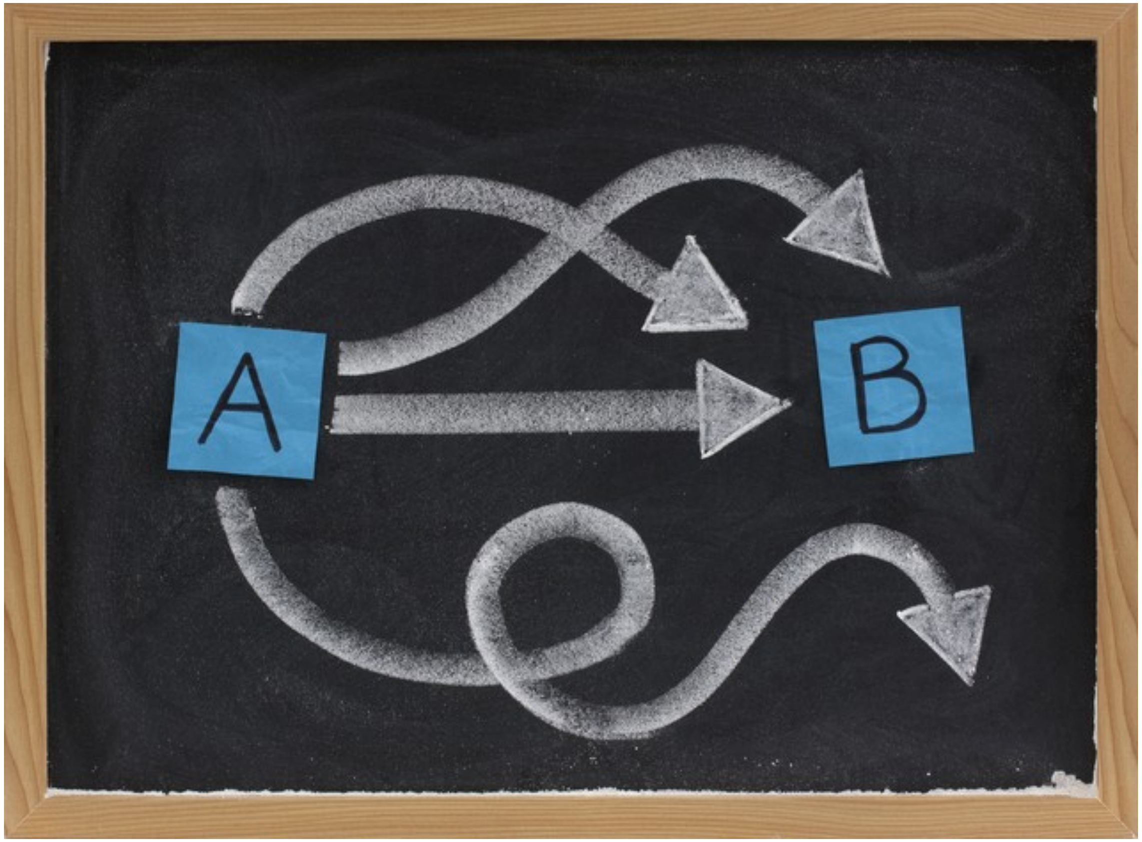
Introduction to Serial Code Optimisations



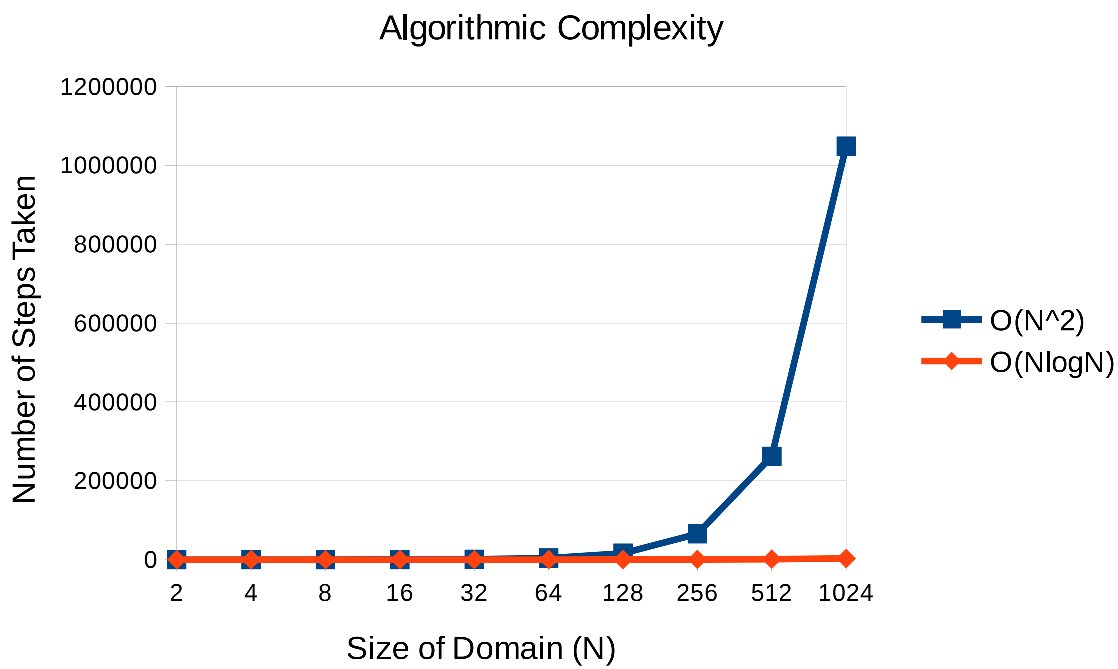
Getting from A to B – faster!

