TACC
TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

# Introduction to Manycore Processors and Programming

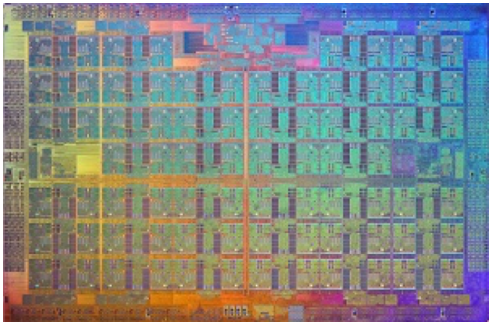
Jerome Vienne
viennej@tacc.utexas.edu

February 14, 2018

# What is a Multicore or Manycore Processor?

## Multicore & Manycore Definition

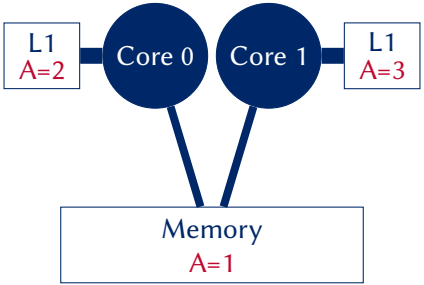
- A processor with multiple execution units (cores)
- Cores are capable of executing different instructions simultaneously
- Multicore vs. Manycore: no formal distinction, $\sim 10s$ vs. $\sim 100s$



## Cores on a processor share resources

- May or may not share caches
- May or may not share memory

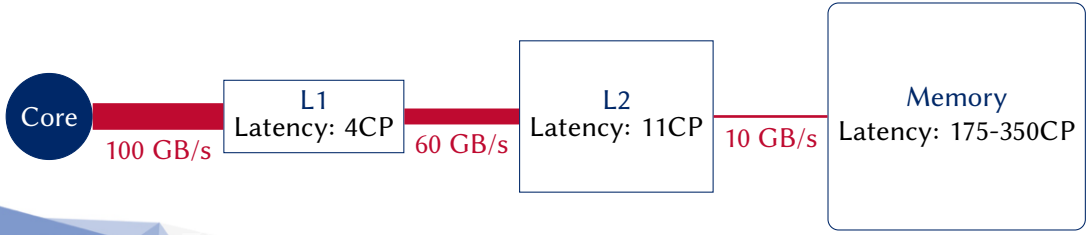
## Cache and Memory Coherence is maintained by hardware and OS in some way



# Aside: Why use caches?

## Memory Hierarchy

- Caches are smaller but faster (higher bandwidth, less latency) than memory
- May have multiple levels of cache: L1, L2 . . .
- Helps keep cores fed with data
- Cache reuse is often critical to performance



# Why are Multicore & Manycore Necessary?

## During 50 years of Moore's Law: transistor density $\times 2$ every 2 years

- transistors grew smaller
- smaller transistors require less voltage and current
  - can switch at higher frequency at same power ($P \propto ACfV^2$)
  - Clock rates increased
  - Capabilities could be added/expanded at constant power
    - Instruction-level parallelism (pipelining)
    - OO executions
    - prefetching
    - SIMD units
    - more cache
- single processor performance $\times 2$ every 18 months

# Why are Multicore & Manycore Necessary?

## Physics limits single processor performance . . .

- We can still make smaller transistors for a few more years
  - more expensive every year
  - transistors cannot be smaller than atoms
- Power no longer decreases with transistor size
  - $P \propto \underbrace{ACfV^2}_{dynamic} + \underbrace{VI_{leakage}}_{static}$
  - Voltage cannot continue decreasing because:
    - $f \propto (V - V_{th})^{\alpha}/V$, where $\alpha > 1$ and $I_{leakage} \propto e^{-V_{th}/T}$

# Why are Multicore & Manycore Necessary?

## Power per transistor cannot be reduced . . .

- more transistors packed in same area $\rightarrow$ higher power density $\rightarrow$ more heat
  - faster processors get too hot ($\sim$ 171 F at 3.8 GHz), cooling gets bulky & expensive
  - thermal runaway: currents leak $\rightarrow$ hotter chip $\rightarrow$ worse leakage
- speed has not really increased since 2009 ($\sim$4GHz is practical max)
- Adding lower frequency cores keeps computational capacity increasing at comparable power

# Multicore & Manycore Drawbacks

## Performance does not typically scale linearly with # cores

- Even for perfectly parallel work
- Cores must communicate and share resources
- Cache and memory coherency require additional logic
  - More logic steps = longer latency
  - Prefetching and out-of-order execution can hide latency
    - complex, expensive, and improvement has slowed
  - Coherency traffic uses up system bandwidth, less bandwidth to move data
- Memory bandwidth per core is decreased $BW/\#cores < BW/1$
- Individual cores are slower to conserve power
  - serial codes may run slower on newer processors!

