ABSTRACTING THE IDEA OF HARDWARE (SIMD) PARALLELISM

Professor Ken Birman
CS4414 Lecture 8

IDEA MAP FOR TODAY

Understanding the parallelism inherent in an application can help us achieve high performance with less effort.

Ideally, by "aligning" the way we express our code or solution with the way Linux and the C++ compiler discover parallelism, we obtain a great solution

There is a disadvantage to this, too. If we write code knowing how that some version of the C++ compiler or the O/S will "discover" some opportunity for parallelism, that guarantee could erode over time.

This tension between what we explicitly express and what we "implicitly" require is universal in computing, although people are not always aware of it

LINK BACK TO DIJKSTRA'S CONCEPT

In early generations of parallel computers, we just took the view that parallel computing was very different from normal computing.

Today, there is a huge effort to make parallel computing as normal as possible.

This centers on taking a single-instruction multiple data model and integrating it with our runtime environment.

IS THIS REALLY AN ABSTRACTION?

Certainly not, if you always think of abstractions as mapping to specific code modules with distinct APIs.

But the abstraction of a SIMD operation certainly aligns with other ideas of what machine instructions "do".

By recognizing this, an advanced language like C++ becomes the natural "expression" of the SIMD parallelism abstraction

OPPORTUNITIES FOR PARALLELISM

Hardware or software prefetching into a cache

File I/O overlapped with computing in the application

Threads (for example, in word count, 1 to open files and many to process those files).