University of BRISTOL

Overview

- Some hints and tips:
  - Lesson #1: Algorithms matter.
  - Lesson #2: Focus your efforts.
    - Never guess, use a profiler instead.
  - Lesson #3: Experiment with compiler flags ***and then re-profile***.
  - Lesson #4: The memory hierarchy has a large effect (and it grows over time).
  - Lesson #5: Making use of wide registers is important on modern processors.

Lesson #1: Algorithms Matter



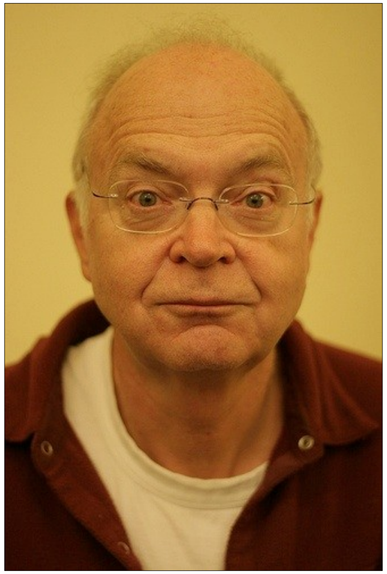
A solution in $O(N^2)$..
What do you mean?
Let's say we have a problem 'of size N'. This could be sorting a list of N items.
Sorting algorithm A takes $O(N^2)$ operations to sort the list. Algorithhm B takes O(NlogN) operations..

- Algorithm A → $O(N^2)$.
- Algorithm B → O(NlogN).
- For large N, even a lousy implementation of B will beat a great implementation of A.

University of BRISTOL

Lesson #2: Focus your Efforts

"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%.** A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but **only after that code has been identified**" — Donald Knuth



For the assignments, we're going to
(a) assume that we have the best algorithm; and
(b) follow Knuth's sage advice:

1 - profile
2 - find the critical code
3 - only attempt to optimise the critical code
4 - **repeat this cycle!**

University of BRISTOL

Lesson #2: Never Guess

<u>Consider the following 2 loops:</u>

```
/* loop 1 */
for (i=0; i<N; i++)
  a[i] = pow(6,0.35);

/* loop 2 */
x = pow(6,0.35);
for (i=0; i<N; i++)
  b[i] = x;
```

This is repeated (and hence wasted) computation, right?

gcc -O3

```
Loop containing invariants
Elapsed time:          0.752673 (s)
Manually 'hoisted' loop
Elapsed time:          0.938888 (s)
```


University of BRISTOL

Lesson #2: Use a Profiler Instead

This is some output from ***gprof***:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
72.69	25.01	25.01	300000	83.37	83.37	collision
15.03	30.18	5.17	300000	17.23	17.23	propagate
9.13	33.32	3.14	300000	10.47	10.47	total_density

This function accounts for the vast majority of the run time.
So let's focus our (initial) efforts there..

Changing the Compiler Flags (Conditionals in a Loop)

```
const size_t arraySize = 32768;
std::vector<int> data(arraySize);

// random sequence in range [0,256]
for (unsigned c = 0; c < arraySize; ++c)
  data[c] = std::rand() % 256;

// branches predictable if sorted, otherwise random
std::sort(data.begin(), data.end());

long long sum = 0;
for (unsigned i = 0; i < 100000; ++i)
  { for (unsigned c = 0; c < arraySize; ++c)
      { if (data[c] >= 128)
           sum += data[c];
      }
  }
```

Fill array with random numbers

Sort, or not..

Conditional triggers Predictably or randomly

University of BRISTOL

Branch Prediction: Processors and Compiler Flags

gcc 4.4.3

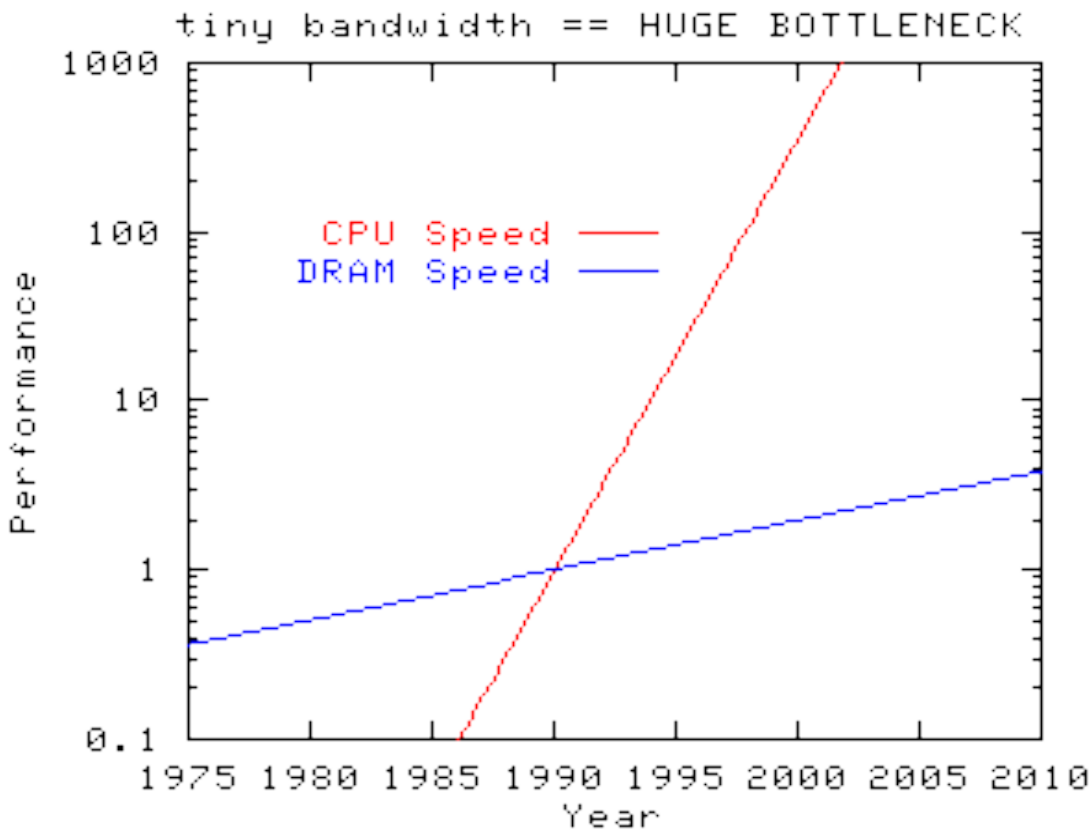
gcc 4.4.6

			Time (s)
Core 2	-O2	sorted	3.2
		random	13.7
	-O3	sorted	6.4
		random	15.6
Xeon E5	-O2	sorted	2.0
		random	10.4
	-O3	sorted	2.6
		random	2.6

