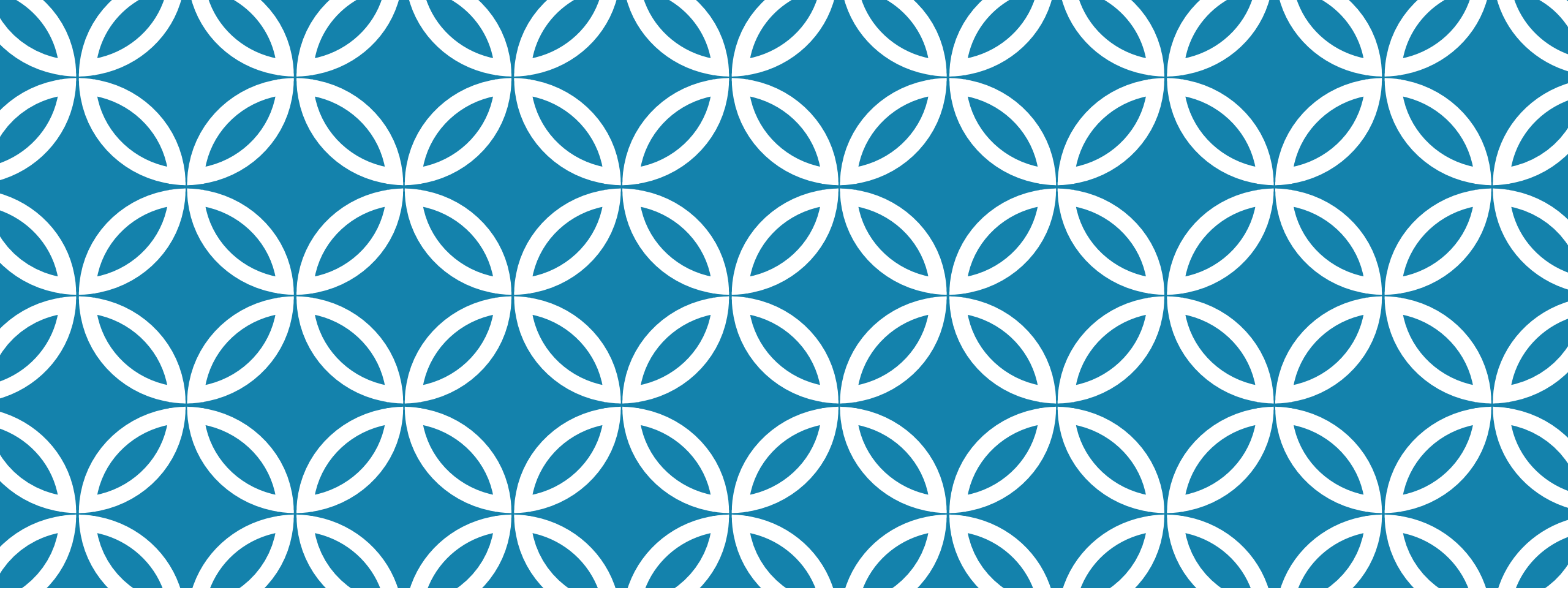


IN SEARCH OF AN ABSTRACT WAY TO THINK ABOUT PARALLELISM

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

MODERN SYSTEMS ARE FULL OF OPPORTUNITIES FOR PARALLISM

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

Parallel instructions in the Intel instruction set (and many others)

A FEW WAYS TO OBTAIN PARALLELISM

Some of these are automatic. E.g.: if Linux notices that the file is being scanned sequentially, it will prefetch blocks.

Some require special logic: To process many blocks in parallel, you launch many threads, one per block. As long as these threads don't interfere with one-another, we get an n-fold speedup.

Some depend on the compiler mapping your code to "parallel instructions" supported by the CPU.

EMBARASSING PARALLELISM

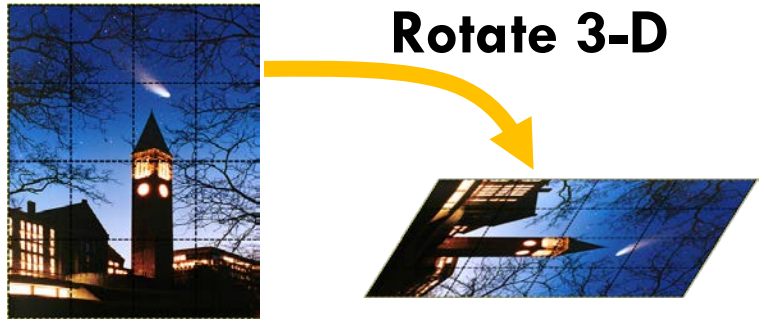
There are entire textbooks and courses on parallel algorithms

But most parallel computing opportunities are totally obvious – things that can easily be done simultaneously if we understand how to "launch" and "control" that pattern of execution.

We call this "embarrassing parallelism" when the opportunity is just sitting there but we neglected to leverage it.

