Lecture 19: Parallel Programming using SIMD Vector Units

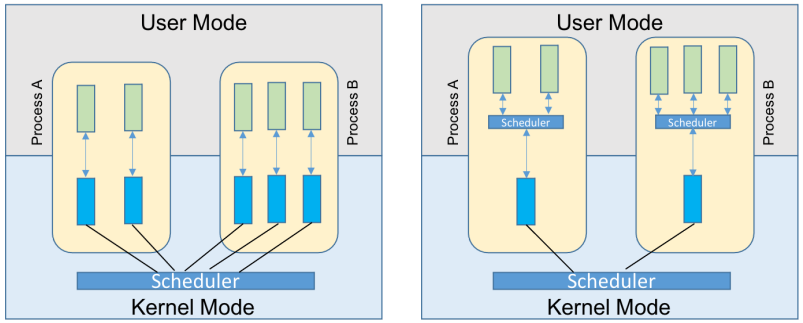
Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture (Recap)

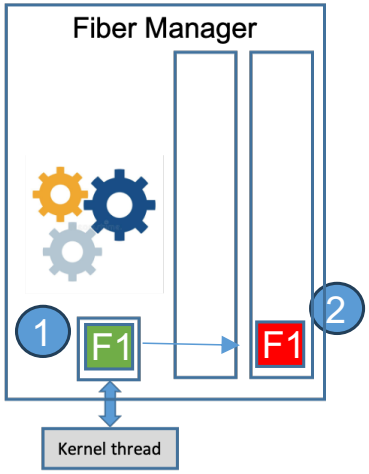


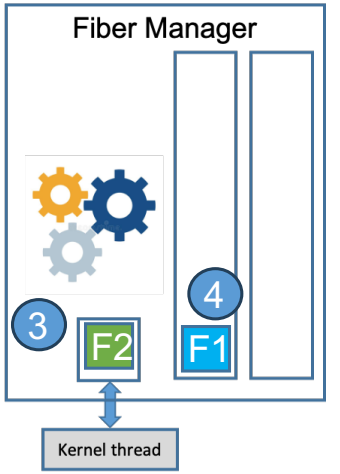

```
std::mutex mtx;
std::condition_variable cnd;
std::string str;

boost::fibers::fiber f1([=]() {                    1
  std::unique_lock<std::mutex> lck(mtx);
  if(str.size() == 0) {
    cnd.wait(lck);                                 2
  }
  cout << str << endl;
});

boost::fibers::fiber f2([=]() {                    3
  std::unique_lock<std::mutex> lck(mtx);
  str = "Hello Fiber";
  cnd.notify_one();
});                                                4

f1.join();
f2.join();
```



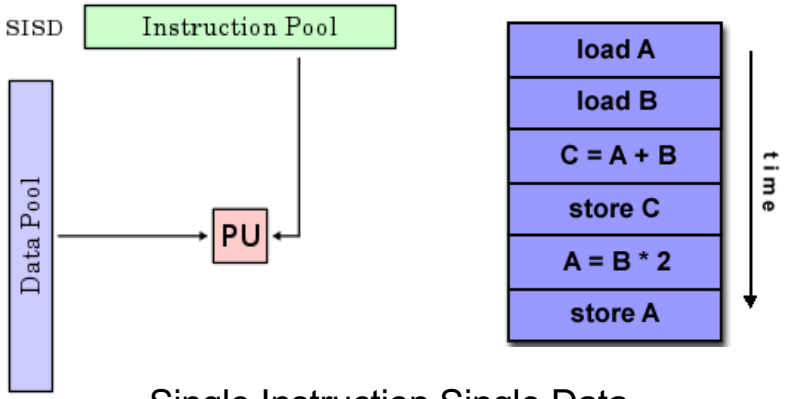


- ULT v/s KLT
- Boost fibers
  - Emulates std::thread operations, but as a ULT instead of a KLT
    - Based on boost context

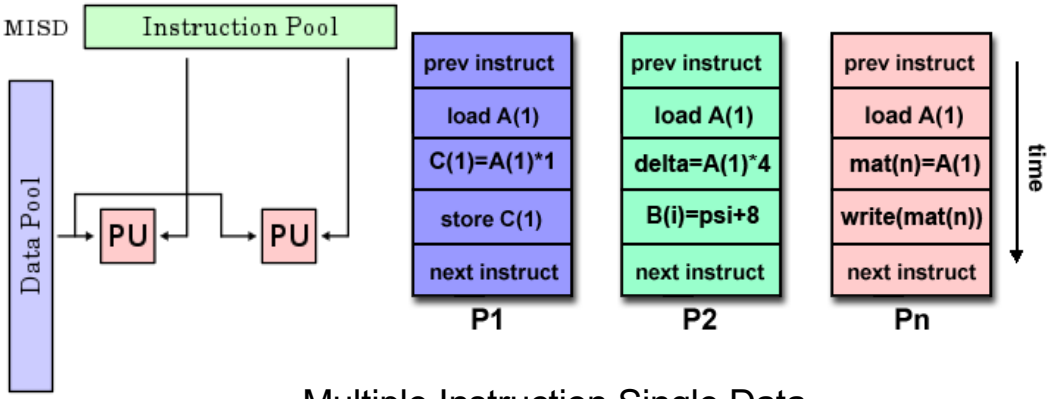
Today's Class

- Flynn's classification
- SIMD vector extensions
- SIMD programming techniques
- Limitations of vectorization
- Vector Class Library for SIMD programming