- Branching in loops can stall the (deep) pipelines in modern processors. (Yet another attempt to find parallelism in code execution)
- Branch prediction clearly matters.
- **BUT**.. the details vary from processor to processor and compiler to compiler.

University of BRISTOL

Lesson #4: The Memory Hierarchy has a Large Effect

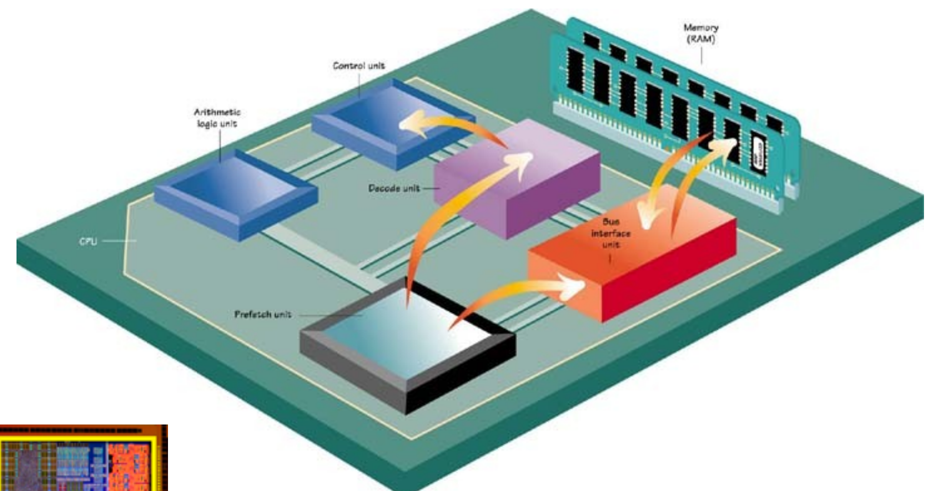


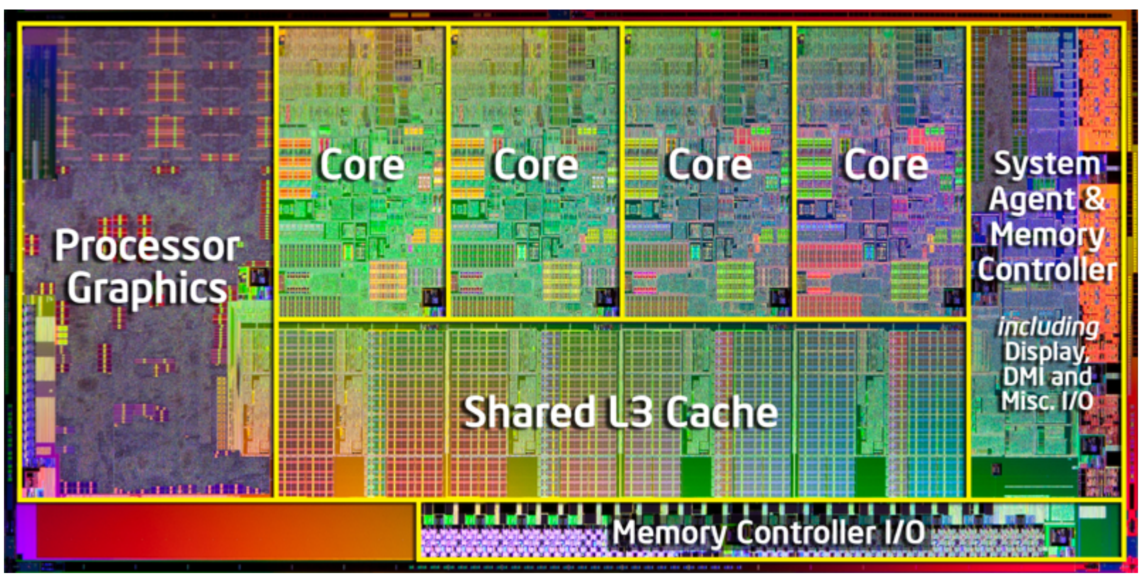
See e.g. the STREAM banchmark:
https://www.cs.virginia.edu/stream/