Linux processes in a pipeline

Daemon processes on a computer

VMs sharing some host machine

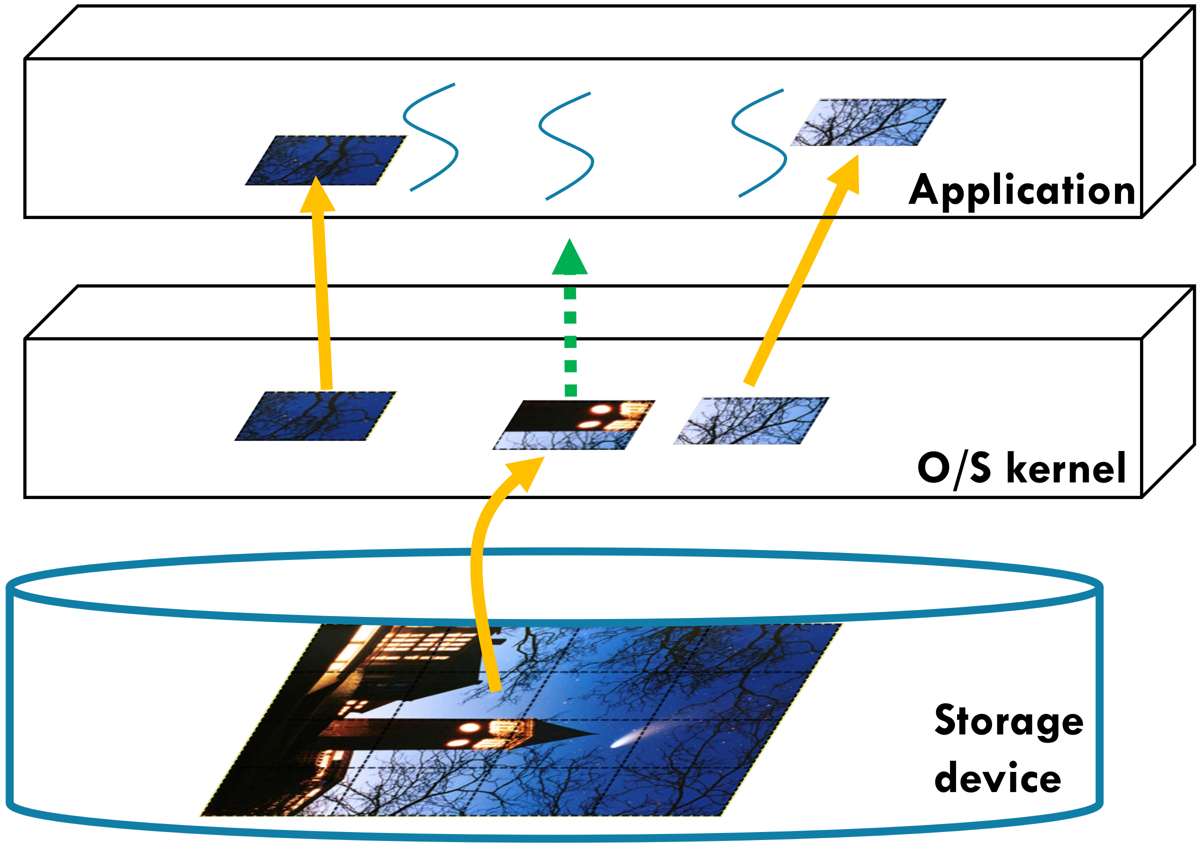
OPPORTUNITIES FOR PARALLELISM

Parallel computation on data that is inherently parallel

- Really big deal for graphics, vision, AI
- These areas are "embarrassingly parallel"

Successful solutions will be ones that are designed to leverage parallel computing at every level!

OPPORTUNITIES FOR PARALLELISM



The application has multiple threads and they are processing different blocks. The blocks themselves are arrays of pixels

Block in the buffer pool was just read by the application. Next block is being prefetched… previously read blocks are cached, for a while

Photo on disk: It spans many blocks of the file. Can they be prefetched while we are processing blocks already in memory?

WHAT ARE THE "REQUIREMENTS" FOR THE MAXIMUM DEGREE OF PARALLELISM?

A task must have all of its inputs available.

It should be possible to perform the task totally independently from any other tasks (no additional data from them, no sharing of any data with them)

There should be other things happening too

WE CALL THIS “EMBARASSING” PARALLELISM

If we have some application in which it is easy to carve off tasks that have these basic properties, we say that it is *embarrassingly easy* to create a parallel solution.

… we just launch one thread for each task.

In word count, Ken’s solution did this for opening files, scanning and counting words in each individual file, and for merging the counts.

SOME PEOPLE CALL THE OTHER KIND “HEROIC” PARALLELISM!

If parallelism isn’t easy to see, it may be very hard to discover!

Many CS researchers have built careers around this topic

ISSUES RAISED BY LAUNCHING THREADS: “UNNOTICED” SHARING

Suppose that your application uses a standard C++ library

If that library has any form of internal data sharing or dependencies, your threads might happen to call those methods simultaneously, causing interference effects.

This can lead to concurrency bugs, which will be a big topic for us soon (but not in today’s lecture)

CONCURRENCY BUGS: JUST A TASTE

Imagine that thread A increments a **node_count** variable, but B was incrementing node_count at the same instant.

Thread A			Thread B		

node_count++	`movq`
	`inc`
	`movq`
		`node_count,%rax`
		`%rax`
		`%rax,node_count`
			node_count++	`movq`
				`inc`
				`movq`
					`node_count,%rdx`
					`%rdx`
					`%rdx,node_count`

We can easily "lose" one of the increments (both load **node_count**), maybe 17. Both increment it (18). Now both store it. The count was incremented twice, yet the value stored is 18, not 19.

HOW ARE SUCH ISSUES SOLVED?

We will need to learn to use locking or other forms of concurrency control (mutual exclusion).

For example, in C++:

```
{
        std::lock_guard<std::mutex> my_lock;
        … this code will be safe …
}
```

LOCKING REDUCES PARALLELISM

Now thread A would wait for B, or vice versa, and the counter is incremented in two separate actions

But because A or B paused, we saw some delay

This is like with Amdahl's law: the increment has become a form of bottleneck!

PARALLEL SOLUTIONS MAY ALSO BE HARDER TO CREATE DUE TO EXTRA STEPS REQUIRED

Think back to our word counter.

We used 24 threads, but ended up with 24 separate sub-counts

- The issue was that we wanted the heap for each thread to be a RAM memory unit close to that thread
- So, we end up wanting each to have its own std::map to count words
- But rather than 24 one-by-one map-merge steps, we ended up going for a parallel merge approach

MORE COSTS OF PARALLELISM

These std::map merge operations are only needed because our decision to use parallel threads resulted in us having many maps.

… code complexity increased

IMAGE AND TENSOR PROCESSING

Images and the data objects that arise in ML are tensors: matrices with 1, 2 or perhaps many dimensions.

Operations like adjusting the colors on an image, adding or transposing a matrix, are embarrassingly parallel. Even matrix multiply has a mix of parallel and sequential steps.

This is why hardware vendors created GPUs.