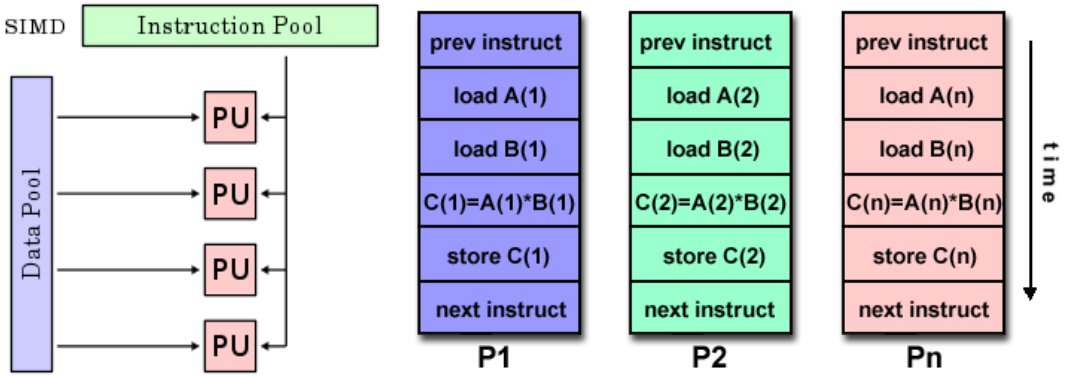
Flynn's Classification of Parallel Computer