University of BRISTOL

Caches try to Bridge the Gap

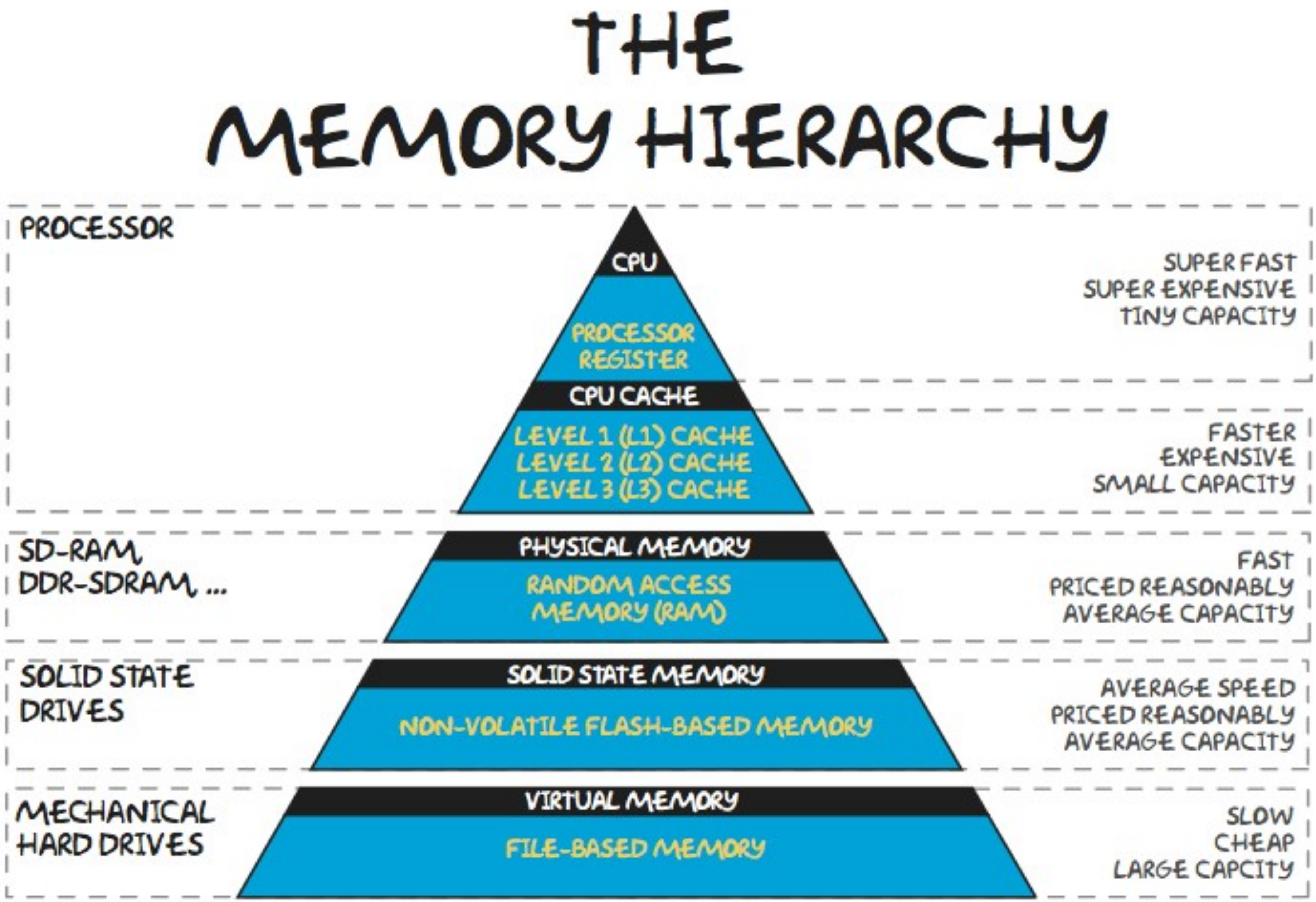
<u>General Layout:</u>
- Registers and Caches on chip
- Bus connection to RAM
- Hard drives slower access times





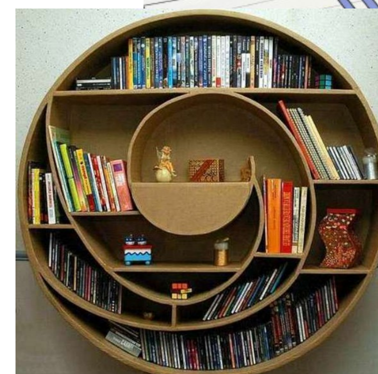
Schematic for Intel SandyBridge

University of BRISTOL



University of BRISTOL

Memory Hierarchies: Analogy


- L1 cache: Like picking up a paper from your desk (~3s)
- L2 cache: Like getting up and going to the bookshelf (~15s)
- Main memory: Like walking down the corridor (several minutes)
- Disk?: Like walking around the coastline of Britain (~ 1yr)!

University of BRISTOL

Make the Most of the Cache that You Have

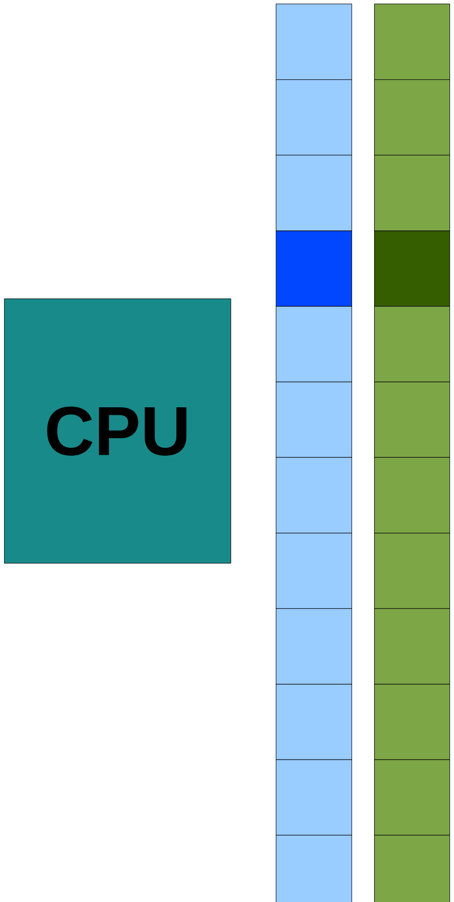
- Explore the use of different data types.
    - Perhaps you can store the same *information* in cache (to a suitable degree of precision), but using less space?
- Make the most of the data that you have in cache already – don't revisit the same data later in your program if you don't have to.