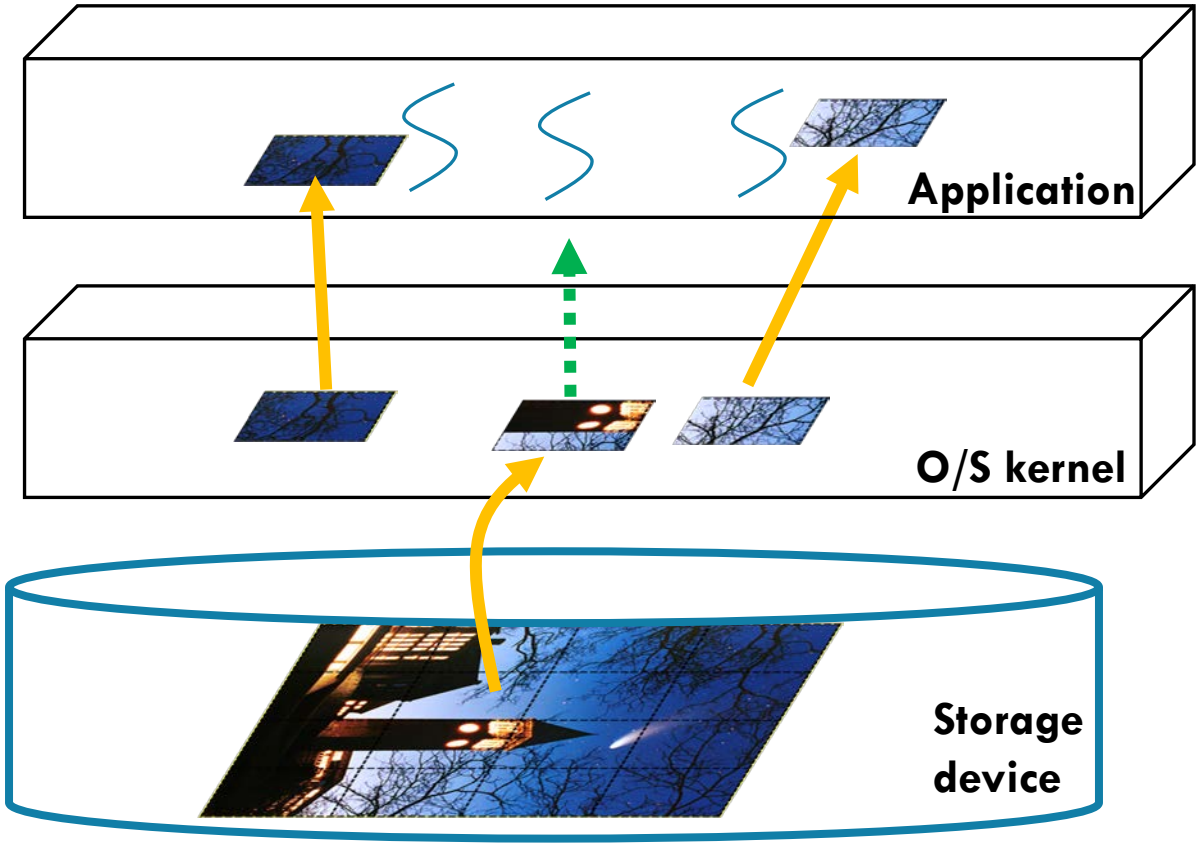
OPPORTUNITIES FOR PARALLELISM

Consider this photo rotation:



Does it have embarrassing parallism in the task?

OPPORTUNITIES FOR PARALLELISM



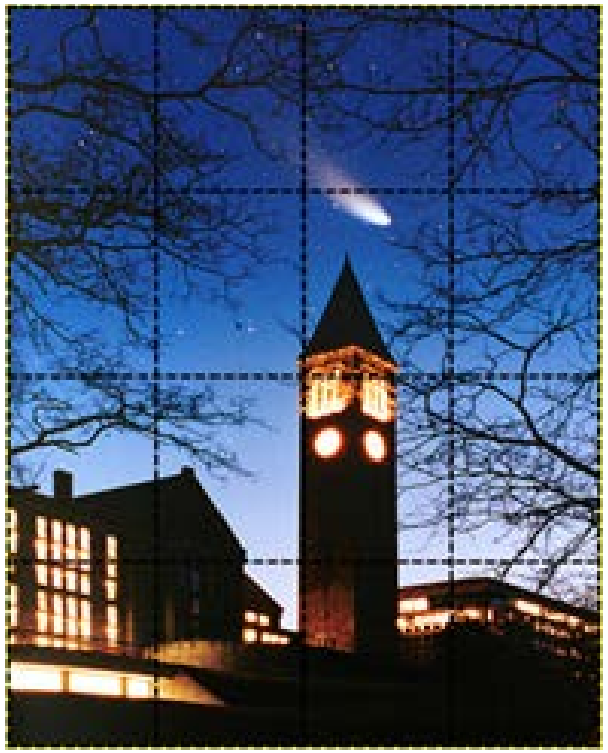
The application has multiple threads and they are processing different blocks.

The blocks themselves are arrays of pixels. We need to multiply each pixel against a small 4x4 tensor describing the rotation

File system could be doing prefetching

On disk, photo spans many blocks

BUT THE EXAMPLE AS SHOWN HAS A "GOTCHA"!