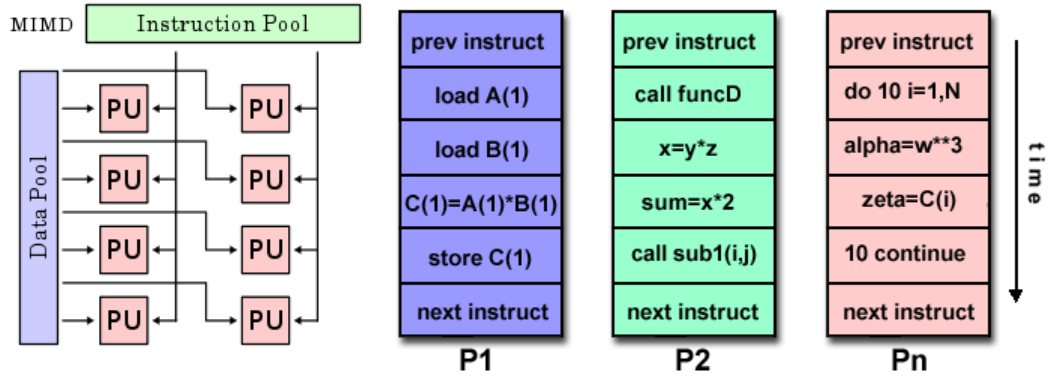
Single Instruction Single Data



Multiple Instruction Single Data

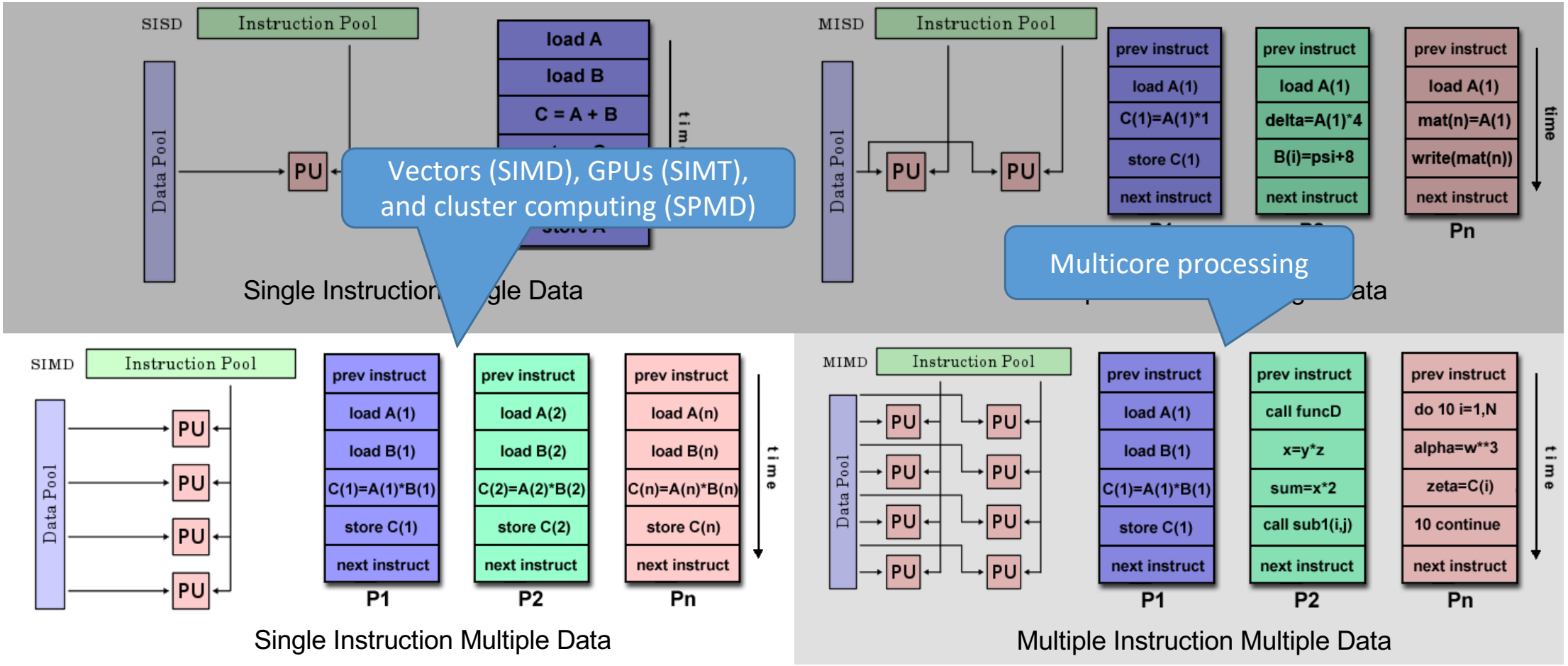


Single Instruction Multiple Data



Multiple Instruction Multiple Data

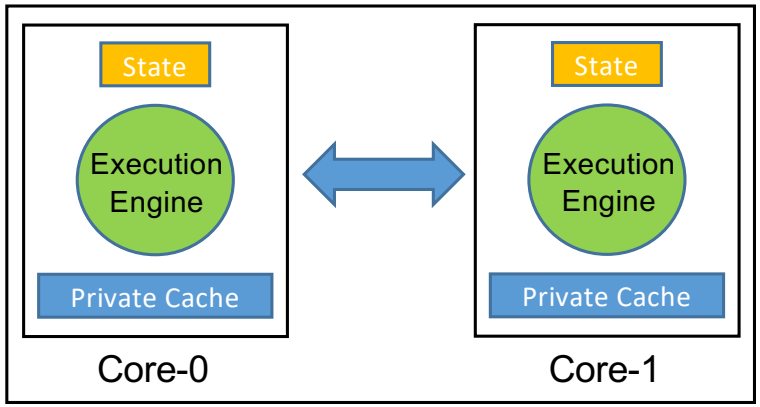
Flynn's Classification of Parallel Computer



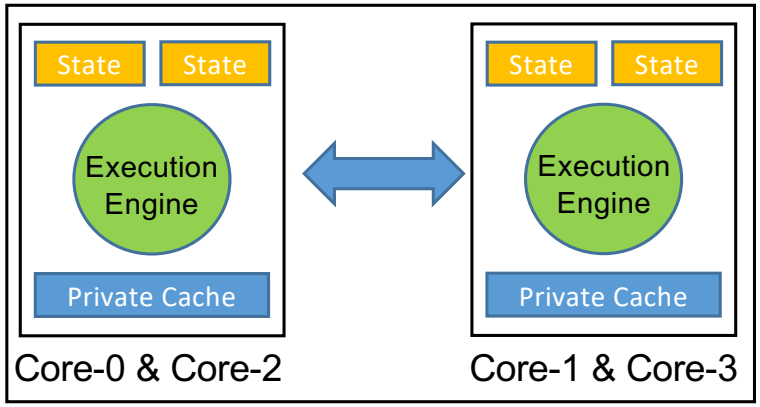
Exploiting Parallelism on Modern Processors

- Modern processors supports three different kinds of parallelism
  - Instruction level parallelism
    - Done automatically by the hardware
  - Thread (Task) level parallelism (multicore)
    - Achieved by the help of compiler/programmer
  - Vector (Data) level parallelism
    - Achieved either by the help of compiler (automatic) or by the programmer (manual)
      - **-O3** switch in gcc triggers automatic vectorization wherever possible

Physical VS. Logical Cores



Dual-core processor with hyperthreading **DISABLED**



Dual-core processor with hyperthreading **ENABLED**

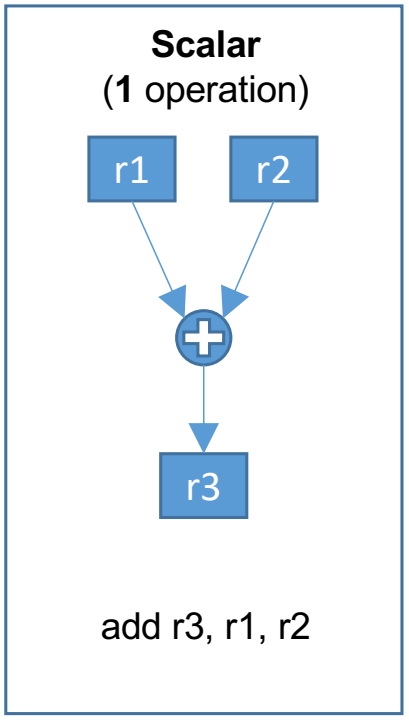
- Architectural state of a core are the registers (EBP, ESP, EIP, etc.)
- Logical cores of a processor share
  - Private cache
  - Execution engine
  - System bus interface
- If the execution of one of the logical core blocks (e.g., Core-0 waiting for a memory fetch from the DRAM) then the other logical core (Core-2) can resume its execution with its own state

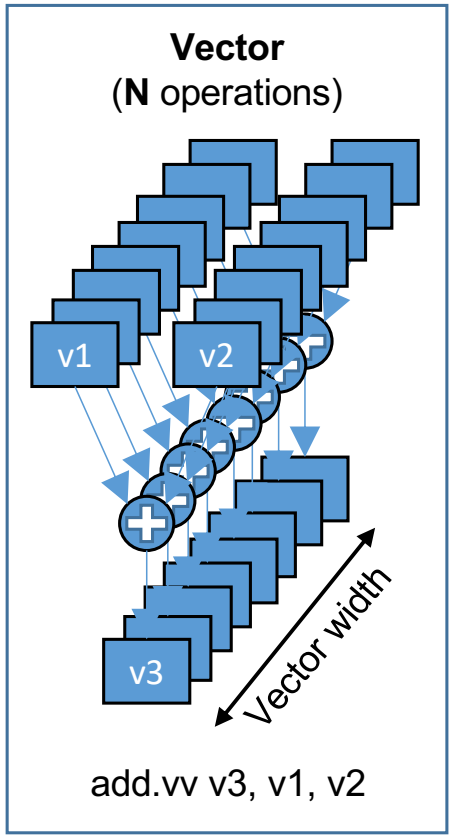
Today's Class

- Flynn's classification
- SIMD vector extensions
- SIMD programming techniques
- Limitations of vectorization
- Vector Class Library for SIMD programming