CONCEPT: SISD VERSUS SIMD

X = Y*3;

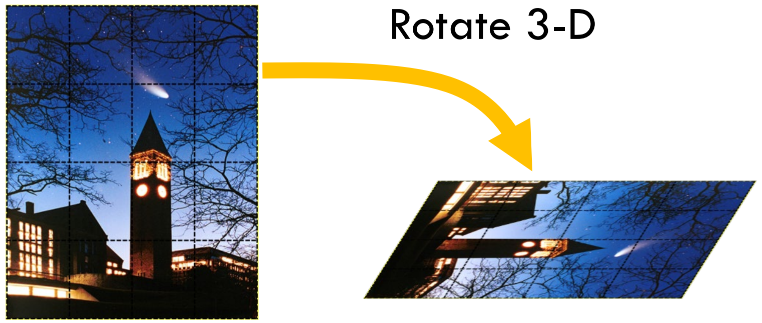
A normal CPU is *single instruction, single data*

An instruction like movq moves a single quad-sized integer to a register, or from a register to memory.

An instruction like addq does an add operation on a single register

So: one instruction, one data item

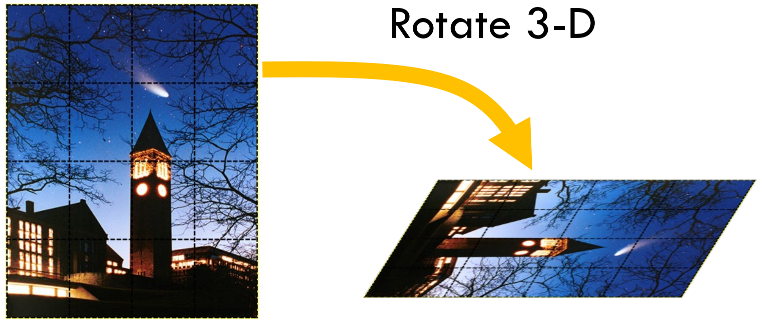
CONCEPT SISD VERSUS SIMD



A SI**<u>M</u>**D instruction is a single instruction, but it operates on a *vector* or *matrix* all as a single operation. For example: apply a 3-D rotation to my entire photo in "one operation"

In effect, Intel used some space on the NUMA chip to create a kind of processor that can operate on multiple data items in a single clock step. One instruction, multiple data objects: SIMD

SIDE REMARK



In fact, rotating a photo takes more than one machine instruction.

It actually involves a matrix multiplication: the photo is a kind of matrix (of pixels), and there is a matrix-multiplication we can perform that will do the entire rotation.

So… a single matrix multiplication, but it takes a few instructions in machine code, **per pixel**. SIMD could do each instruction on many pixels at the same time.

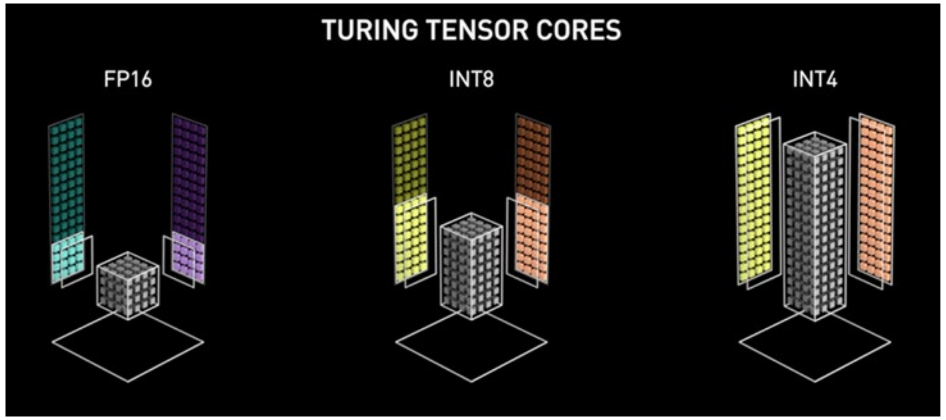
SIMD LIMITATIONS

A SIMD system always has some limited number of CPUs for these parallel operations.

Moreover, the computer memory has a limited number of parallel data paths for these CPUs to load and store data

As a result, there will be some limit to how many data items the operation can act on in that single step!

INTEL VECTORIZATION COMPARED WITH GPU



A vectorized computation on an Intel machine is limited to a total object size of 64 bytes.

- Intel allows you some flexibility about the data in this vector.
- It could be 8 longs, 16 int-32's, 64 bytes, etc.