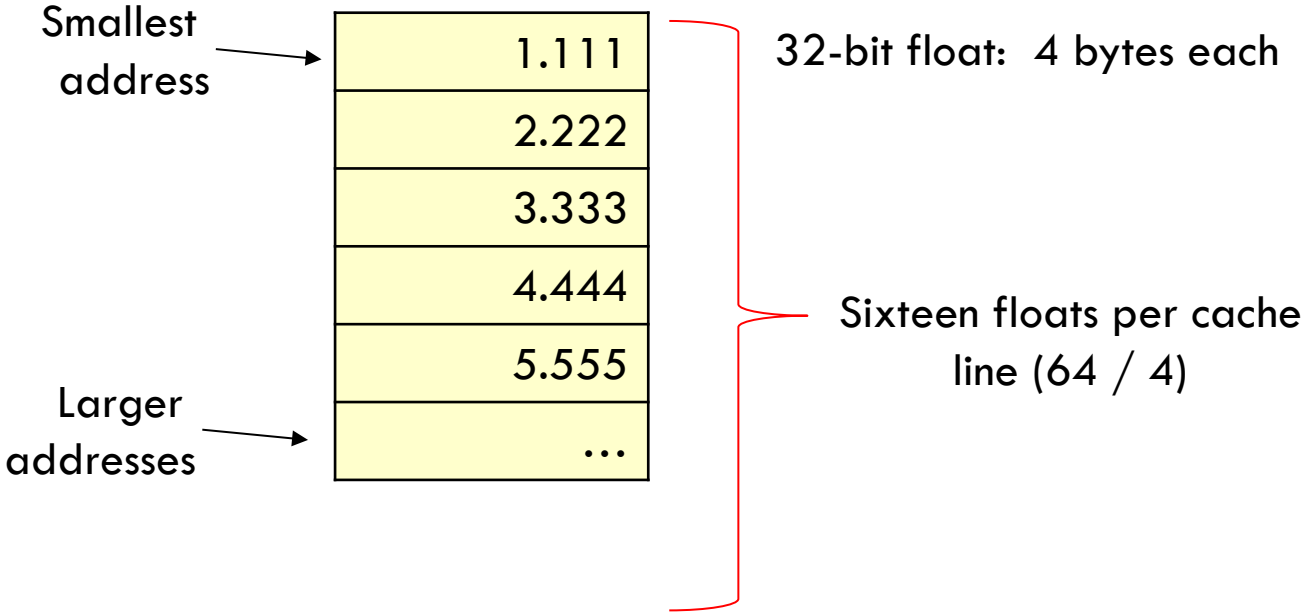
Are these submatrices actually adjacent data, in the image as held in memory?

In C++ (like most languages), a matrix is represented in "row major" layout: first all the data in row 0, sequentially, then row 1, etc.

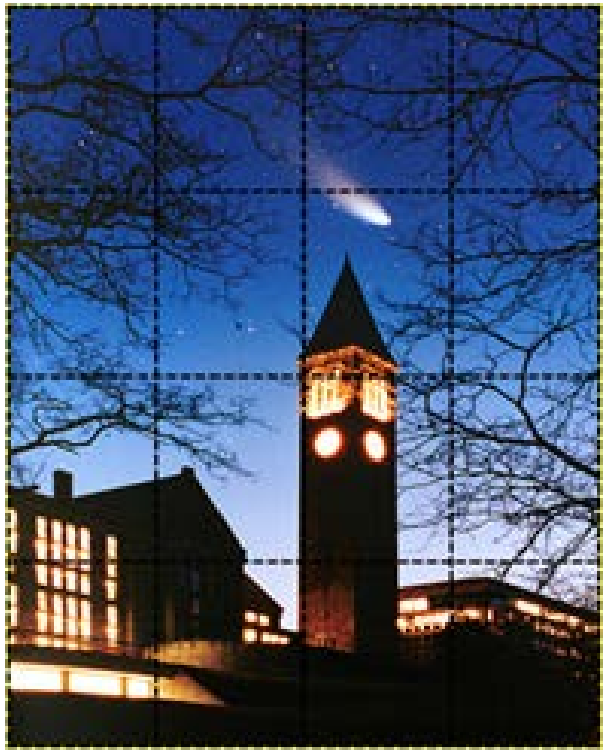
HOW IS AN IN-MEMORY ARRAY REPRESENTED?

float myArray[4][3]

1.111	2.222	3.333
4.444	5.555	….



BUT THE EXAMPLE AS SHOWN HAS A "GOTCHA"!



… so, data in a single row is contiguous. A raster of the image is a row in a matrix!

… and a slice holding several complete rows would also be contiguous

But these submatrices are "scattered" in the larger matrix. They only look contiguous when we visualize the image!

HOW BAD COULD IT BE? … PRETTY BAD!

Potentially, each raster for one of those sub-boxes is in a different disk block. *(Why is this the case?)*

So one thread might need to read hundreds of blocks just to process a single chunk of the rotate task

This will be incredibly slow!

OPPORTUNITIES FOR PARALLELISM

Smarter photo rotation:

Now each slice we are rotating is a contiguous submatrix of the image!

DISK READS?

