Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Parallel Programming

Lecture 3

István Reguly
reguly.istvan@itk.ppke.hu

Data dependencies

- Most directive based parallelisation consists of splitting up big for loops into independent chunks that the many processors can work on simultaneously

- For example in case of a simple loop

```
// vector add example
for (int i = 0; i < 100000; i++) {
            c[i] = a[i] + b[i]
}
```

- You can run on a 1000 processors, each executing a 100 iterations

No data dependencies

```
// Processor 1
for (int i = 0; i < 100; i++) {
            c[i] = a[i] + b[i]
}
```

…

```
// Processor 2
for (int i = 100; i < 200; i++)
{
            c[i] = a[i] + b[i]
}

// Processor 3
for (int i = 200; i < 300; i++)
{
            c[i] = a[i] + b[i]
}
```

```
// Processor N
for (int i = N*100; i < (N+1)*100; i++) {
            c[i] = a[i] + b[i]
}
```

Data dependency

- What if the iterations of the loop are not independent?

- Take the following loop

```
// Another example: inclusive scan
for (int i = 0; i < 100000; i++) {
          c[i] = c[i] + c[i-1]
}
```

i=100 -> need c[99]

```
// Processor 1
for (int i = 0; i < 100; i++) {
          c[i] = c[i] + c[i-1]
}
```

```
// Processor 2
for (int i = 100; i < 200; i++) {
          c[i] = c[i] + c[i-1]
}
```

- There is a “loop carried dependency”: need result from previous iteration

Data dependencies

- This is a data dependency – if the compiler even suspects that there is a data dependency, it will, for the sake of correctness, refuse to parallelise that loop

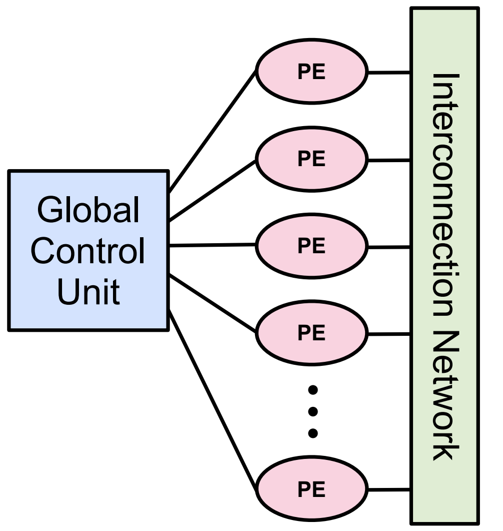
```
Loop carried dependence of 'c' prevents parallelization
Loop carried backward dependence of 'c' prevents vectorization
```

- What can you do?
  - Rearrange code to make clear that there is not really any data dependency
  - Eliminate a real dependency by changing the code
  - Override the compiler’s judgement risking invalid results. `#pragma omp for` does just that
- Why is this important?
  - Looking at your for loops, compilers will try to automatically parallelise them, particularly with Single-Instruction-Multiple-Data

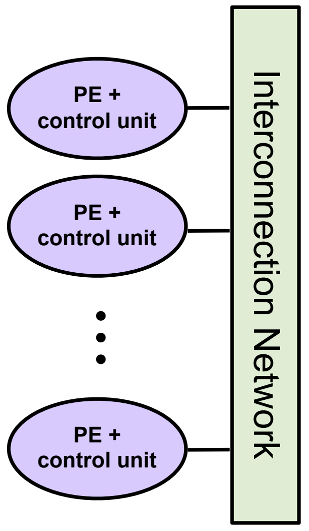
Control Structure of Parallel Platforms

- Parallelism ranges from instructions to processes
- Processor control structure alternatives:
  - Work independently
  - Operate under the centralised control of a single control unit
- MIMD
  - Multiple Instruction streams
    - Each processor has its own control unit
    - Each processor can execute different instructions
  - Multiple Data streams
    - Processors work on their own data
- SIMD
  - Single Instruction Stream
    - Single control unit dispatches the same instruction to processors
  - Multiple Data Streams
    - Processors work on their own data

SIMD and MIMD processors



SIMD architecture



MIMD architecture

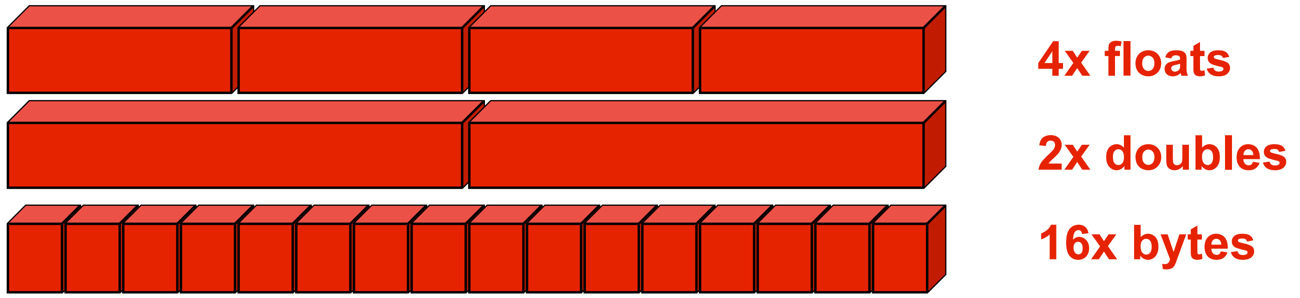
PE = Processing Element

SIMD control

- SIMD is very good with computations that have a regular structure
  - Media processing, linear algebra, etc...
- Activity mask
  - Per PE predicated execution: turn off operations on certain PEs
    - Each PE tests own conditional and sets own activity mask