Cache Thrashing: A Synthetic Example

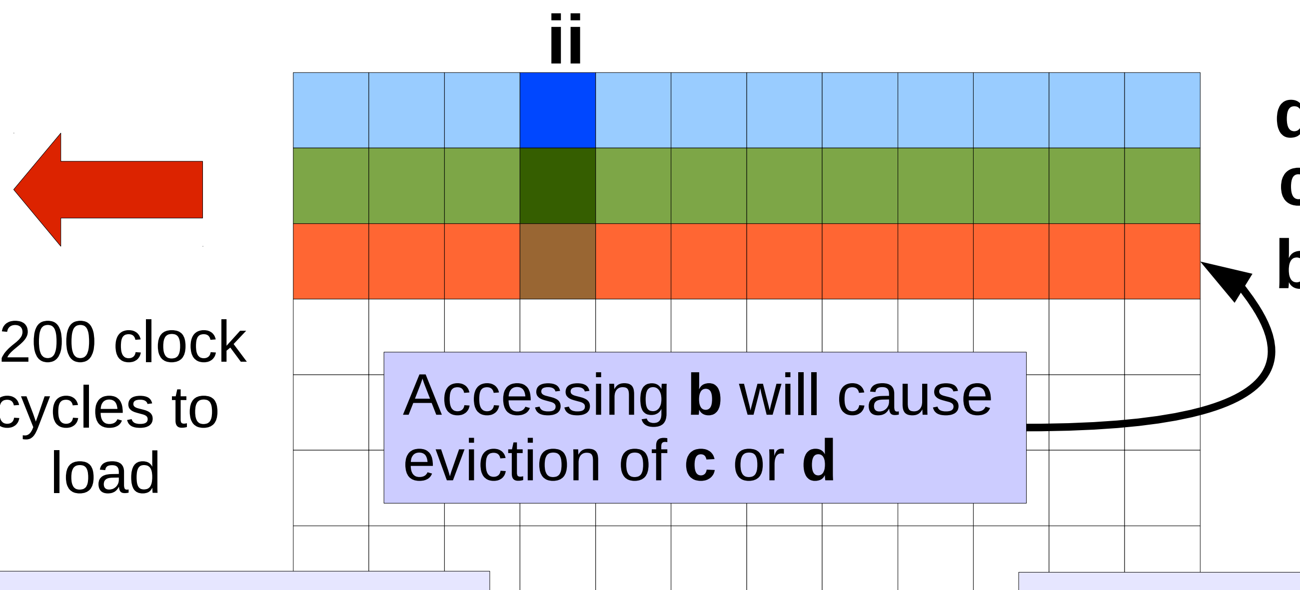
Thrashing is the opposite of re-use:

```
#define max 1024*1024
double a[max], b[max], c[max], d[max]
for(ii=0; ii<max; ii++) {
   a[ii] = b[ii] + c[ii]*d[ii];
}
```