Scanned from 0 to N, each raster is in one (or more) disk blocks

So each of our rotational tasks do a minimal number of disk reads, and issues them in sequential order

WHAT DOES THIS SAY ABOUT "REQUIREMENTS" FOR THE MAXIMUM DEGREE OF PARALLELISM?

A task must be able to run independently from any other tasks on *data that is independently accessible, and ideally, contiguous and in distinct pages (normally 4K)*

There should be an opportunity to have many of these running

Individual tasks shouldn't "stall" (by waiting for I/O, or paging, or a lock). Our original partitioning of the photo might stall.

ISSUES RAISED BY LAUNCHING THREADS: "UNNOTICED" SHARING

Recall that we want Linux to prefetch each block.

With n threads, we have n separate tasks requesting blocks.

It will be important that Linux still sees these requests in order, as sequential reads. If reads "jump around" in the file, as with our original blocking, Linux won't notice the sequence and won't prefetch. The reads "surprise" the OS and your reading threads stall…

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). Preventing bugs requires locks

… LOCKING CAN INVOLVE WAITING (STALLS).

We will need to learn to use locking or other forms of concurrency control (mutual exclusion). For example, in C++:

```
std::mutex my_mutex;                              // Defines a form of lock
…
{
      std::lock_guard my_lock(my_mutex);    // Obtains the lock, may wait here
      … this code will be safe …
}
```

… LOCKING CAN INVOLVE WAITING… THUS IS ONE EXAMPLE

We will need to learn to
control (mutual exclusion

std::lock_guard works, but modern C++ has other options too.

In an upcoming lecture we will see the "best standard practice", but it involves a C++ language feature we haven't talked about yet

```
std::mutex my_mutex;                                    // Defines a form of lock
…
{
        std::lock_guard my_lock(my_mutex);    // Obtains the lock, may wait here
        … this code will be safe …
}
```

ANY FORM OF STALLING REDUCES PARALLELISM

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

But if A or B paused, we saw some delay

This is like with Amdahl's law: the lock has become a bottleneck!

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

Think back to our word counting programs. *It avoided locks!*

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;

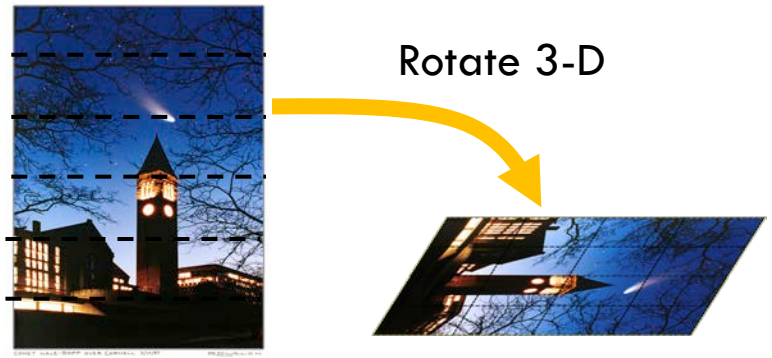
A normal CPU is *single instruction, single data*

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

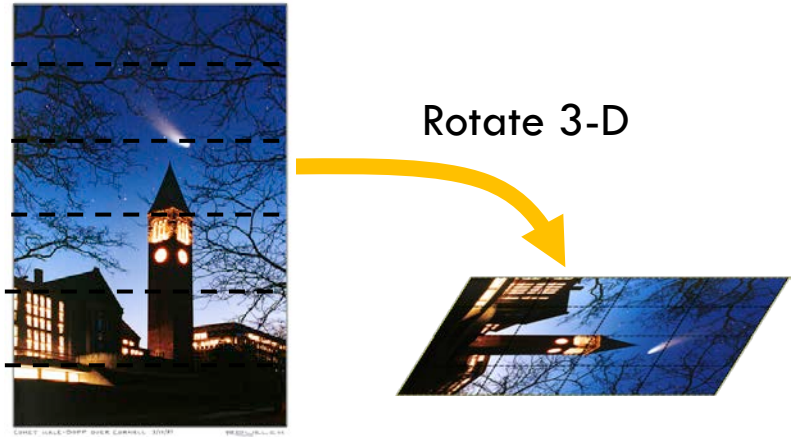
CONCEPT SISD VERSUS SIMD



A SI**M**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.

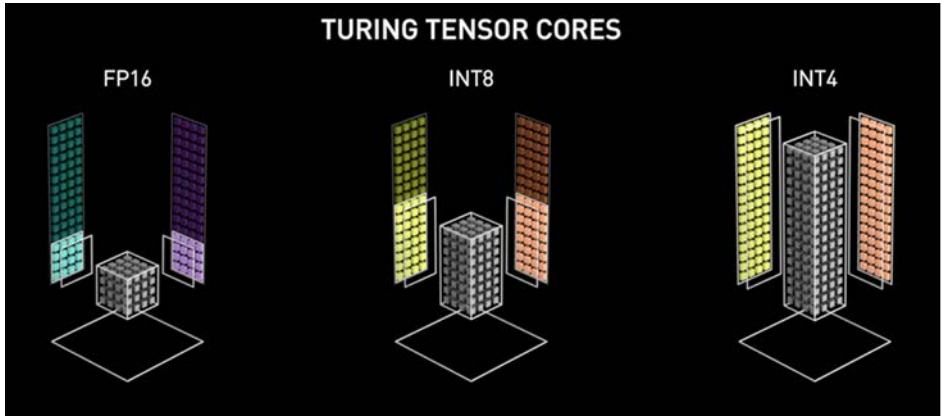
SIMD LIMITATIONS

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.

