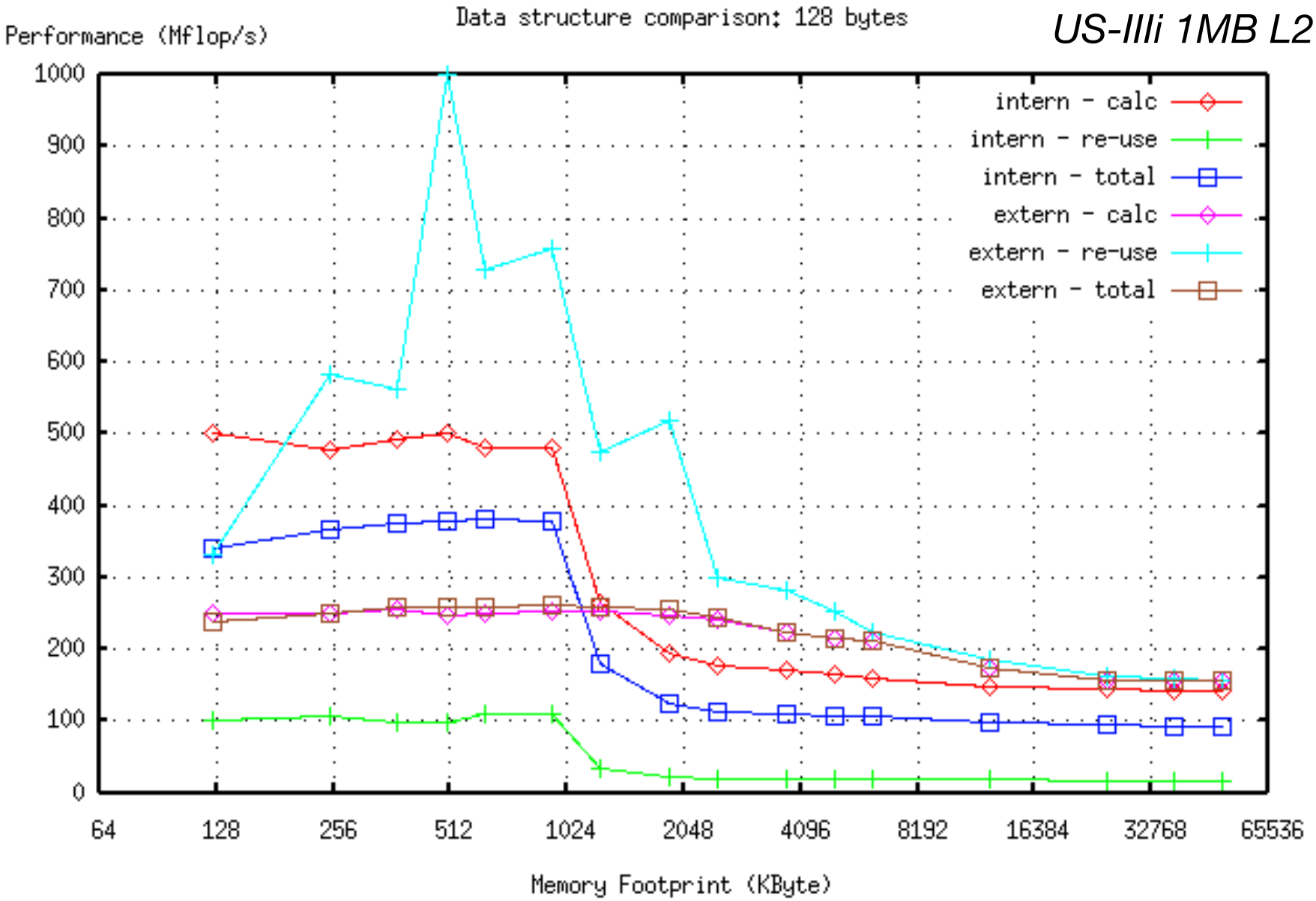
*US-IIIi 1MB L2 cache*



# Data structure design