2-way associative cache



~200 clock cycles to load



Best to step contiguously through cache line

(a block of) main memory

University of BRISTOL

Row vs. Column Major Order

2D array

A	B	C
D	E	F
G	H	I

Row Major Order

e.g. C

A	B	C	D	E	F	G	H	I

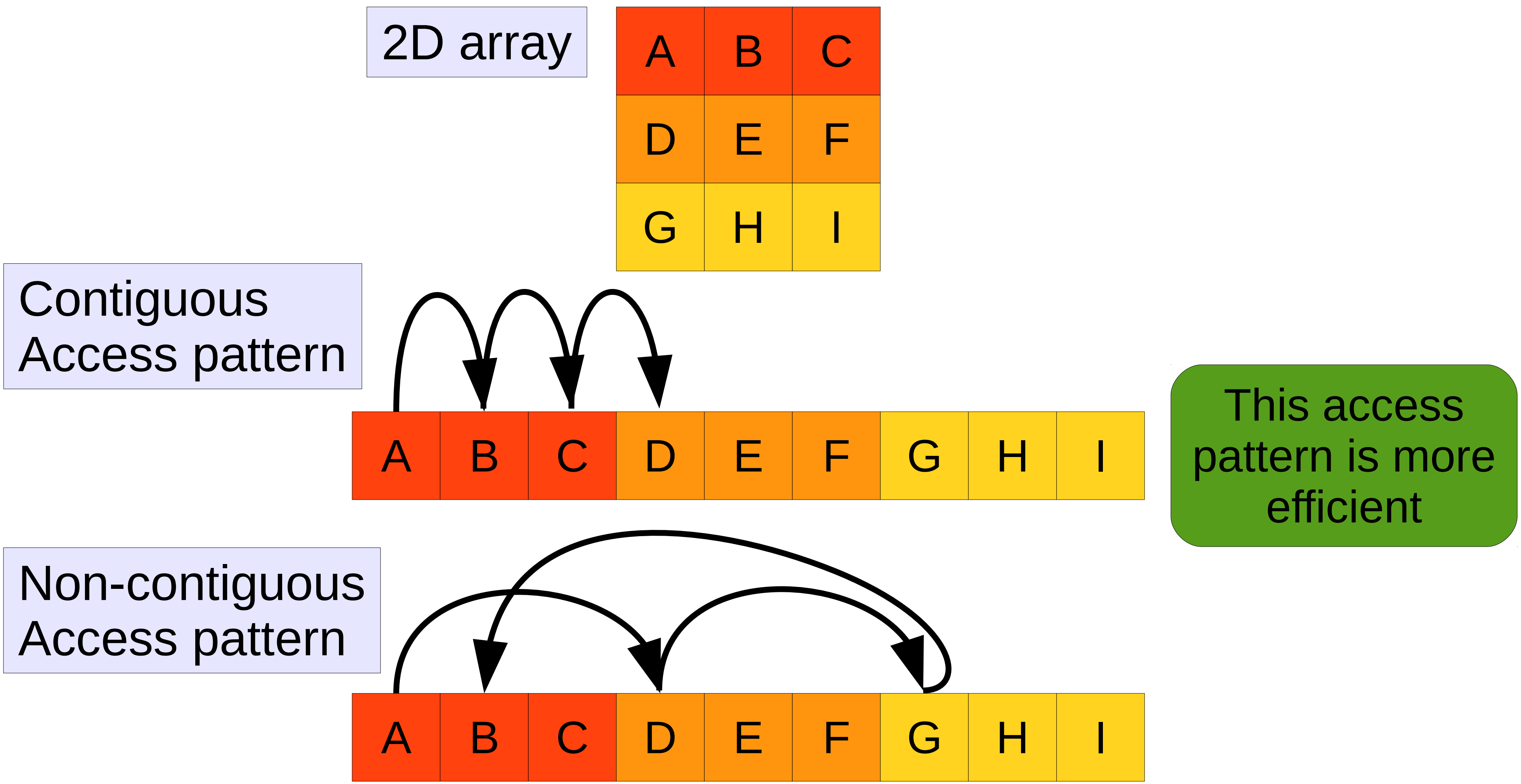
Column Major Order

e.g. Fortran

A	D	G	B	E	H	C	F	I

University of BRISTOL

Row vs. Column Major Order



Don't Re-Visit Memory More Often Than You Need To

A trivial example:

```
for(ii=0; ii<N; ii++) {
    grid[ii] = 0;
}

for(ii=0; ii<N; ii++) {
    grid[ii] += ...
}
```

If N is large, we reload the data into cache..

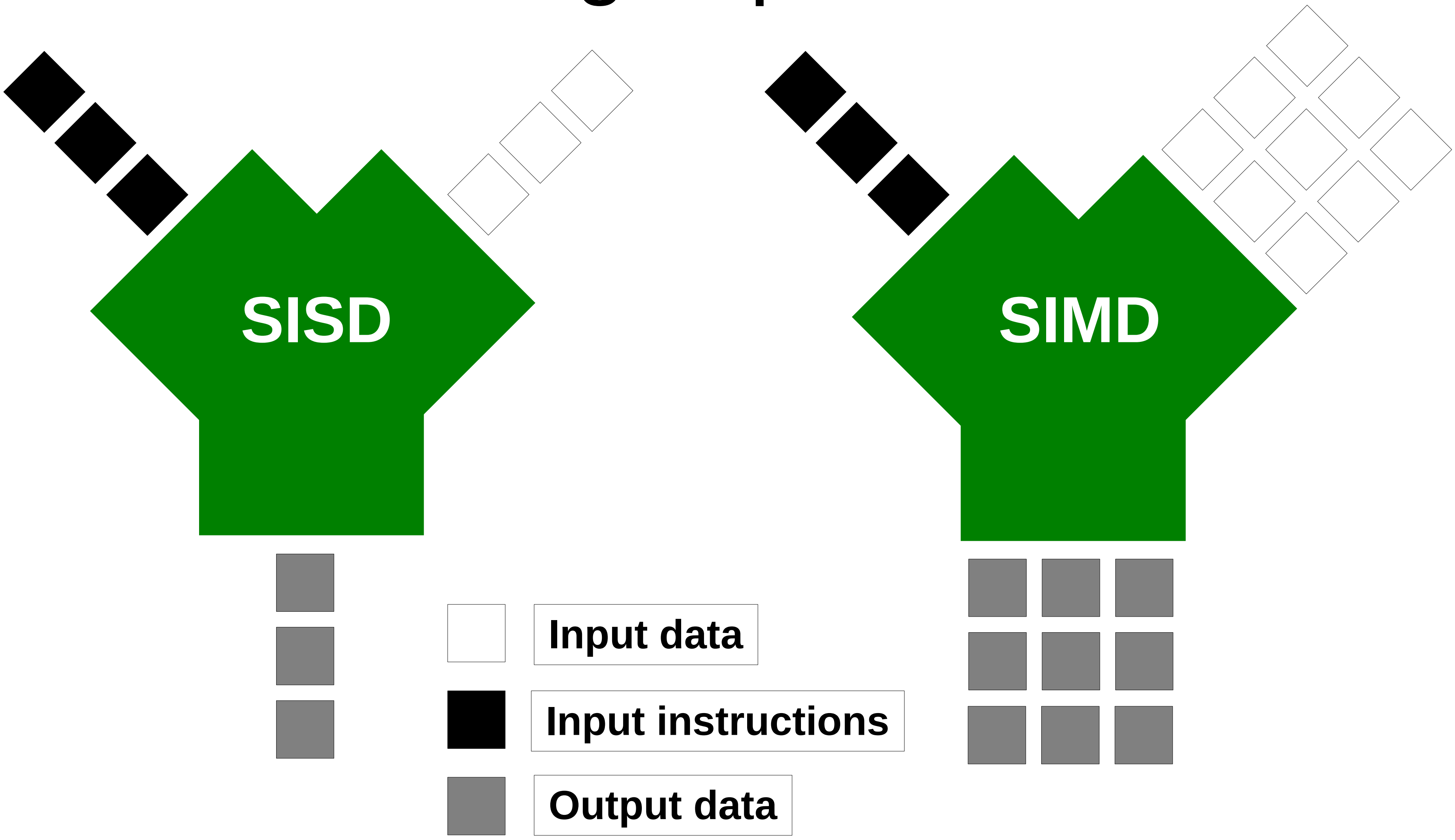
```
for(ii=0; ii<N; ii++) {
    grid[ii] = 0;
    grid[ii] += ...
}
```

merging loops

Look for similar opportunities
in your assignment.
You may need to re-write the code so that the same operations are performed on the data but with potentially radically different source code.