Example: 128-bit SIMD vectors

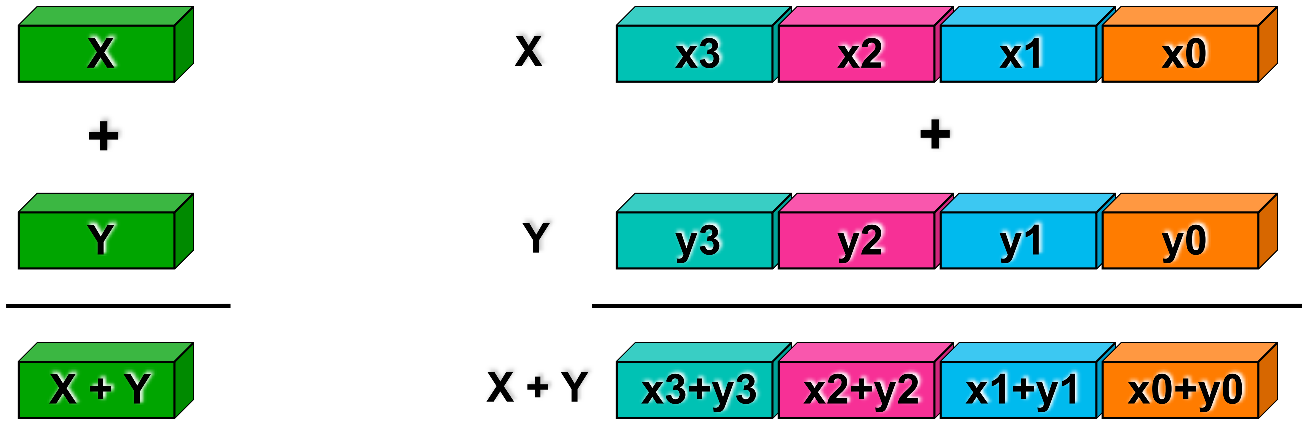
- Data types: anything that fits into 16 bytes, e.g.



- Instructions operate in parallel on data in this 16 byte register
  - Add, multiply, etc.
  - Data bytes must be contiguous in memory and aligned
- Additional instructions needed for:
  - Masking data
  - Moving data from one part of a register to another

Computing with SIMD units

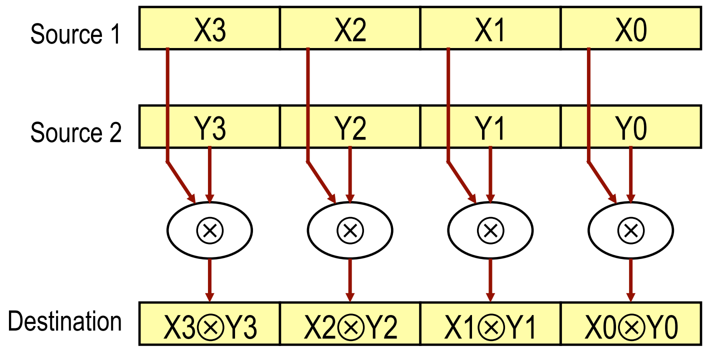
- Scalar processing
  - One operation produces one result
- SIMD vector units
  - One operation produces multiple results



Alex Klimovitski & Dean Macri, Intel Corporation

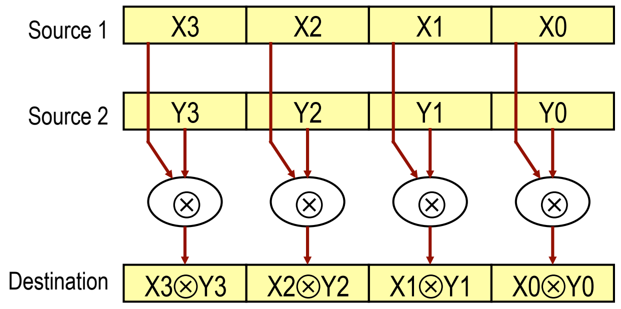
SIMD operation

- SIMD execution
  - Performs an operation in parallel on an array of 2,4,8,16, or 32 values, depending on the size of the values
  - Data parallel operation
  - The operation can be
    - Data movement
    - Arithmetic instruction
    - Logical instruction
    - Comparison instruction
    - Conversion instruction
    - Shuffle instruction