EXAMPLE: PHOTO ROTATION

With a SIMD approach, we can rotate "one raster at a time"

We would want each raster to be a fixed number of cache lines in length, holding a fixed set of pixels per raster. We also need the entire image object to start on a cache-line boundary, and we need C++ to realize this.

Then we would get a 16x or 32x speedup!

IRREGULAR SIZES?

They will be slower because C++ will generate a mix of pixel by pixel operations and cache-line parallel SIMD ones.

It does this transparently… yet your code will run more slowly!

So… as the developer… you will be rewarded (by a speedup) for designing code to have the ideal properties!

… 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

… AND ABOUT ABSTRACTIONS

Unfortunately, we need new programming language ideas to do a better job of abstracting parallelism opportunities

- Threads work well, and we'll learn about them. Abstracted concurrency.
- But the kind of parallelism where one instruction triggers a "row" of micro-CPUs to transform a whole vector of data in one shot is simply not easy to "abstract". Leveraging it feels very manuel (hands-on).

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

INTEL IS NOT THE ONLY CPU DESIGNER. GCC IS NOT THE ONLY C++ COMPILER…

AMD and ARM are other major players in the CPU design space. They have their own vector-parallel design, and the instructions are different (but similar in overall approach).

Clang is another major C++ compiler. It aligns with GCC on most things, but has slightly different rules for how it detects opportunities to generate parallel code

MODERN C++ SUPPORT FOR SIMD

Requires –O3 option to gcc (older option name: -ftree-vectorize)

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.

EXAMPLE OF A REQUIREMENT

A matrix should be “densely” layed out, in memory, and start on a cache-line boundary (an address that is a multiple of 64)

We mentioned this earlier. Now we will see how it can be harder than it sounds!

EXAMPLE OF A REQUIREMENT

… Thought questions:

- Is a C++ std::vector<float> densely represented in memory?
- What about std::vector<std::vector<float>>?
- Do they start on cache-line boundaries? Even if so, will C++ know this at compile time?

WHY WOULD WE ASK THIS QUESTION?

When reading a table of data ("a structured file") each line generally is read into a std::vector<std::string>, but this is easily converted to a std::vector<float>

```
std::vector<std::string> stringVector = {"3.14", "2.718", "42.0"};

  // Convert strings to floats
  std::vector<float> floatVector;
  for (const auto& str : stringVector) {
      try {
          float value = std::stof(str);
          floatVector.push_back(value);
      } catch (const std::invalid_argument& e) {
          // Handle invalid strings (e.g., non-numeric)
          std::cerr << "Error parsing string: " << str << std::endl;
      }
  }
```

… SO EACH ROW IS A STD::VECTOR<FLOAT>

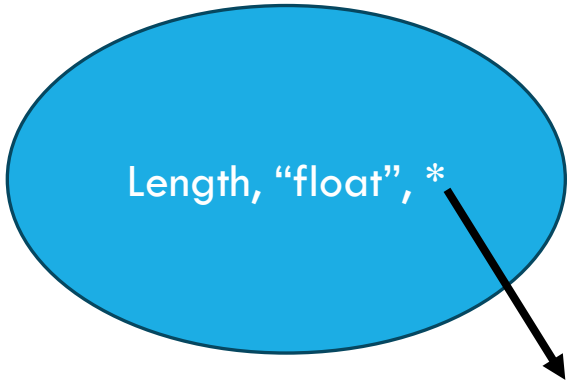
A table with R rows becomes a vector of vectors! Suppose each row has F floats in it.

std::vector<std::vector<float>>, with R*F entries in total

But to leverage parallel instructions, we need this to be physically contigurous in memory

HOW IS STD::VECTOR<FLOAT> "REPRESENTED"?

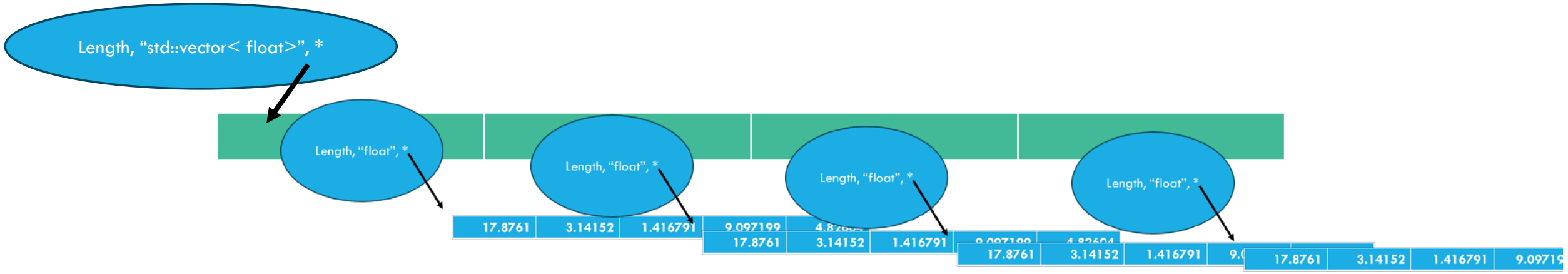
Internally, C++ has a small object holding the length, type and a pointer to the actual data, which is allocated using malloc



17.8761	3.14152	1.416791	9.097199	4.82604

HOW IS STD::VECTOR<STD::VECTOR<FLOAT>> "REPRESENTED"?