In contrast, the NVIDIA Tesla T4 GPU we talked about in lecture 4 has thousands of CPUs that can talk, simultaneously, to the special built-in GPU memory. A Tesla SIMD can access a far larger vector or matrix in a single machine operation.

… CS4414 IS ABOUT PROGRAMMING A NUMA MACHINE, NOT A GPU

So, we won’t discuss the GPU programming case.

But it is interesting to realize that normal C++ can benefit from Intel’s vectorized instructions, if your machine has that capability!

To do this we need a C++ compiler with vectorization support and must write our code in a careful way, to “expose” parallelism

SPECIAL C++ COMPILER?

There are two major C++ compilers: gcc from GNU and clang, created by LLVM (an industry consortium)

But many companies have experimental extended compilers. The Intel one is based on Clang but has extra features.

All are "moving targets". For example, C++ has been evolving (C++ 11, 17, 20….) each compiler tracks those (with delays).

THE INTEL VECTORIZATION INSTRUCTIONS

When the MMX extensions to the Intel x86 instructions were released in 1996, Intel also released compiler optimization software to discover vectorizable code patterns and leverage these SIMD instructions where feasible.

The optimizations are only available if the target computer is an Intel chip that supports these SIMD instructions.

INITIALLY, C++ DID NOT SUPPORT MMX

It took several years before other C++ compilers adopted the MMX extensions and incorporated the associated logic.

Today, C++ will search for vectorization opportunities if you ask for it, via -ftree-vectorize or –O3 flags to the C++ command line.

… so, many programs have vectorizable code that doesn't exploit vector-parallel opportunities even on a computer than has MMX

ALSO, INTEL IS NOT THE ONLY CPU DESIGNER

AMD is another major player in the CPU design space. They have their own vector-parallel design, and the instructions are different from the Intel ones (but similar in overall approach).

ARM is an open-source CPU design. It aims at mobile phones and similar systems, and has all sorts of specializations for tasks such as video replay and image or video compression.

MODERN C++ SUPPORT FOR SIMD

Requires -ftree-vectorize or –O3

You must write your code in a vectorizable manner: simple for loops that access the whole vector (the loop condition can only have a simple condition based on vector length), body of the loop must map to the SIMD instructions.

GNU C++ EXAMPLES THAT WOULD PARALLELIZE AUTOMATICALLY

This simple addition
can be done in parallel.

The compiler will eliminate the loop if a single operation suffices. Otherwise it will generate one instruction per "chunk"

Example 1:

```
int a[256], b[256], c[256];
foo () {
  int i;

  for (i=0; i<256; i++){
    a[i] = b[i] + c[i];
  }
}
```

https://gcc.gnu.org/projects/tree-ssa/vectorization.html

GNU C++ EXAMPLES THAT WOULD PARALLELIZE AUTOMATICALLY

Here we see more difficult cases

The compiler can't predict the possible values n could have, making this code hard to "chunk"

Example 2:

```
int a[256], b[256], c[256];
foo (int n, int x) {
   int i;
   /* feature: support for unknown loop bound  */
   /* feature: support for loop invariants  */
   for (i=0; i<n; i++)
      b[i] = x;
   }
   /* feature: general loop exit condition  */
   /* feature: support for bitwise operations  */
   while (n- -){
      a[i] = b[i]&c[i]; i++;
   }
}
```

https://gcc.gnu.org/projects/tree-ssa/vectorization.html

GNU C++ EXAMPLES THAT WOULD PARALLELIZE AUTOMATICALLY

Parallelizing a 2-d matrix seems “easy” but in fact data layout matters.

To successfully handle such cases, the dimensions must be constants known at compile time!

```
Example 8:
int a[M][N];
foo (int x) {
   int i,j;

   /* feature: support for multidimensional arrays  */
   for (i=0; i<M; i++) {
     for (j=0; j<N; j++) {
       a[i][j] = x;
     }
   }
}
```

https://gcc.gnu.org/projects/tree-ssa/vectorization.html

GNU C++ EXAMPLES THAT WOULD PARALLELIZE AUTOMATICALLY

This sum over differences is quite a tricky operation to parallelize!

C++ uses a temporary object, generates the diff, then sums over the temporary array

```
Example 9:
unsigned int ub[N], uc[N];
foo () {
  int i;

  /* feature: support summation reduction.
     note: in case of floats use -funsafe-math-optimizations
*/
  unsigned int diff = 0;
  for (i = 0; i < N; i++) {
    udiff += (ub[i] - uc[i]);
  }
```

https://gcc.gnu.org/projects/tree-ssa/vectorization.html

SUMMARY: THINGS YOU CAN DO

Apply a basic mathematical operation to each element of a vector.