SIMD Vector Extensions

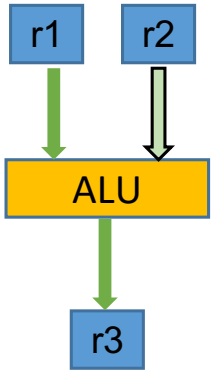
- What it is?
  - Extension of the ISA
  - Special registers that support instructions to operate upon **vectors** than **scalar** values
    - **Each core has its own SIMD execution units**
    - Parallel computation on short (length 2, 4, 8..) vectors of integers or floats
  - Names: SSE, SSE2, AVX, AVX2, AVX512, etc.
- Where do they exists?
  - On almost all modern processor, e.g., Intel & AMD
- What is their usage?
  - Free data parallelism units capable of providing (theoretical) speedups equal to the **vector width**
    - Single instruction operates on multiple data elements simultaneously



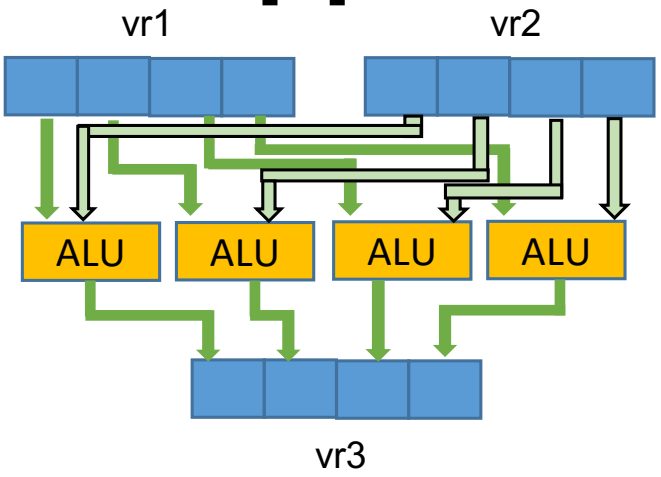


Picture source: Prof. Patterson's Lecture on vector processing

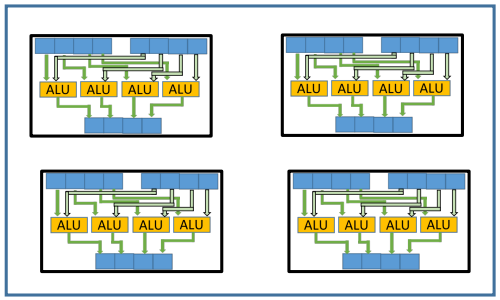
Architectural Support for SIMD



SISD operation on scalars



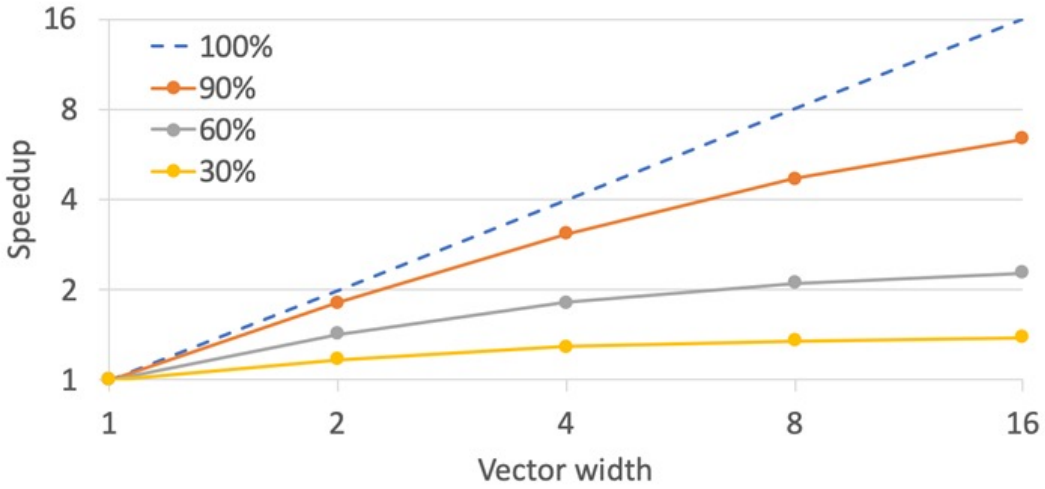
SIMD operation on vectors



Multicore processor supporting SIMD operations

- SIMD operation is supported on processors by adding more ALUs to each core, and by using wide registers (greater than 32 bit)
  - Thanks to Moore's law that small size transistors leave ample space for adding other functionalities
- Each CPU cycle can now operate on more than one 32-bit value
- Increasing vector register width require adding new instructions