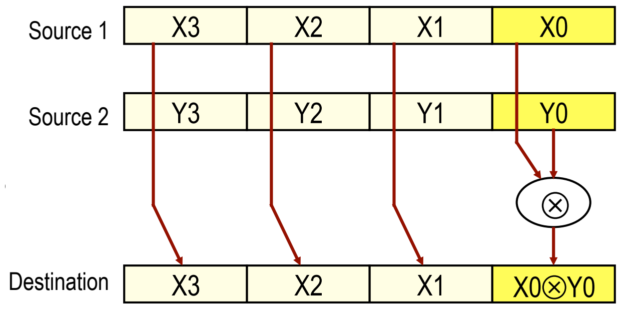


Packed and scalar operations

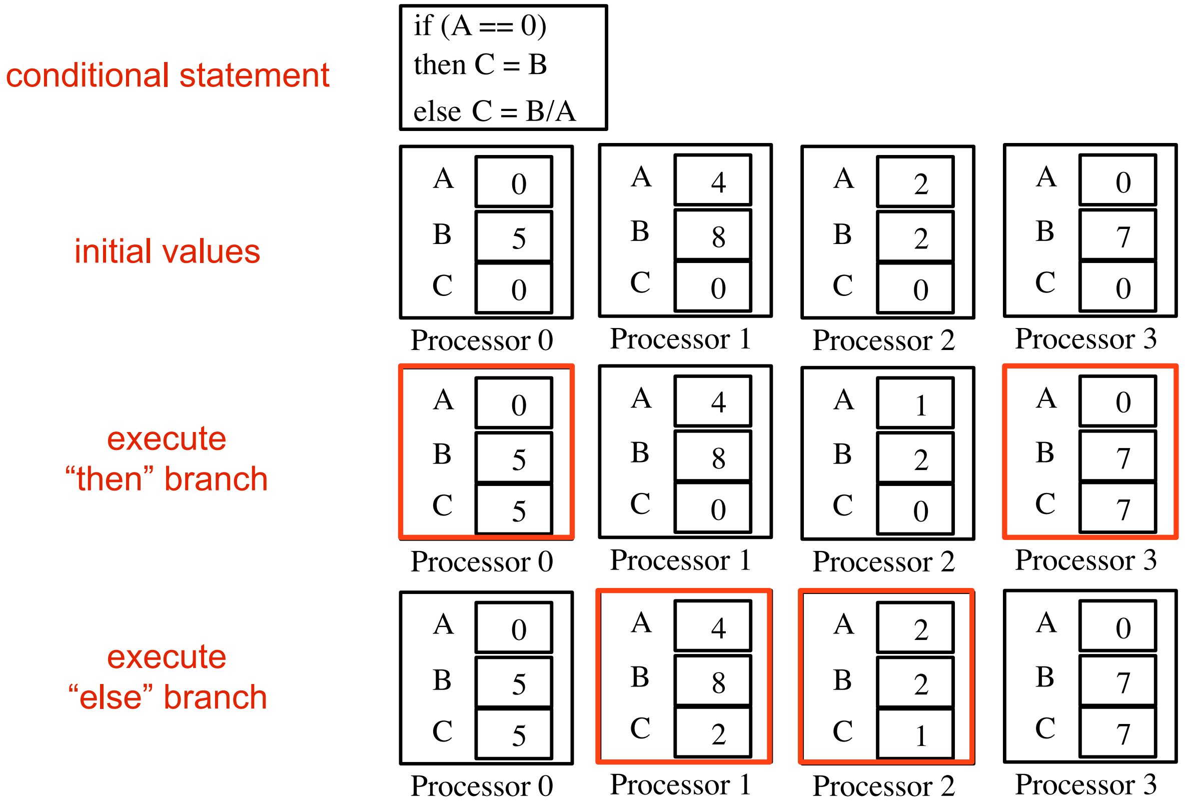
- Packed operations apply in parallel to 2, 4 or 8 floating-point values



- Scalar operations apply an operation on a single floating-point value
- Cost is the same!



Conditional execution

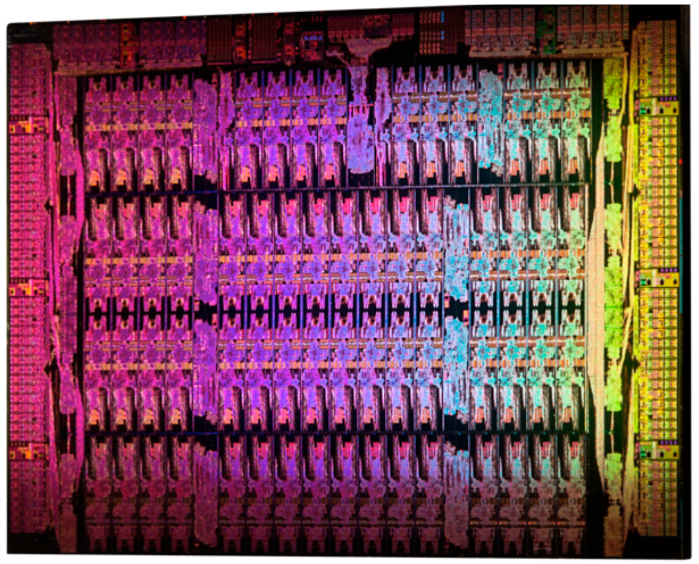


SIMD examples

- Historically: SIMD computers
  - Connection Machine CM-1/2: 65536 1-bit processors
- Today: SIMD units or accelerators
  - Vector units
    - SSE/2/3/4 – Streaming SIMD Extensions
      - 128 bit registers
    - AVX/2/512 – Advanced Vector Extensions
      - 256 or 512 bit registers (Sandy Bridge/Skylake)
  - Co-processors
    - NVIDIA GPUs
    - Intel Xeon Phi