Perform element-by-element operations on two vectors of the same size and layout

Apply a very limited set of conditional operations on an item by item basis

ADVICE FROM INTEL

Think hard about the *layout* of data in memory

- Vector hardware only reaches its peak performance for carefully "aligned" data (for example, on 16-byte boundaries).
- Data must also be densely packed: instead of an array of structures or objects, they suggest that you build objects that contain arrays of data, even if this forces changes to your software design.
- Write vectorization code in simple "basic blocks" that the compiler can easily identify. Straight-line code is best.
- "inline" any functions called on the right-hand of an = sign

WITHIN THAT CODE…

On the right hand slide of expressions, limit yourself to accessing arrays and simple “invariant” expressions that can be computed once, at the top of the code block, then reused.

Avoid global variables: the compiler may be unable to prove to itself that the values don’t change, and this can prevent it from exploring many kinds of vectorization opportunities.

LEFT HAND SIDE…

When doing indexed data access, try to have the left hand side and right hand side “match up”: vectors of equal size, etc.

Build for loops with a single index variable, and use that variable as the array index – don’t have other counters that are also used.

- SIMD code can access a register holding the for-loop index, but might not be able to load other kinds of variables like counters

THINGS TO AVOID

No non-inlined function calls in these vectorizable loops, other than to basic mathematical functions provided in the Intel library

No non-vectorizable inner code blocks (these disable vectorizing the outer code block)

No "data dependent" end-of-loop conditions: These often make the whole loop non-vectorizable

POTENTIAL SPEEDUP?

With Intel MMX SIMD instructions, you get a maximum speedup of about 128x for operations on bit vectors.

More typical are speedups of 16x to 64x for small integers.

Future processors are likely to double this every few years

FLOATING POINT

Given this form of vectorized integer support, there has been a lot of attention to whether floating point can somehow be mapped to integer vectors.

In certain situations this is possible: it works best if the entire vector can be represented using a single exponent, so that we can have a vector of values that share this same exponent, and then can interpret the vector as limited-precision floating point.

C++ VECTORIZATION FOR FLOATS

There is a whole ten-page discussion of this in the compiler reference materials!

With care, you can obtain automatically vectorizable code for floats, but the rules are quite complicated.

… However, GPU programming would be even harder!

COULD THIS SOLVE OUR PHOTO ROTATION?

We can think of a photo as a flat 3-D object. Each pixel is a square. A 3-D rotation is a form of matrix multiplication.

X-Rotation in 3D

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Z-Rotation in 3D

$$\begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale in 3D

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(4x4)*(4x1) = (4x1)

Y-Rotation in 3D

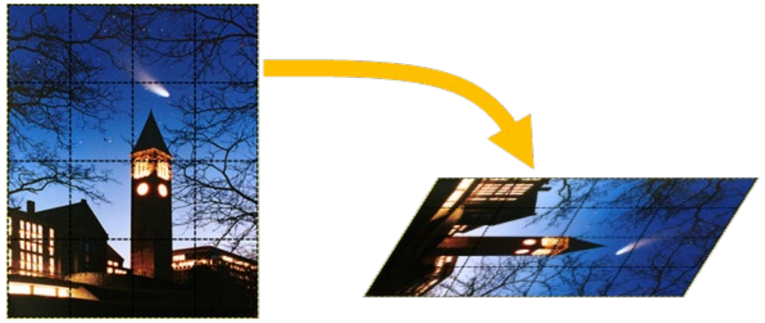
$$\begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Translation in 3D

$$\begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrix Multiplication

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ q \end{bmatrix}$$



TWO FLOATING POINT OPTIONS

We could “construe” our pixels as floating point numbers.

But we could also replace a floating point number by a rational number.

For example: $\pi \cong 22/7$. So, $x^* \pi \cong (x^*22)/7$.

RATIONAL ARITHMETIC LETS US LEVERAGE THE INTEL VECTOR HARDWARE

The Intel vector instructions only work for integers.

But they are fast, and parallel, and by converting rational numbers to integers, we can get fairly good results.

Often this is adequate!