Amdahl's Law for Vectorized Code

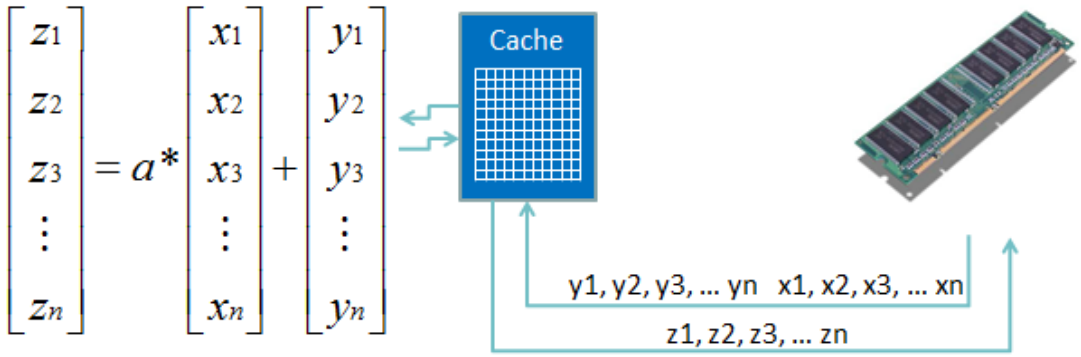


- Assume some work takes “W” time on a scalar CPU
- Time taken on a CPU with vector width “N” for total vectorized fraction “f” available in that work
  - $Time_{scalar} + Time_{vector}$ => (1-f)W + fW/N
- Hence, maximum possible speedup
  - W / {(1-f)W + fW/N} => 1 / { (1-f) + f/N }

Picture source: https://cvw.cac.cornell.edu/vector/performance_amdahl

- Linear speedup is possible only for perfectly parallel code
- The exact upper bound depends significantly on the percentage of code that is vectorized
  - At a vector width of 16, code that is 60% vectorized performs only twice as fast as non-vectorized code
- Sequential or scalar code would limit the performance
  - What about memory access pattern?

Memory Access Pattern Affects Performance



- Moving data into and out of vector registers involves several levels of the memory hierarchy
- Make use of temporal and spatial locality for getting best performance
  - True for all kinds of parallelization

Picture source: https://cvw.cac.cornell.edu/vector/performance_memory

History of SIMD Vector Support in Intel Chips

Year Released	Name	Register Width (BIT)	Width (Float)
1996	MMX (Multimedia Extension)	64	**2**
1999	SSE (Streaming SIMD Extension)	128	**4**
	SSE2	128	4
	SSE3	128	4
	SSE4	128	4
2011	AVX (Advanced Vector Extensions)	256	**8**
	AVX2	256	8
2013	AVX-512	512	**16**

- Every new generation of SSE or AVX supports new and improved set of instructions
- Backward compatibility with every new generation

Source: https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

Check Supported SIMD Instructions

- Use the following command to check the SIMD instructions supported by your processor

```
$ cat /proc/cpuinfo | grep flag | tail -1
```

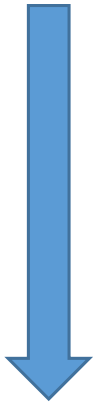
```
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2
ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc
arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc
cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx
smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1
sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave
avx f16c rdrand lahf_lm abm cpuid_fault epb
invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi
flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2 smep
bmi2 erms invpcid xsaveopt dtherm ida arat pln pts
md_clear flush_l1d
```

Today's Class

- Flynn's classification
- SIMD vector extensions
- SIMD programming techniques
- Limitations of vectorization
- Vector Class Library for SIMD programming

SIMD Programming Techniques

- Applied either at compile time or link-time
  - Compiler based auto-vectorization
  - Compiler pragmas (e.g., OpenMP simd)
  - Calls to Vector Class Library (VCL)
  - Hand coded compiler intrinsic
  - Inline assembly code



Easy to use, but low performance

Ninja Level, but best performance

Compiler Perspective

- Vectorization is similar to loop unrolling
  - Unroll by “N” iterations, where “N” is vector width

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

```
for (i=0; i<N; i+=4) {
    c[i+0] = a[i+0] + b[i+0];
    c[i+1] = a[i+1] + b[i+1];
    c[i+2] = a[i+2] + b[i+2];
    c[i+3] = a[i+3] + b[i+3];
}
```

```
for (i=0; i<N; i+=4) {
    Load a[i+0 : i+3] in v0;
    Load b[i+0 : i+3] in v1;
    v0 = v1 + v0;
    Store v0 in c[i+0 : i+3];
}
```

Vector width = 128 bit

- How to inform the compiler for using vectorization?
  - Intel compiler starts vectorization with -O2 optimization flag
  - GCC compiler starts vectorization with -O3 optimization flag

Today's Class

- Flynn's classification
- SIMD vector extensions
- SIMD programming techniques
- Limitations of vectorization
- Vector Class Library for SIMD programming

General Limitations of Vectorization (1/6)

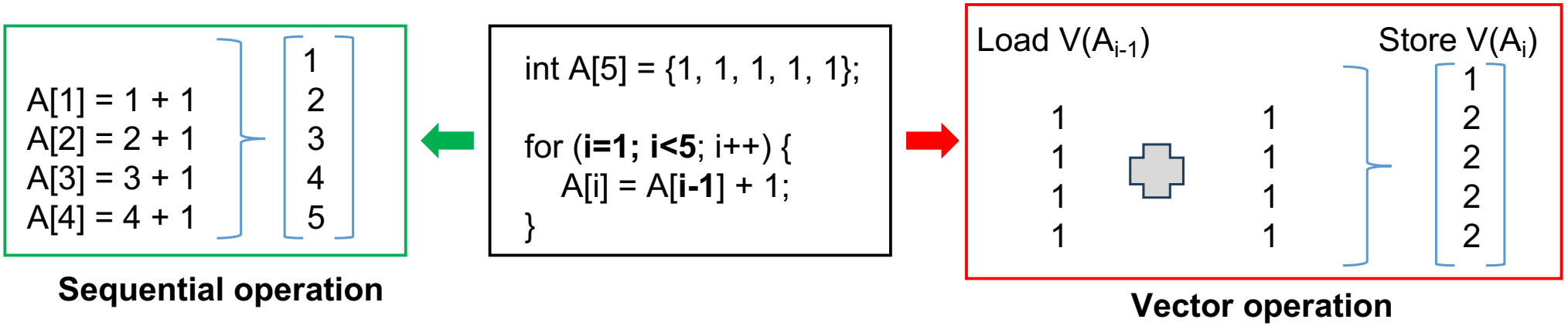
- Loop size should be countable at runtime
  - Loop size is not required at compile time, but it should not change during execution (runtime)
    - Implies single entry and single exit for the loop (no break statements)
- Loop iterations should not have different control flow
  - ”if” or “switch” statements cannot be used for selective calculation of data elements
    - Although, ”if” or “switch” statements may be used as masked statements, i.e., calculation is performed for all elements, but result is stored selectively