Internally, C++ has a small object holding the length, type and a pointer to the actual data, which is allocated using malloc

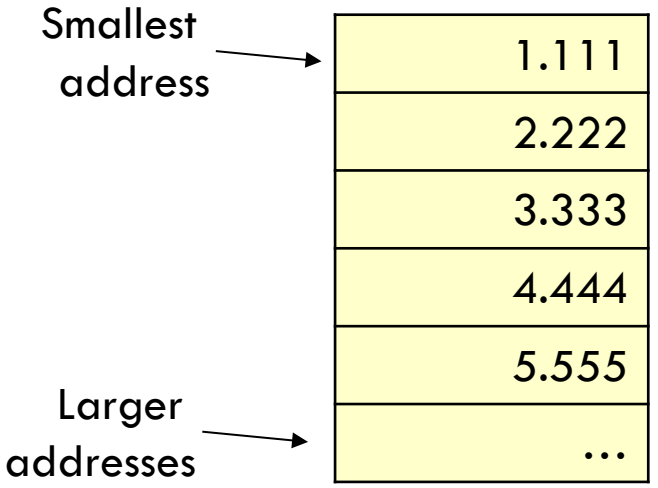


These four vectors might not be contiguous in memory! And unless you use aligned_malloc, they might not be cache-line aligned!

REMINDER: DENSE IN-MEMORY ARRAY IS REPRESENTED SEQUENTIALLY IN MEMORY

float myArray[4][3]

1.111	2.222	3.333
4.444	5.555	….

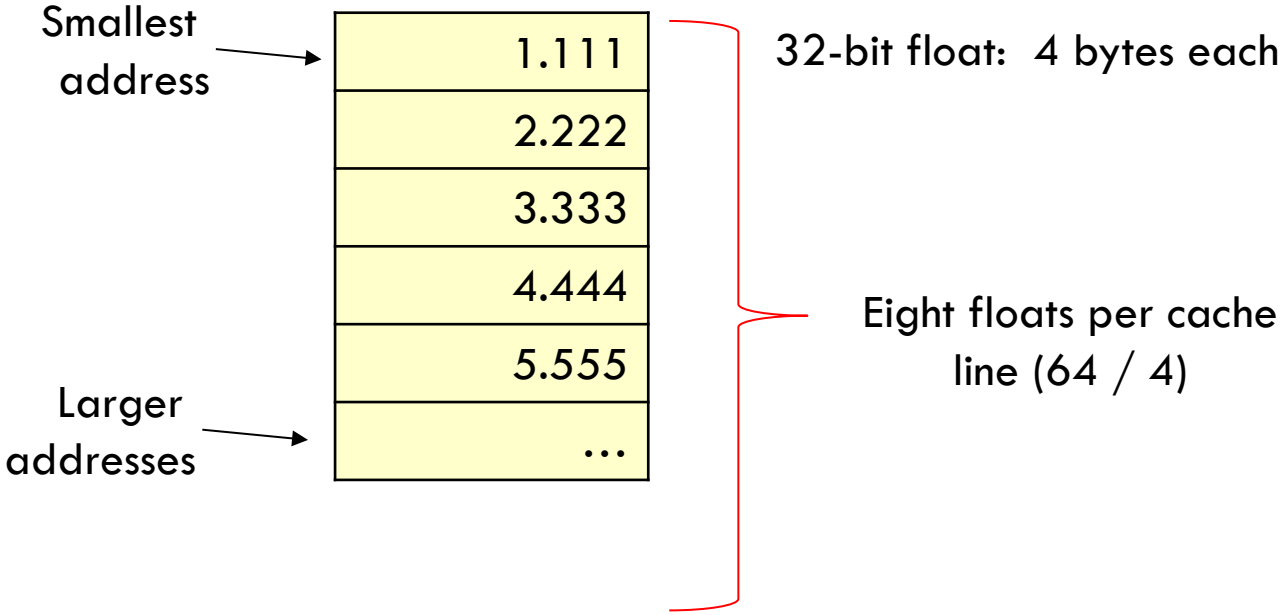


32-bit float: 4 bytes each

…. AND THE MMX INSTRUCTIONS WANT A DENSE SEQUENCE!

float myArray[4][3]