University of BRISTOL

Lesson #5: Vectorisation is of Growing Importance



University of BRISTOL

Trends in Register Width

- SSE: 128bit, e.g. 2 doubles or 4 floats
  - BCp1 (AMD Opteron 2218)
  - BCp2 (Intel Xeon E5462, 'Harpertown')
- AVX: 256bit, e.g. 4 doubles
  - BCp3 (Intel SandyBridge)
- Intel MIC: 512bit, e.g. 8 doubles
  - BCp3
- Employ through:
  - compiler options - portability.
  - language intrinsics – most control (& hence performance?).

Vectorisation – the Principles

- Memory allocations and data structures aligned to the appropriate boundaries (RAM→caches->registers).
- No data dependencies within loop iterations (e.g. a[i] = a[i-1] + 1), as we want to process serveral iterations at once.
- ***Compilers prefer correct programs to fast but incorrect ones!*** Watch out for aliased pointers (you may need to use compiler hints, e.g. #pragma ivdep, 'restrict').
- What function calls are you allowed? Often only inlined and math intrinsics, no switch statements.
- Newer compilers and chipsets offer progressively more support to the programmer: pay attention to the compilers vectorisation reports.

University of BRISTOL

Vectorisation: Many Useful Tutorials

- https://software.intel.com/en-us/articles/vectorization-essential
- http://hpac.rwth-aachen.de/teaching/sem-accg-16/slides/08.Schmitz-GGC_Autovec.pdf
- and many others besides.

Appendix – Using gprof

https://sourceware.org/binutils/docs/gprof/

'Instrument' the code for profiling..

```
cc -O2 -pg myprog.c -o myprog.exe
./myprog.exe
gprof myprog.exe gmon.out >
profile.txt
less profile.txt
```

Will run more slowly with -pg

Parses the output and assembles into something readable..

Summary

- For the fastest possible serial code:
  - Choose the best algorithm.
  - First, examine your within-core performance.
  - Never guess. Do experiment. Always use a profiler.
  - Look at your use of the memory hierarchy.
  - Don't revisit memory more frequently than you need to.
  - To get the most out of modern processors, you will need code that will vectorise well.
- Appendix: Cache thrashing and tiled loops.

University of BRISTOL

Inner Product

$$\begin{array}{ccccc} c & & a & & b \\ \begin{bmatrix} 1x + 2y + 3z \\ 4x + 5y + 6z \\ 7x + 8y + 9z \end{bmatrix} & = & \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} & \mathrm{x} & \begin{bmatrix} x \\ y \\ z \end{bmatrix} \end{array}$$

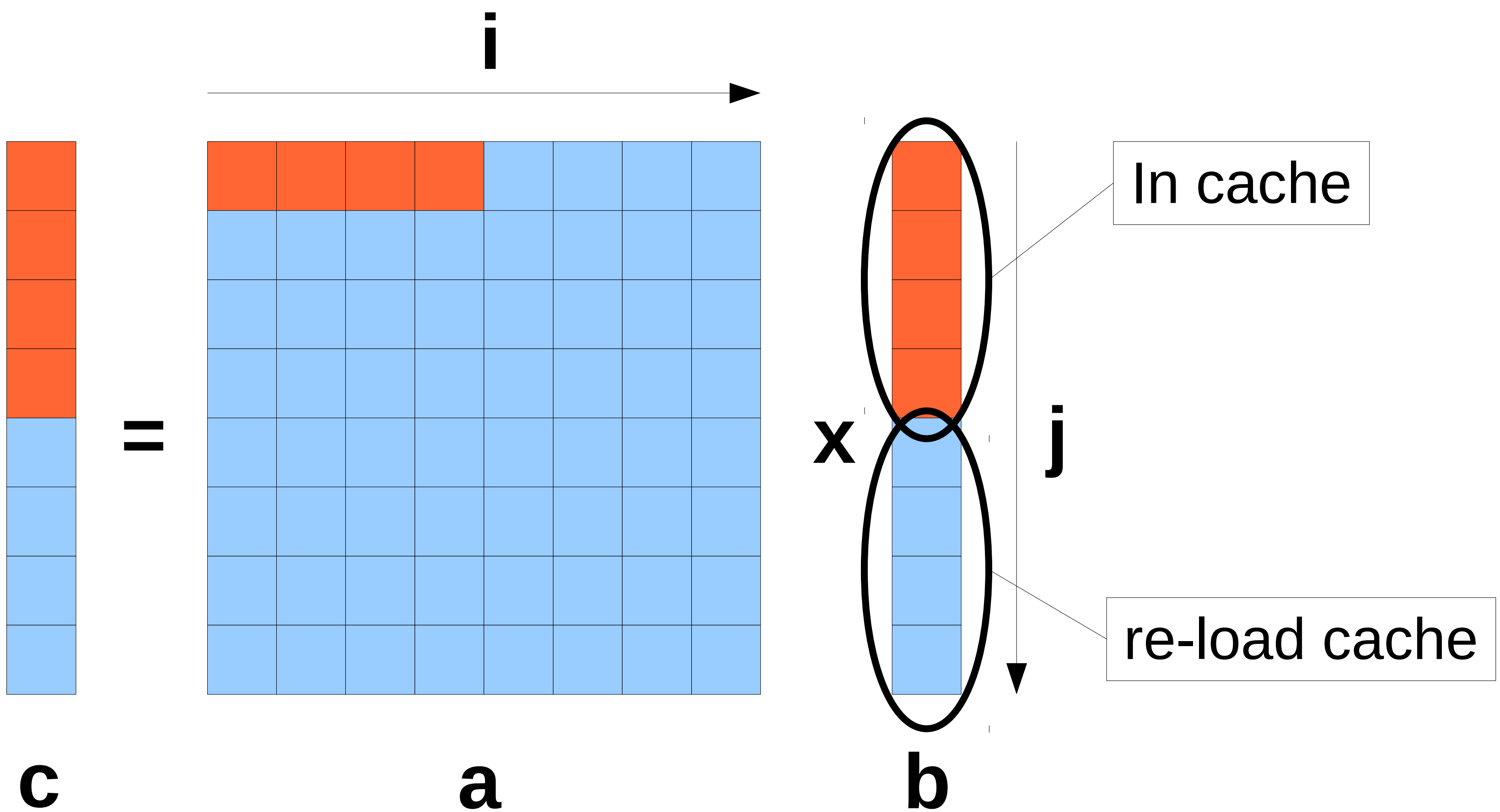
$N=3$

```
/*
** matrix vector multiply--a classic example
** from: http://en.wikipedia.org/wiki/Loop_tiling
*/
for (i = 0; i<N; i++) {
  for (j=0; j<N; j++) {
    c[i] = c[i] + a[i][j] * b[j];
  }
}
```

But what if N is large?