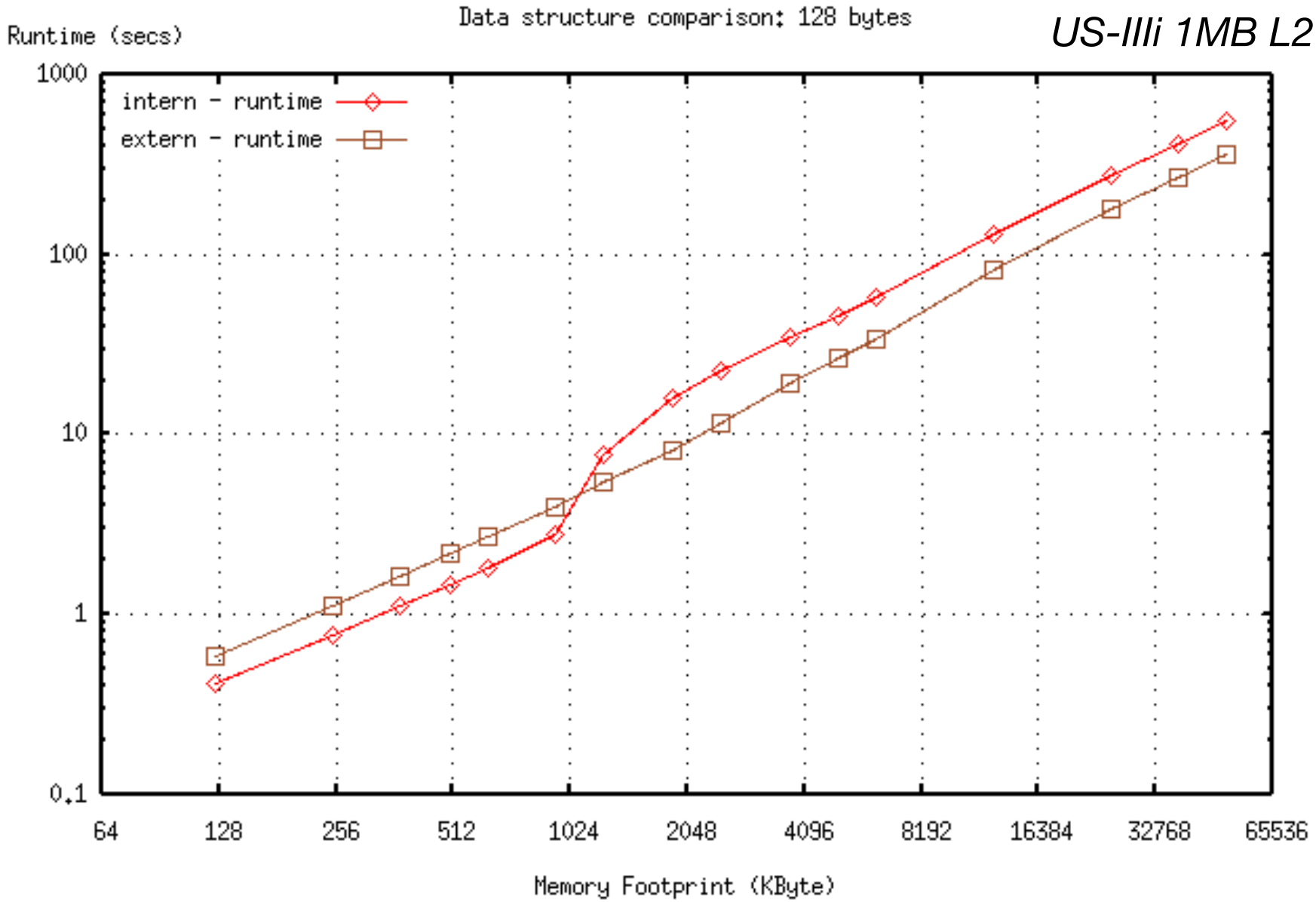


*US-IIIi 1MB L2 cache*



# Data structure design



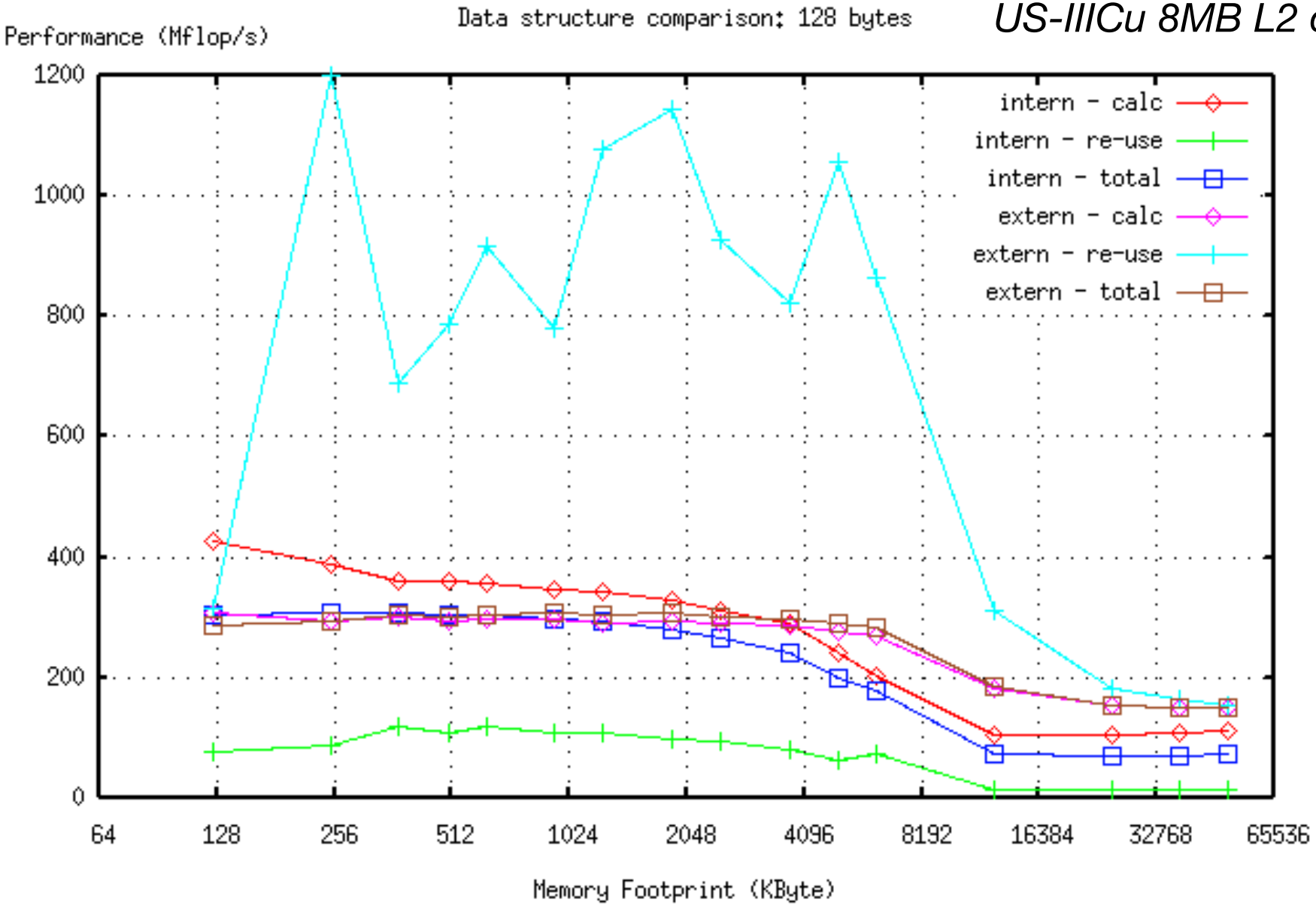