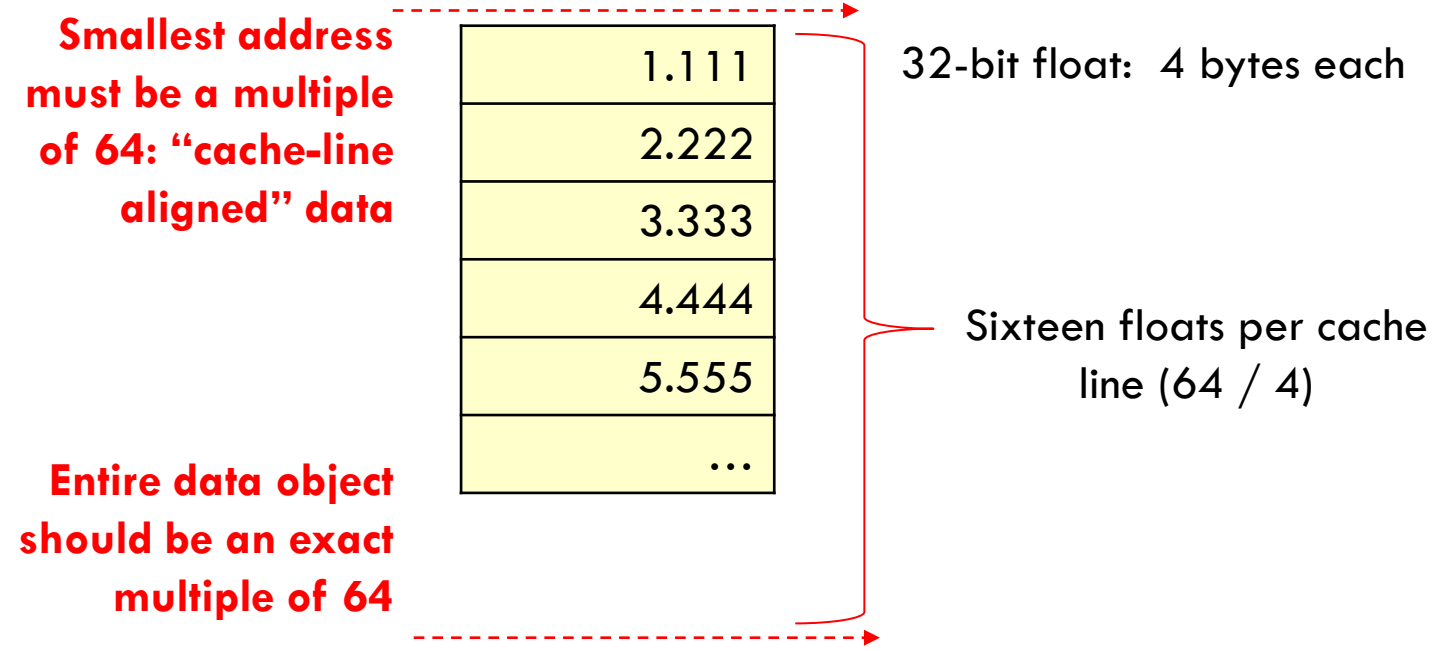
1.111	2.222	3.333
4.444	5.555	….



…. AND THE MMX INSTRUCTIONS WANT A DENSE SEQUENCE!

float myArray[4][3]

1.111	2.222	3.333
4.444	5.555	….



PARALLEL INSTRUCTIONS

They operate on an entire cache line in one shot, or two cache lines for vector-vector operations

- Example: Multiply every float-32 by 2.5
- Example: Y = A + B
- Y = A + B*2.5 requires two instructions
- Can also perform row * column in one instruction

A cache-line is 64 bytes long, and a float-32 is a 4 byte object, so a single instruction performs 16 operations

HOW DOES IT HANDLE THE COLUMNS?

They aren't "sequential in memory", yet MMX also can handle columns because rows have fixed length.

The distance from element k of column k to element k+1 will be exactly the row length plus 1.