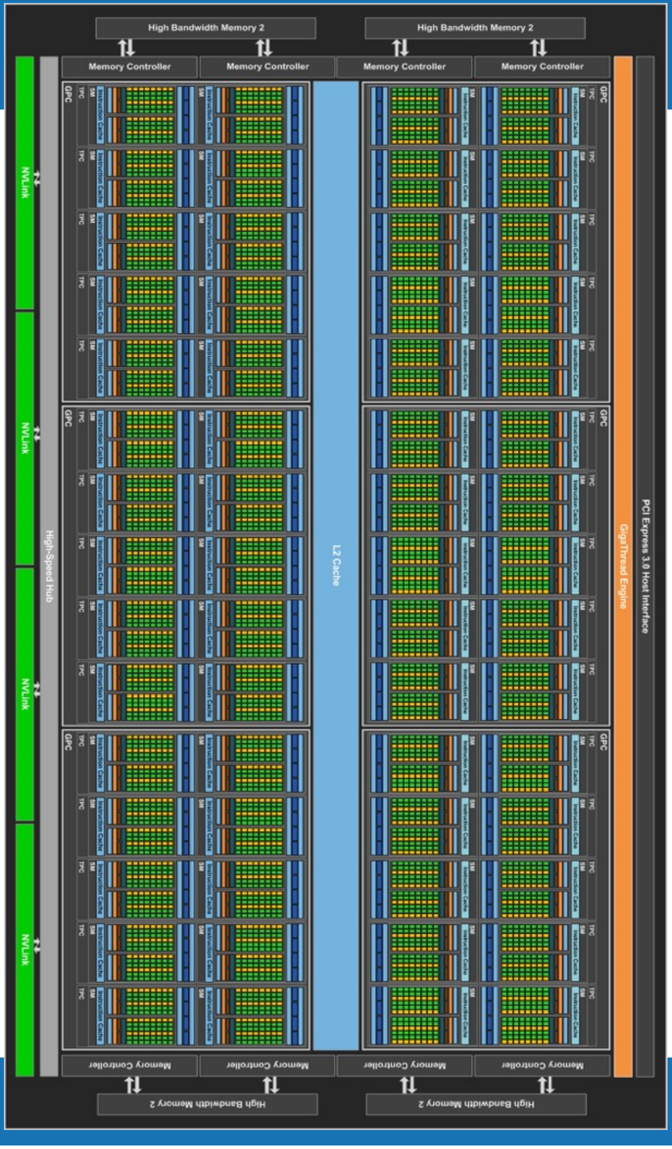
Intel Xeon Phi

- 64 cores
  - 64 KB L1 cache
  - 512 KB L2 cache
  - 32 512-bit vector registers
  - 4-way SMT per core
- Up to 1.5 GHz
- 16 GB Stacked memory
  - Max 450 GB/s
- 3 TFlops



NVIDIA A100 GPU

- 108 Streaming Multiprocessors (SMX)
- Each SMX
  - 64 CUDA cores
    - Fully pipelined FP and INT units
  - Four warp schedulers
    - 32-thread groups (warps)
    - 4 warps issue and execute concurrently
    - 2 instructions/warp/cycle
  - 64 double precision units
  - 32 Load-Store Units
  - 32 Special Function Units
- 9.7/19.5 TFLOP



Programming for vector units

- Different ways to use vector instructions in your program

1. Automatic vectorisation by the compiler
   - No explicit vectorised programming required, but we have to arrange the code so that the compiler can recognize possibilities for vectorisation
2. Express the computation as arithmetic expressions on vector data types
   - Declare variables of a vector type
   - Express computations as normal arithmetic expressions
3. Use compiler intrinsic functions for vector operations
   - Functions that implement vector instructions in a high-level language
   - Requires detailed knowledge of the vector instruction set

Programming with intrinsics

- We really do not want to write vector code ourselves...

```
void saxpy(int n, float alpha, float *X, float *Y) {
  for (int i=0; i<n; i++)
    Y[i] = alpha*X[i] + Y[i];
}

void saxpy(int n, float alpha, float *X, float *Y) {
  __m128 x_vec, y_vec, a_vec, res_vec; /* Vector variables */
  a_vec = _mm_set1_ps(alpha);
  for (int i=0; i<n; i+=4) { /* Vector of 4 alpha values */
    x_vec = _mm_load_ps(&X[i]); /* Load 4 values from X */
    y_vec = _mm_load_ps(&Y[i]); /* Load 4 values from y */
    /* Compute */
    res_vec = _mm_add_ps(_mm_mul_ps(a_vec, x_vec), y_vec);
    _mm_store_ps(&Y[i], res_vec); /* Store the result */
  }
}
```

Automatic vectorization

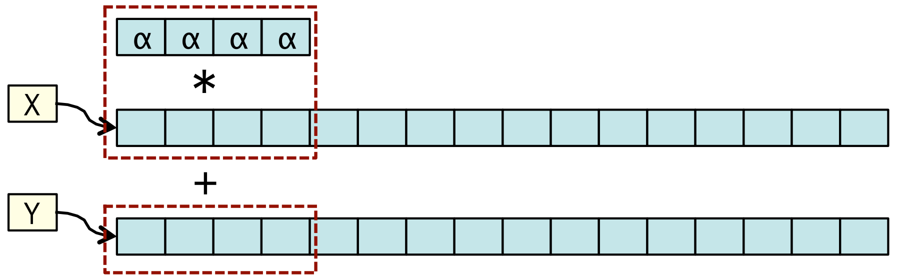
- Requires a compiler with vectorising capabilities
  - In g++, enabled above -O3 (add -ffast-math), or -Ofast
  - Intel compiler is the best
- The compiler automatically recognises loops that can be implemented with vectorised code
  - Really difficult thing to do, the compiler has to guarantee that correct code is generated!
- Help the compiler by giving hints:
  - Pointers with “__restrict”: no aliasing
  - Align arrays to 32 bytes: faster load