*US-IIIi 1MB L2 cache*

# Data structure design



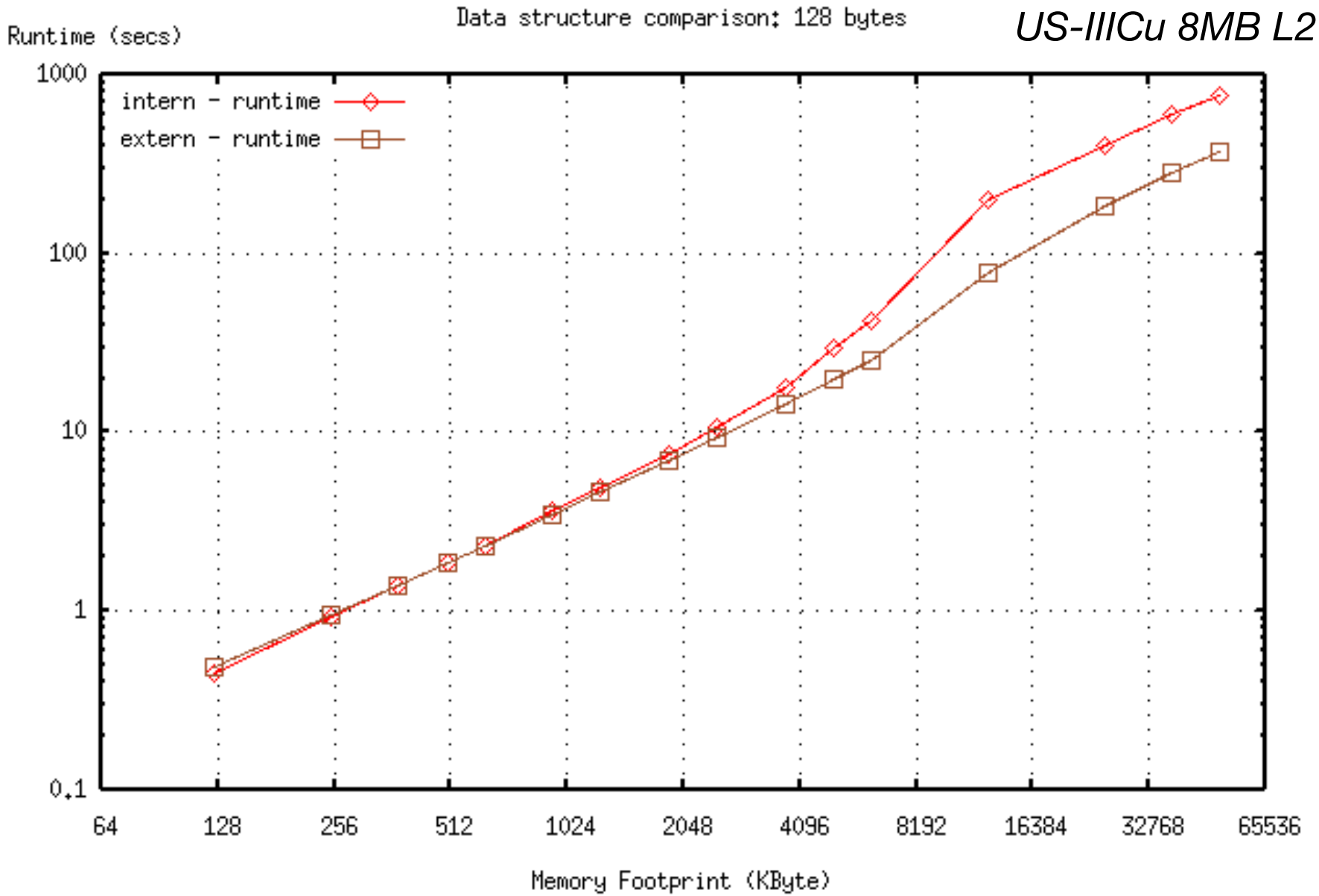
*US-IIICu 8MB L2 cache*



# Data structure design



*US-IIICu 8MB L2 cache*



End of lecture 1