    for (i=0; i<N; i++) { int s = B[i] + C[i]; if (s>10) A[i] = s; else A[i] = 0; }
- Should not have non-contiguous memory accesses

  for (i=0; i<N; **i+=2**) Scalar_A[i] = Scalar_ B[i] + Scalar_ C[i];
- Early exits should not be used (break, and continue statements)
- Loop iterations should be independent, e.g., a[b[i]] not allowed
- Loop should only use basic math functions, e.g., pow, sqrt,..

General Limitations of Vectorization (2/6)

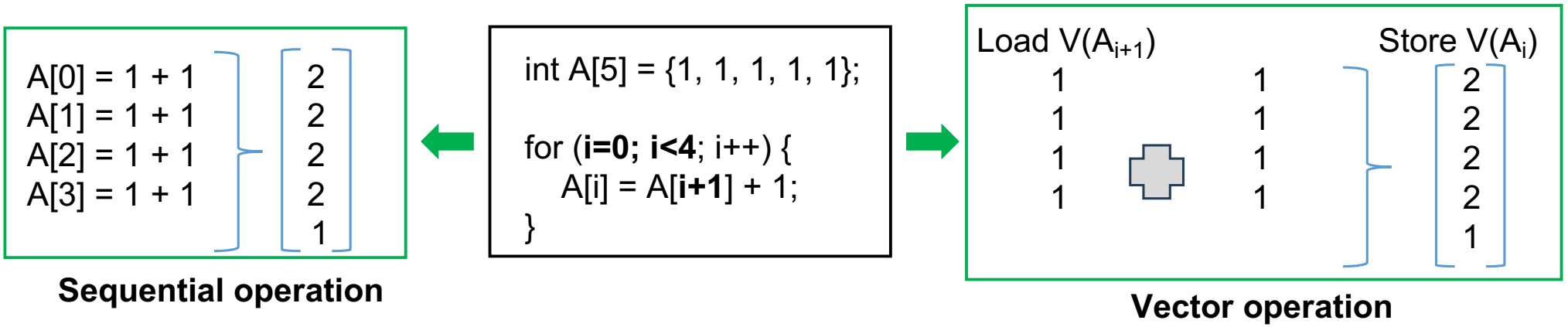
- Data dependency (1/4)
  - Read-After-Write (RAW) or Flow Dependency
    - Happens when a variable being **written** in one iteration is being **read** in the next iterations



    - Not vectorizable
    - Is it parallelizable using the traditional multicore parallel_for?

General Limitations of Vectorization (3/6)

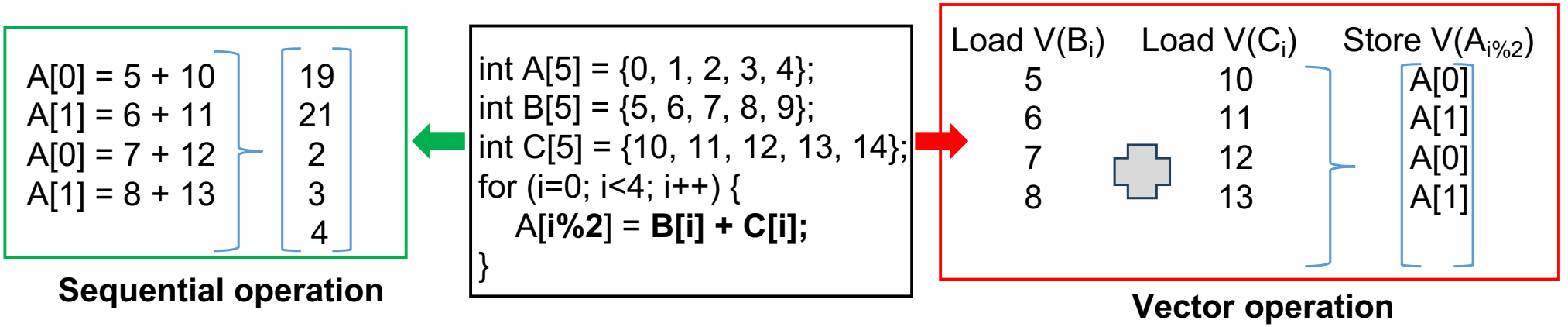
- Data dependency (2/4)
  - Write-After-Read (WAR) or Anti Dependency
    - Happens when a variable being **read** in one iteration is being **written** in the next iterations



    - Totally safe for vectorization
    - Is it parallelizable using the traditional multicore parallel_for?