Example: SAXPY

- Single-precision Alpha X Plus Y

```
void saxpy(int n, float alpha, float *X, float *Y) {
  for (int i=0; i<n; i++)
    Y[i] = alpha*X[i] + Y[i];
}
```

- Vectorised code will do the computation on 4 values at a time



Try: CC saxpy.cpp -o saxpy -Ofast -fopenmp

Using compiler vectorisation

- Use the compiler switches –Ofast and -fopt-info[-vec] (for gcc) to see reports about which loops were vectorised

```
g++ saxpy.cpp -o saxpy -Ofast –std=c++11 -fopenmp -fopt-info-vec
saxpy.cpp:8:3: note: loop vectorized
saxpy.cpp:8:3: note: loop versioned for vectorization because of possible aliasing
```

Later g++ more clever:

```
saxpy.cpp:8:18: optimized: loop vectorized using 16 byte vectors
saxpy.cpp:8:18: optimized: loop versioned for vectorization because of possible aliasing
saxpy.cpp:8:18: optimized: loop vectorized using 8 byte vectors
saxpy.cpp:8:18: optimized: loop vectorized using 16 byte vectors
```

Cray CC

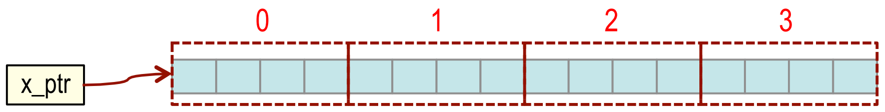
```
CC saxpy.cpp -o saxpy -Ofast –fopenmp -Rpass="loop|vect" -Rpass-missed="loop|vect" -Rpass-analysis="loop|vect"
remark: saxpy.cpp:8:3: vectorized loop (vectorization width: 8, interleaved count: 4) [-Rpass=loop-vectorize]
remark: saxpy.cpp:8:3: List vectorization was possible but not beneficial with cost 0 >= 0 [-Rpass-missed=slp-ve
remark: saxpy.cpp:8:3: preserved loop [-Rpass=loop-delete]
remark: saxpy.cpp:8:3: unrolled loop by a factor of 4 with run-time trip count [-Rpass=loop-unroll]
remark: saxpy.cpp:8:3: loop not distributed: use -Rpass-analysis=loop-distribute for more info [-Rpass-missed=lo
remark: saxpy.cpp:8:3: vectorized loop (vectorization width: 8, interleaved count: 4) [-Rpass=loop-vectorize]
remark: saxpy.cpp:8:3: preserved loop [-Rpass=loop-delete]
```

Explicit vector data types

- Declare variables of vector data types and express the computations with normal arithmetic expressions
  - +,-,*,etc. are overloaded with the corresponding vector operations

```
#include <xmmintrin.h>
void saxpy(int n, float alpha, float * __restrict X, float * __restrict Y) {
  const __m128 * __restrict x = (__m128*)__builtin_assume_aligned (X, 32);
  __m128 * __restrict y = (__m128 *)__builtin_assume_aligned (Y, 32);
  for (int i=0; i<n/4; i++)
    y[i] = alpha*x[i] + y[i];
}
```

- Note that the number of operations in the loop is now n/4
  - x[i] and y[i] are now vectors of 4 floating-point values
  - Alpha is a scalar



Using compiler intrinsics

- Functions for performing vector operations on packed data
  - Implemented as functions which call the corresponding vector instructions
  - Implemented with inline assembly code
- Vectorised programming with intrinsics is very low-level
  - Also non-portable...
- Operate on vector data types __mm128, __mm256
- Often used for vector operations that can not be expressed as normal arithmetic operations
  - Loading/storing, shuffle, masking...

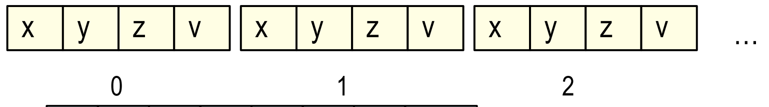
SAXPY with vector intrinsics

```
void saxpy(int n, float alpha, float *X, float *Y) {
  __m128 x_vec, y_vec, a_vec, res_vec; /* Vector variables */
  a_vec = _mm_set1_ps(alpha);
  for (int i=0; i<n; i+=4) { /* Vector of 4 alpha values */
    x_vec = _mm_load_ps(&X[i]); /* Load 4 values from X */
    y_vec = _mm_load_ps(&Y[i]); /* Load 4 values from y */
    /* Compute */
    res_vec = _mm_add_ps(_mm_mul_ps(a_vec, x_vec), y_vec);
    _mm_store_ps(&Y[i], res_vec); /* Store the result */
  }
}
```