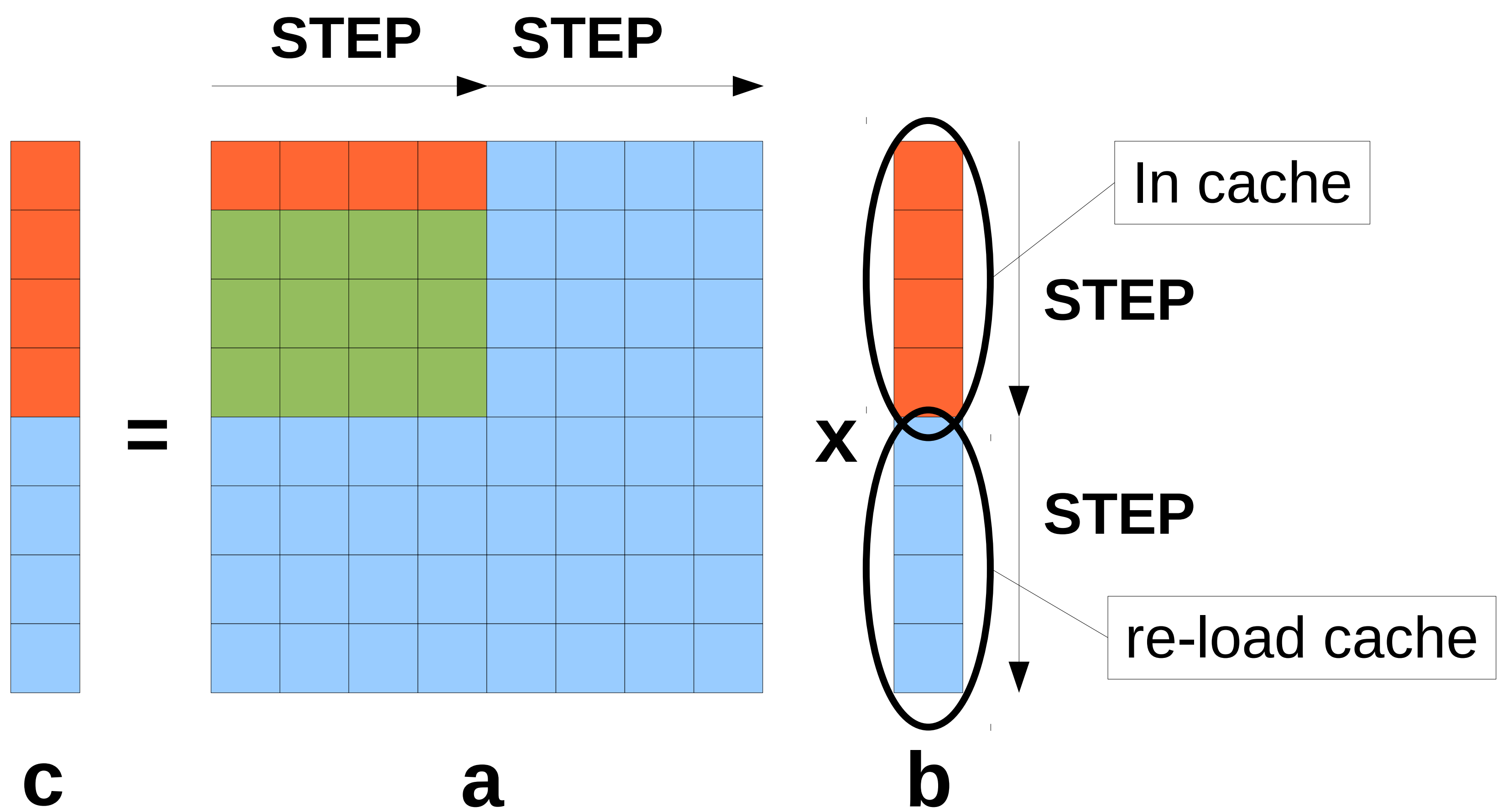
University of BRISTOL

Many Cache Misses



End up re-loading b and c many times

Fewer Cache Misses



Work through arrays in block-size: STEP

University of BRISTOL

'Blocked' Loop: Code

```
/*
** tiling of STEPxSTEP blocks
*/
for (i=0; i<N; i+=STEP) {
  for (j=0; j<N; j+=STEP) {
    for (x=i; x<min(i+STEP, N); x++) {
      for (y =j; y<min(j+STEP, N); y++) {
        r[x] = r[x] + p[x][y] * q[y];
      }
    }
  }
}
```


University of BRISTOL

'Blocked' Loop: Run-time..

gcc v4.4.6 on BCp1

Naive nested loops	
Elapsed time:	2.585584 (s)
Tiled loop	
Elapsed time:	1.743811 (s)

So all the extra coding was worth it!

Naive nested loops	
Elapsed time:	1.900453 (s)
Tiled loop	
Elapsed time:	1.993969 (s)

icc v12.1 on BCp1

Not if your compiler will block loops for you!

University of BRISTOL