General Limitations of Vectorization (4/6)

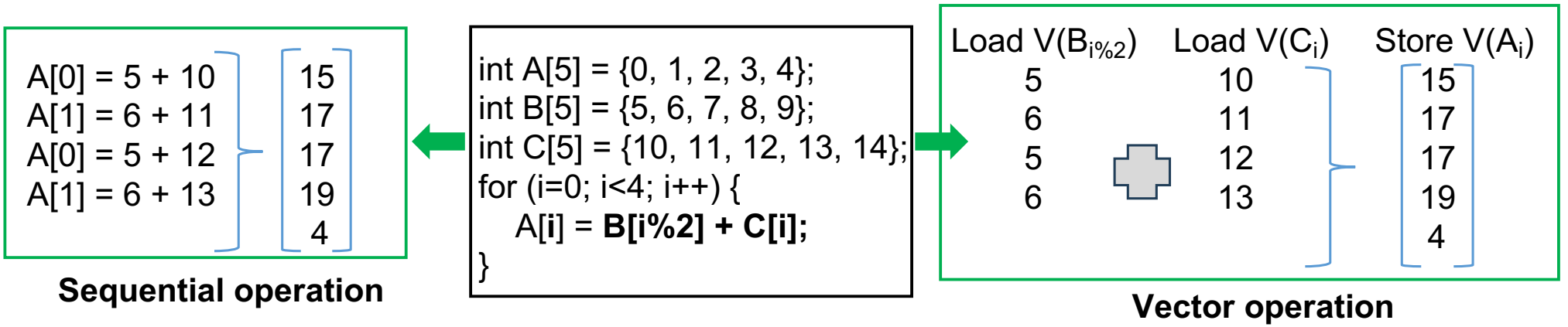
- Data dependency (3/4)
  - Write-After-Write (WAW) or output dependency
    - Happens when same variable is **written** in more than one iterations



- Not vectorizable (unpredictable result)
- Is it parallelizable using the traditional multicore parallel_for?

General Limitations of Vectorization (5/6)

- Data dependency (4/4)
  - Read-After-Read (RAR)
    - Happens when same variable is **read** in more than one iterations



- Totally safe for vectorization
- Is it parallelizable using the traditional multicore parallel_for?

General Limitations of Vectorization (6/6)

- Pointer aliasing
  - Aliasing refers to a situation where two different expressions or symbols refer to the same object
  - Pointer aliasing may lead to data dependencies

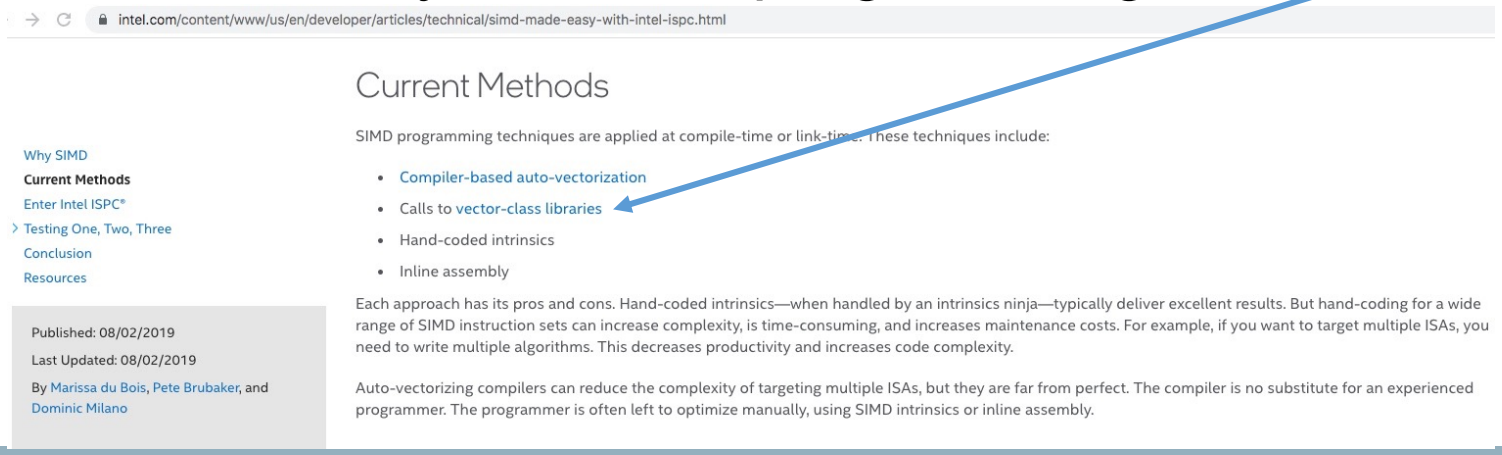
```
void compute(int*A, int*B) {
    for (i=1; i<5; i++) {
        A[i] = B[i] + 1;
    }
}
……
compute (A, A-1);
```

```
A[1] = A[0] + 1
A[2] = A[1] + 1
A[3] = A[2] + 1
A[4] = A[3] + 1
```

Pointer aliasing causing Read-After-Write (RAW) data dependency

Today's Class

- Flynn's classification
- SIMD vector extensions
- SIMD programming techniques
- Limitations of vectorization
- Vector Class Library for SIMD programming



Programming using Vector Class Library (VCL)

- C++17 library for writing vector code without using assembly language or compiler intrinsic
- **Header only implementation**, i.e., no installation required
- Programmer can use appropriate width vector class, and compile with native compiler (GNU, Clang, Intel icc, etc.)
  - Compiler flag used to specify the desired SIMD instruction set (SSE4, AVX2, AVX512, etc.)
    - Must be supported by the processor
- Supported on Windows, Linux, and Mac, 32-bit and 64-bit, with Intel, AMD, etc.