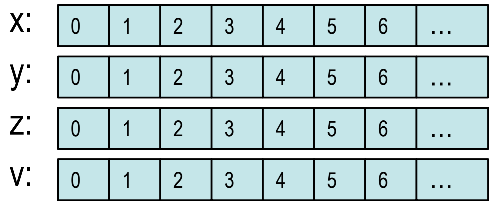
- Declare vector variables of appropriate type
- Load data values into the variables
- Do arithmetic operations by calling intrinsics
- Load/store require alignment
  - Unaligned: _mm_loadu_ps

Arranging data for vector operations

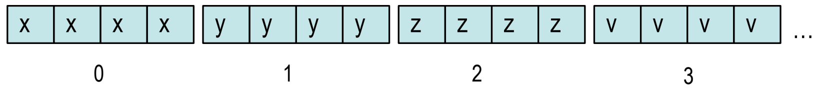
- It is important to organize data in memory so it can be accessed as vectors
  - Consider a structure with four elements: x,y,z,v
- Array of structures



- Structure of arrays



- Hybrid structure:



- Rearranging data in memory for vector operation is called data swizzling: there are shuffle intrinsics

Portability

- Explicitly vectorised code is not portable
  - Only runs on the architectures which support them
  - Sometimes not even portable across compilers
- Can use conditional compilation in the code
  - Program contains both scalar and vectorised versions

```
#ifdef __SSE2__
   // SSE2 version of the code
#else
  // Normal scalar version of the code
#endif
```

- Can use CPU dispatch: multiple versions, runtime detection (but no portability across compilers)

Exercise

- Take our matrix-matrix multiplication code (from last class)
- Determine memory access patterns of the innermost loop
  - Does it vectorise? (see if it reports vectorized for the given line number)
- Can we rearrange computations so it will vectorise?
  - With k as innermost loop we have data dependency across *k* iterations (since the same value in matrix3 is written)
  - Do you get the same result if you swap the *k* and *j* loops?
- How many floating point operations per second?
- Optional excercise: create a bar chart with achieved floating point operations per second with the vectorizing and non-vectorizing variants at different matrix sizes
- Running with OpenMP:

```
OMP_PROC_BIND=TRUE OMP_NUM_THREADS=xx ./matmat
```

Measuring parallel performance

- Speedup is helpful for comparing to the theoretical metrics
    - Time a parallel region, and see how close speedup is to the number of threads
- But it can also be very deceiving
    - Is it really a fair comparison? If you significantly rewrote/optimised your algorithm to get it running in parallel, then perhaps not
    - All too often, as you increase the number of threads, the speedup doesn’t follow – is something wrong with your code?
    - Taking a poorly written algorithm and making it run in parallel can look great, but is it actually efficiently making use of the hardware?
- On any given machine, there is a finite set of resources – question is, how efficiently are we using them, and which one becomes the bottleneck?

Examples

```
#pragma omp parallel for
for (j=0; j<STREAM_ARRAY_SIZE; j++)
         a[j] = b[j]+scalar*c[j];
```

- Take for example a classical benchmark (STREAM Triad, 10 million)

1 thread	2 threads	4 threads	8 threads
0.018692 sec	0.017075 sec	0.017179 sec	0.017270 sec
1x	1.09x	1.08x	1.08x

- Compare it to the matrix-matrix multiply example (2048^2)

1 thread	2 threads	4 threads	8 threads
3.346 sec	1.743 sec	1.062 sec	0.873 sec
1x	1.91x	3.15x	3.83x

```
cc -Ofast stream.c -o stream -fopenmp -DSTREAM_ARRAY_SIZE=50000000
```

Computations & Data Movement

- It all comes down to moving data and computing on data
  - Intel Core i7-6700K – can do 8 single precision fused-multiply-add operations (2 ops) per core per clock cycle -> 2*8*4*4.2 (GHz) = 268 GFLOPs (Giga floating operations per second). Has a 34 GB/s bandwidth to DDR4.
    - 31 operations per byte
  - NVIDIA GTX 4090 – can do 82000 single precision GFLOPs, has a 1008 GB/s bandwidth to GDDR6X
    - 325 operations per byte
- The cost of data movement is enormous
  - 200-600 cycles (vs 1 cycle for compute), 100x in terms of power
  - Fortunately we have on-chip caches which are ~10x faster and more efficient – but they are small

Examples

- Coming back to the vector add example, we see that at each iteration, we are moving 3 values, and performing 2 operations (1 add, 1 mul)
  - We are not using those values again
  - A ratio of 0.6 – this algorithm is limited by data movement
  - The key bottleneck (resource) is the amount of available bandwidth – around 18 GB/s on my laptop

1 thread	2 threads	4 threads	8 threads
0.017679 sec	0.015957 sec	0.015065 sec	0.015125 sec
1x	1.11x	1.17x	1.16x
13.5 GB/s	15.04 GB/s	15.9 GB/s	15.8 GB/s

  - A single thread can almost fully utilize this resource – on larger machines you will need more threads, but usually less than the number of cores

Examples

- The matrix-matrix multiplication on the on the other hand moves 3*N^2 values and performs 2*N^3 operations
  - Assuming of course that each value is only moved once and then re-used N times
  - We have a ratio of up to 1.5N – the algorithm is limited by computational throughput
  - The key bottleneck (resource) is the amount of available computational throughput (up to 73 Gflops/s on my laptop)