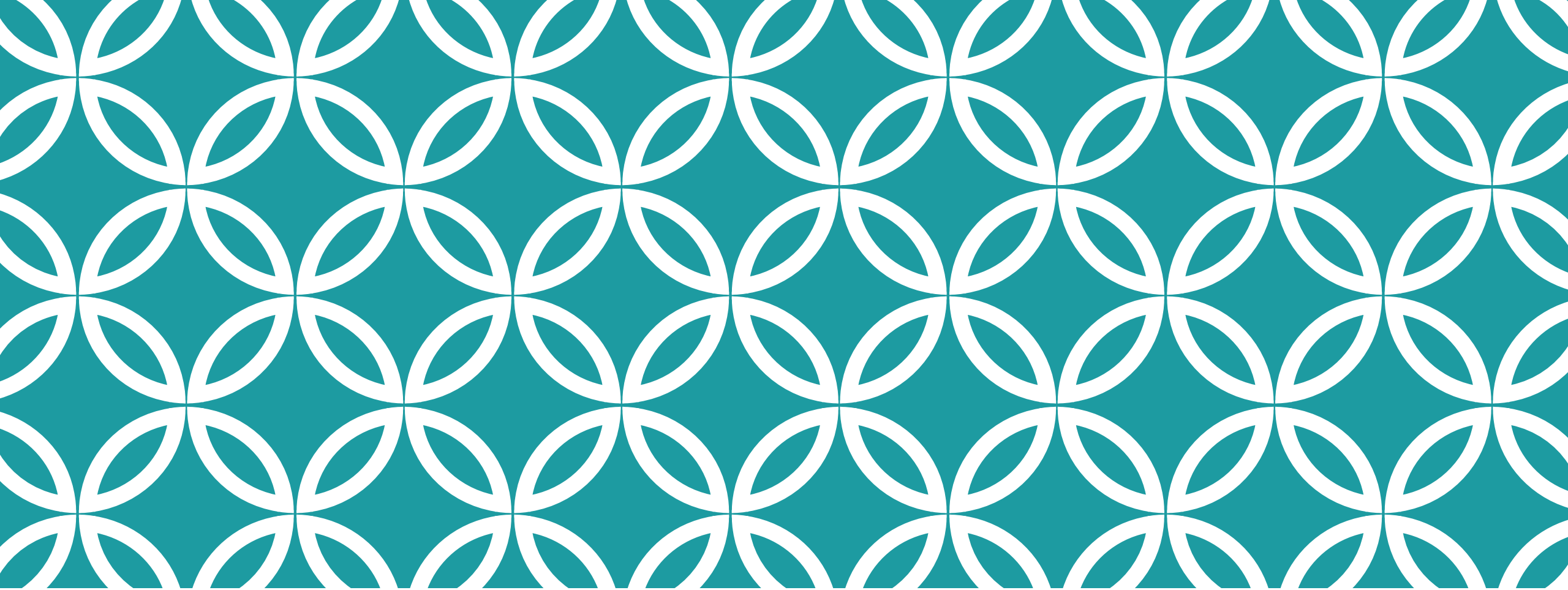
The feature is much faster if row length is power of 2, because it allows MMX to "multiply" using shift-left, which is faster

WHAT ABOUT NON-CACHE-LINE MULTIPLES

Cache-line boundary: A memory address that is a multiple of 64

C++ compiler will use one-by-one logic until it reaches a cache-line boundary, then cache-line-at-a-time logic until there is less than one cache-line of data still to do, then one-by-one again.

This is quite slow, and you'll notice the slowdown if you measure



AND NOW… A GLIMPSE OF THIS WEEK'S RECITATION!

Material Alicia will actually cover!

THE COMPILER NEEDS YOUR HELP!

Random C++ code won't be very vectorizable

But if you code in a careful way, you can arrange for your logic to vectorize nicely. You need to give "hints" to help the compiler

C++ needs to be able to see that the data is properly cache aligned, and dense in memory, and of fixed chunk-sizes that are multiples of the cache-line length

A HELPFUL DATATYPE DECLARATION:

For a vector or matrix declared inline, C++ will automatically memory align it, and track that it did so.

For complex structures, declared inline, C++ might need help. This example is GCC-specific but would work:

```
__declspec(align(64)) struct Str1{
        int a, b, c, d, e;
};
```

WHAT ABOUT POINTERS?

For a pointer, use a declaration like this (GCC-specific):

```
typedef double aligned_double __attribute__((aligned (16)));
// Note: sizeof(aligned_double) is 8, not 16
void some_function(aligned_double *x, aligned_double *y, int n)
{
        for (int i = 0; i < n; ++i) {
                // math!
        }
}
```

… BUT A WARNING

You could “lie” and it would result in strange program crashes

Once you promise to put an aligned pointer into your pointer variable, C++ will trust that you did so, and will generate MMX code that only works with an aligned pointer!

Type checking helps… but would be relatively easy to fool

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, $x*\pi \cong (x*22)/7$. We could relace all operations involving π with 22/7: integer arithmetic!

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!

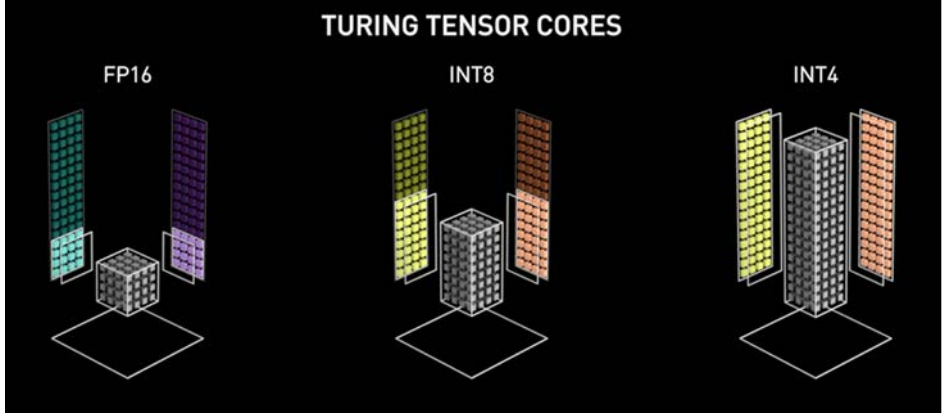
THIS IS WIDELY USED IN MACHINE LEARNING!

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.

An earlier "new age"

NEW-AGE OPTIONS

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: "This program has not been optimized for your device, and may perform poorly".

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.