VCL Usage

1. Constructing vectors using VCL
   - Vec4i a;
   - Vec4i a(5);
   - Vec4i a = 6;
   - Vec4i a(1, 4, 9, 0);
   - Vec4i a
2. Loading data into vectors
   - Vec4i a(0);
     a.insert(/*index*/ 2, /*value*/ 9);
   - Vec4i a;
     a.load(array + index);
3. Getting data from vectors
   - Vec4i a(1, 4, 9, 0); int array [SIZE]
     a.store(array + index);
   - Vec4i a(1, 4, 9, 0);
     int element_index2 = a[2];
4. Arithmetic operations on vectors
   - +, -, *, /, ++, +=, -=, *-, /=, /*many more*/
5. Logical operations on vectors
   - ==, !=, >, <, <=, >=, /*many more*/
6. Functions operating on single vectors
   - horizontal_add, horizontal_min, horizontal_max, /*many more*/
7. Function operating on two vectors
   - min, max, abs, /*many more*/

Vector Addition using VCL

```
int A[1024], B[1024], C[1024];

void sum() {
  for (int i=0; i<1024; i++) {
    A[i] = B[i] + C[i];
  }
}
```

128 bit vector

```
#include "vectorclass.h"
int A[1024], B[1024], C[1024];
void sum() {
  Vec4i Av;
  for (int i=0; i<1024; i+=4) {
    Vec4i Bv = Vec4i().load(B+i);
    Vec4i Cv = Vec4i().load(C+i);
    Av = Bv + Cv;
    Av.store(A+i);
  }
}
```

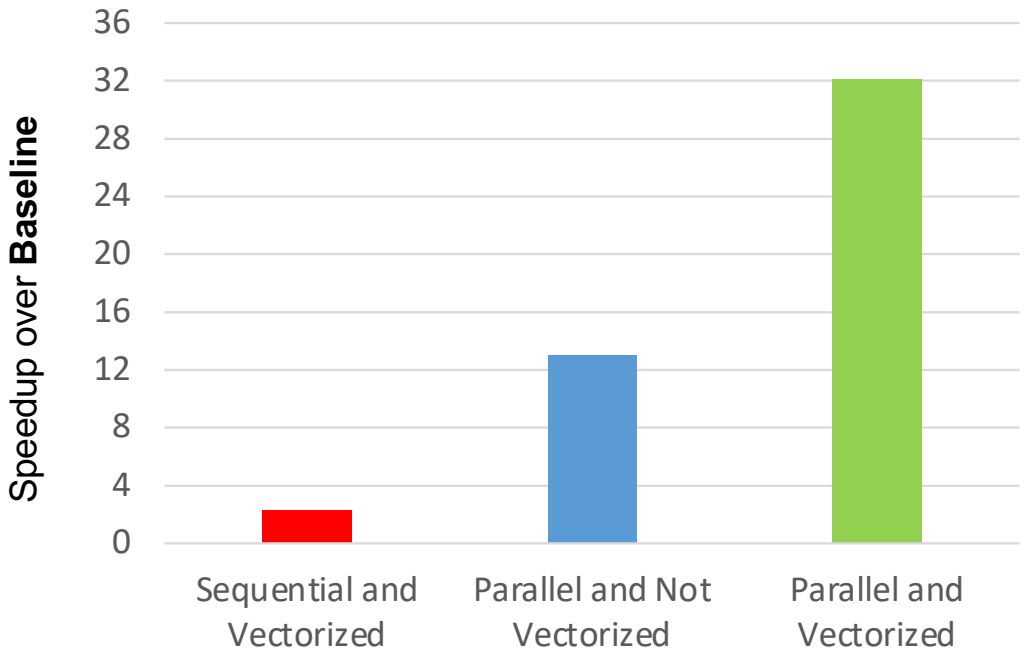
256 bit vector

```
#include "vectorclass.h"
int A[1024], B[1024], C[1024];
void sum() {
  Vec8i Av;
  for (int i=0; i<1024; i+=8) {
    Vec8i Bv = Vec8i().load(B+i);
    Vec8i Cv = Vec8i().load(C+i);
    Av = Bv + Cv;
    Av.store(A+i);
  }
}
```

- VCL program compilation

  g++ -std=c++17 -O3 -msse4 -fopt-info-vec -I/path_to/VCL/version2 sum.cpp

Performance Benefits using SIMD



- Four different variants of matrix multiplication of size 1024x1024 of floats
  1. Sequential (**Baseline**)
  2. Sequential but using vectorization with Vec8f
  3. Recursive task parallelism but without vectorization (20 threads)
  4. Recursive task parallelism where leaf tasks are using vectorization with Vec8f (20 threads)
- Two sockets of 10-core Intel Xeon E5-2650 v3 processor
  - Total 20 cores
- GNU compiler 7.5.0 using -O3 and SSE4 instruction set
- VCL version2 commit id 08959eb
- Ubuntu 16.04.7 LTS

Code available in CSE513 GitHub repo:
https://github.com/hipec/cse513/blob/main/lec19/tests/par_matmul.cpp

Reference Materials

- Intel guide for auto vectorization
  - https://www.intel.com/content/dam/www/public/us/en/documents/guides/compiler-auto-vectorization-guide.pdf
- Cornell virtual workshop for vectorization
  - https://cvw.cac.cornell.edu/vector/
- VCL: Vector Class Library
  - https://www.agner.org/optimize/vcl_manual.pdf
  - https://github.com/vectorclass/version2

Next Lecture

- GPU programming