1 thread	2 threads	4 threads	8 threads
3.346 sec	1.743 sec	1.062 sec	0.873 sec
1x	1.91x	3.15x	3.83x
5.1 Gflops/s	9.8 Gflops/s	16.2 Gflops/s	19.7 Gflops/s

  - Our naïve implementation is not great, but still compute limited

Data movement and communication

- Latency: how long does a single operation take?
  - Measured in cycles or microseconds
- Bandwidth: What data rate can be sustained?
  - Measured in MBytes or Gbytes per seconds

- These terms can be applied to
  - Memory access
  - Messaging

The Cache

- The Cache is integrated onto the CPU
  - fast but small memory
  - used to store a small part of main memory for faster access
  - typically useful when you access the same small part of main memory multiple times
- CPU has different strategies for which small part of main memory to store in the Cache
  - Tries to be clever and predict what you don’t need anymore and what you will need soon
- Different kinds of cache – proximity to the core & size
  - L1, L2, L3

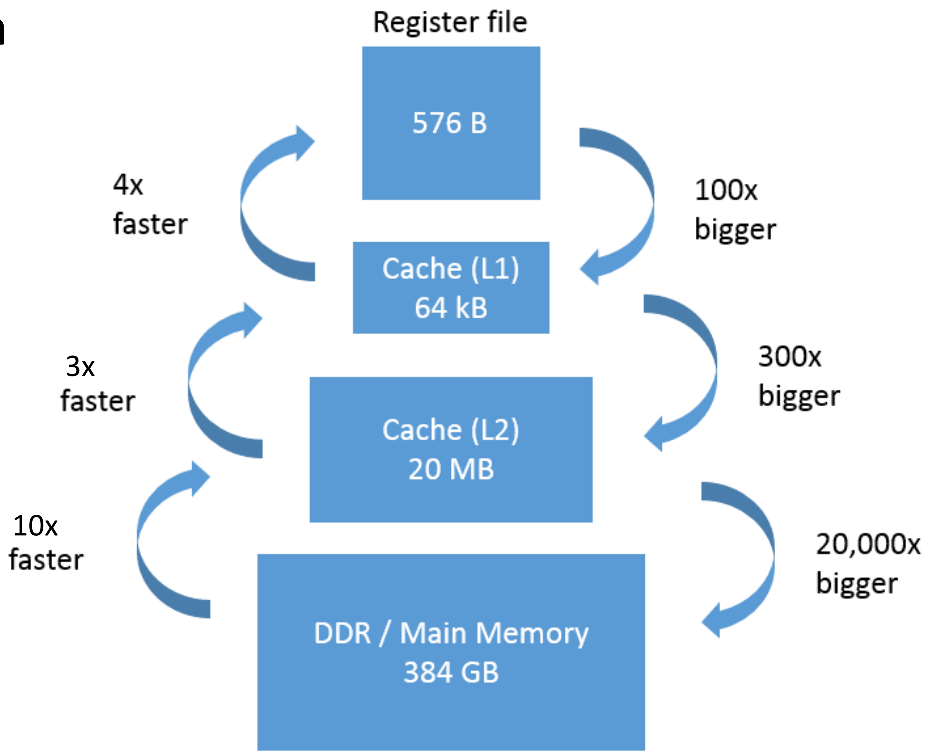
A memory hierarchy

Bandwidth

900 GB/s

300 GB/s

30 GB/s



Latency

1 cycle (0.5 nsec)

4 cycles (2 nsec)

65 cycles (33 nsec)

~120 cycles (60 nsec)

Memory bandwidth

- Limited by both:
  - the bandwidth of the memory bus
  - the bandwidth of the memory modules
- Can be improved by increasing the size of memory blocks
- Memory system takes l time units to deliver b units of data
  - l is the latency of the system
  - B is the block size

Reusing data in the memory hierarchy

- Spatial reuse: using more than one “word” in a multi-word line
  - Unit of transfer is a cache line: 64B
  - Using multiple values on a cache line
- Temporal reuse: using a “word” repeatedly
  - Accessing the same word in a cache line more than once

- Think about matrix-matrix multiply!

Cache line and fetch utilisation

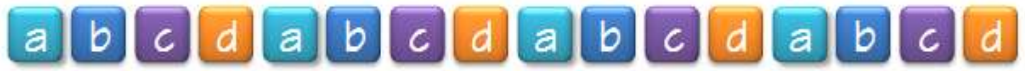
Program A

```
struct DATA
{
    int a;
    int b;
    int c;
    int d;
};
DATA * pMyData;

for (long i=0; i<10*1024*1024; i++)
{
    pMyData[i].a = pMyData[i].b;
}
```

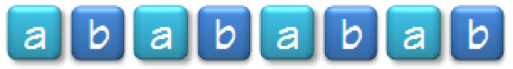
Program B

```
struct DATA
{
    int a;
    int b;
};


DATA * pMyData;

for (long i=0; i<10*1024*1024; i++)
{
    pMyData[i].a = pMyData[i].b;
}
```

- Compare the two versions: cachetest.cpp
- Why do they not run at the same speed?
- The unit of loading is a cache line: 64 bytes



A full 64 byte cache line for Program A. Every block represents a 32 bit integer.