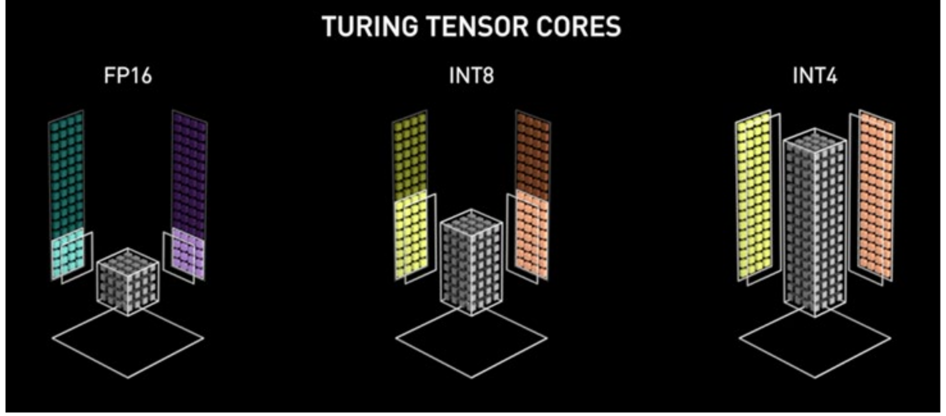
THIS IS WIDELY USED IN MACHINE LEARNING!

We noted that many ML algorithms are very power-hungry

Researchers have shown that often they are computing with far more precision than required and that reduced-precision versions work just as well, yet can leverage these vector-parallel SIMD instructions.

These are available in reduced-precision ML libraries and graphics libraries today.

GPU VERSUS SIMD



Why not just ship the parallel job to the GPU?

- GPUs are costly, and consume a lot of power. A standard processor with SIMD support that can do an adequate job on the same task will be cheaper and less power-hungry.
- Even if you do have a GPU, using it has overheads:

  The system must move the data into the GPU. Like a calculator where you type in the data.

  Then it asks the GPU to perform some operation. "Press the button"

  Then must read the results out.

NEW-AGE OPTIONS

An earlier "new age"

These include TPU accelerators: "tensor processing units"

FPGA: A programmable circuit, which can be connected to other circuits to build huge ultra-fast vision and speech interpreting hardware, or blazingly fast logic for ML.

RDMA: Turns a rack of computers or a data center into a big NUMA machine. Every machine can see the memory of every other machine

STEPPING BACK WE FIND… CONCEPTUAL ABSTRACTION PATTERNS.

When you look at a computer, like a desktop or a laptop, what do you see?

Some people just see a box with a display that has the usual applications: Word, Zoom, PowerPoint…

Advanced systems programmers see a complex machine, but they think of it in terms of *conceptual building blocks*.

SPEED VERSUS PORTABILITY

One risk with this form of abstract reasoning is that code might not easily be portable.

We are learning about SIMD opportunities because most modern computers have SIMD instruction sets (Intel, AMD, etc).

A feature available on just one type of computer can result in a style of code that has poor performance on other machines.

APPLICATIONS CAN HAVE BUILT-IN CHECKS

If you do create an application that deliberately leverages hardware such as a particular kind of vectorization, it makes sense to have unit tests that benchmark the program on each distinct computer.

The program can then warn if used on an incompatible platform: “Portal has not been optimized for your device, and may perform poorly”.

THOUGHT QUESTION

What would be the best way for a program to “learn” which platforms it has been properly tested on?

Some options to consider:

- Linux command line arguments
- Environment variables
- Some form of “compile time constants”
- Code you could implement to “self test” the platform

THOUGHT QUESTION

What would be the best way for a program to "learn" which
plc

So

➢

➢

➢

➢ Code you could implement to "self test" the platform

If at the time we compile a program, we could check the "target" machine (the one we are compiling on) for compatibility, we could only compile versions for hardware in some sort of list of validated options.

The Linux "makefile" technology can be used for this

REVISITING THE ABSTRACTION PERSPECTIVE

If SIMD instructions are the mechanistic feature, what was the abstraction?

For advanced C++ developers, the abstraction here is a coding pattern: a conceptual abstraction – a mental/visualization tool.

When we write code that deliberately exposes parallelism opportunities that match the SIMD hardware, the compiler can “pattern match” and generate SIMD instructions.

SUMMARY

Understanding the computer architecture, behavior of the operating system, data object formats and C++ compiler enables us to squeeze surprising speedups from our system!

Because SIMD instructions have become common, it is worth knowing about them. When you are able to leverage them, you gain speed and reduce power consumption.