A full 64 byte cache line for Program B. Every block represents a 32 bit integer.

Struct alignment

- Any data has to be aligned to a multiple of its size
  - E.g. an int has to be 32-bit aligned, a double 64-bit, etc.

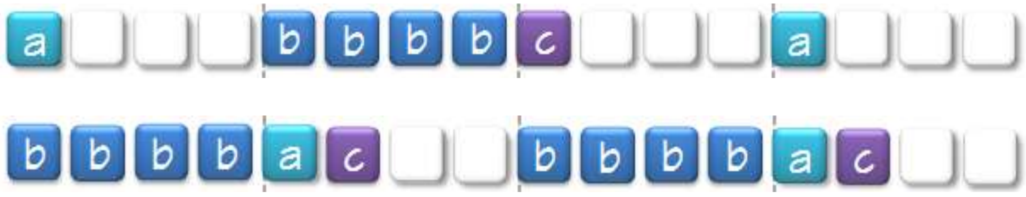
Program C

```
struct DATA
{
    char a;
    int b;
    char c;
};

DATA * pMyData;

for (long i=0; i<36*1024*1024; i++)
{
    pMyData[i].a++;
}
```

Program D

```
struct DATA
{
    int b;
    char a;
    char c;
};

DATA * pMyData;

for (long i=0; i<36*1024*1024; i++)
{
    pMyData[i].a++;
}
```

- Take a look at aligned.cpp – why the size of the struct?



Strided accesses

Program E

```
char * p;

p = new char[SIZE];

for (long x=0; x<sRowSize; x++)
for (long y=0; y<nbRows; y++)
{
    p[x+y*sRowSize]++;
}
```

Program F

```
char * p;

p = new char[SIZE];

for (long y=0; y<nbRows; y++)
for (long x=0; x<sRowSize; x++)
{
    p[x+y*sRowSize]++;
}
```

- Take a look at strided.cpp – why the performance difference?
  - We write the same data
  - Just not in the same order
- Try program E with different nbRows (keeping total size the same)
- Check cache accesses, hits & misses:

```
valgrind --tool=cachegrind ./strided
```

Memory System Performance

- Exploiting spatial and temporal locality is critical for
  - Amortizing memory latency
  - Increasing effective memory bandwidth
- Ratio #operations / #memory accesses
  - Good indicator of anticipated tolerance to memory bandwidth
- Memory layout and computation organisation significantly affect spatial and temporal locality

Hiding latency with multithreading

- A thread is a single stream of control in the flow of a program
- Take e.g. the matrix-vector multiply

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

- Each dot product (row) is independent of each other
- Can rewrite with OpenMP:

```
#pragma omp parallel for
for (int i = 0; i < n; i++)
  for (int j = 0; j < n; j++)
    c[j] += a[i][j]*b[i];
```

On a single core

- Consider how the code would execute on a single CPU core, with two threads
  - First thread accesses a pair of data elements, and waits for them
  - Second thread can access two other data elements in the next CPU cycle
- After l units of time (l is the latency of the memory)
  - First thread gets its data and performs its multiply-add
- Next cycle
  - Data for the second thread arrives, performs its multiply add
- Then next loads, etc...
- If we have l threads, then for every cycle, we can perform a computation

Latency hiding

- Two major assumptions:
  - Memory system can service multiple outstanding requests
  - Processor can switch between threads at every clock cycle
- Really far from the truth on CPUs
  - Requires e.g. replicated register files
  - Intel has 2 threads per core, IBM 8
- GPUs do almost exactly this: up to ~28000 threads active

Vector type library

- There is an open source C++ vector types library
  - Take a look (vectorclass.cpp): https://www.agner.org/optimize/vectorclass.pdf
  - Use the Vec8f (float) or Vec4d (double) classes, the load and store methods, and horizontal_add methods to implement the saxpy example and matrix-matrix multiplication
    To compile: g++ -mavx2 -mfma -Ofast

```
// Simple vector class example C++ file
#include <stdio.h>
#include <vectorclass/vectorclass.h>
int main() {
 // define and initialize integer
 //vectors a and b Vec4i
 a(10,11,12,13); Vec4i
 b(20,21,22,23); // add the two vectors
 Vec4i c = a + b; // Print the results
 for (int i = 0; i < c.size(); i++) {
  printf(" %5i", c[i]);
 }
 printf("\n");
 return 0;
}
```

Homework due March 24th midnight

- Time each method, and estimate the achieved bandwidth

```
for (int j = 0; j < NY; j++) {
  for (int i = 0; i < NX; i++) {
    for (int f = 0; f < 9; f++) {
      workArray[(j*NX+i)*9 + f] = workArray[(j*NX+i)*9 + f] - 1.5 * (ux[j*NX+i]*ux[j*NX+i] +
uy[j*NX+i]*uy[j*NX+i]);
    }
  }
}
```

bytes moved: (NX)(NY)*9**
((write workArray = 8 bytes)+(read workArray = 8 bytes))+
(NX)(NY)*((read ux = 8 bytes)+(read uy = 4 bytes))*

Scalars neglected
Multiple accesses to same array neglected

- **Output at the very end of execution**:

```
Name              Count    Time        GB/s
kernel1             10       0.2s      35
kernel2             7        1.3s      22
...
```