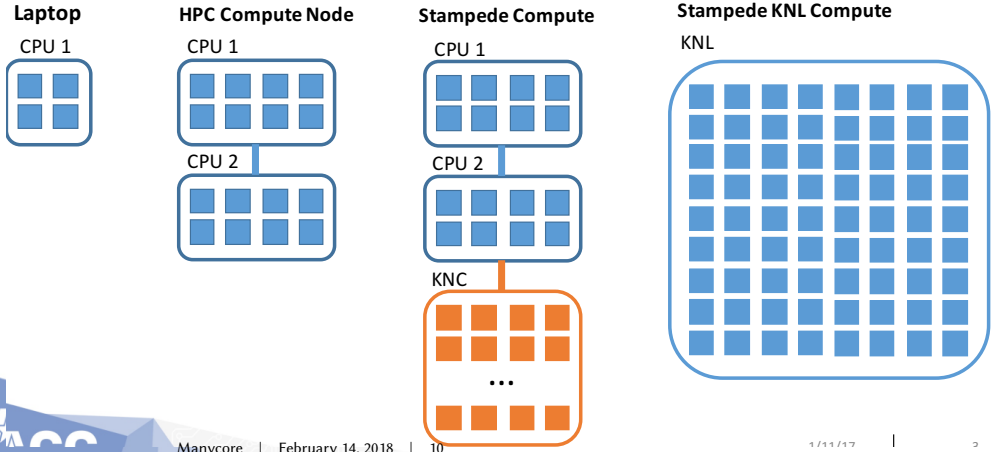
# Why Manycore over Multicore?

## Logical Extension to Multicore

- 100s relatively slow (1.5GHz) processors vs 10s relatively fast (2.5GHz) processors
  - Manycore processors run at lower frequency than multicore to constrain
- Overall more compute available for comparable power
- Manycore specialized for parallel computation
- Multicore is more general purpose

# In HPC Performance ~ Parallelism

## From Laptop to Next-Gen Supercomputer



# Impact of Manycore on Software Development

## If you want to run faster you may have to refactor code

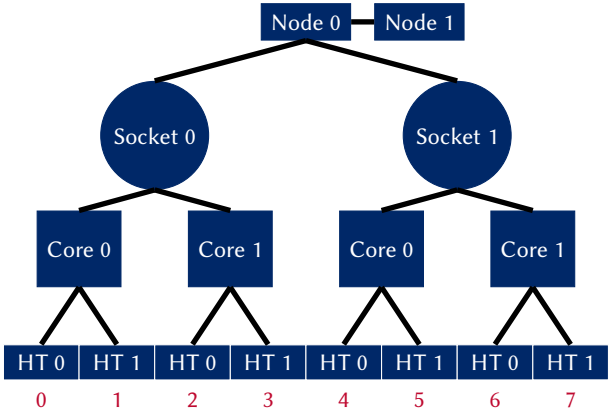
- Some types of algorithms will run slower on manycore
- Optimal serial code/algorithm not always parallelizable
- Use predictable and linear code and data in compute sensitive kernels
  - Less pointer chasing
  - Less Object-oriented techniques
  - Less branches and conditionals
- Parallelism is required for performance!

# Levels of Parallelism in Computing

## Largest to Smallest

- Multinode: multiple compute nodes connected by network
  - compute nodes have independent processors and memory
- Multisocket: multiple sockets per node
  - sockets usually have one processor and one memory each
  - mostly transparent to the user
- Multi/Manycore: Multiple cores per processor
  - cores share memory and may share cache
- Hardware threads: Multiple “logical cores” per physical processor
  - Logical cores share physical core
  - One logical core executes on physical core at a time
  - If one logical core stalls another can immediately execute

# Levels of Parallelism




Multinode

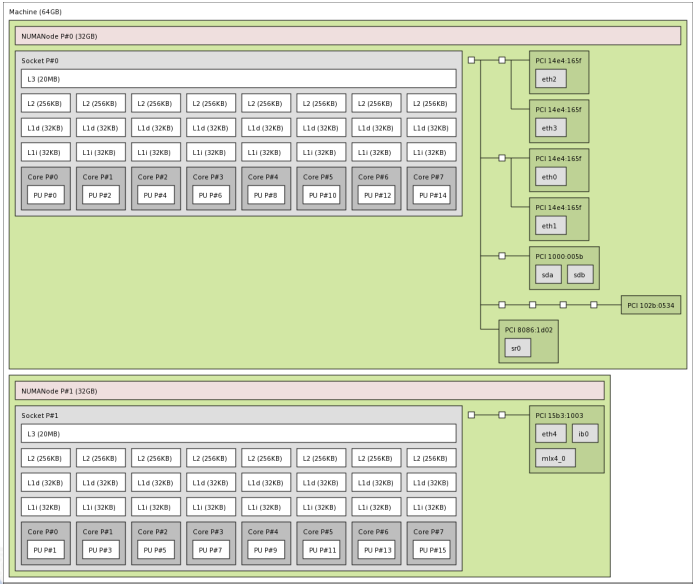
Multisocket

Multicore

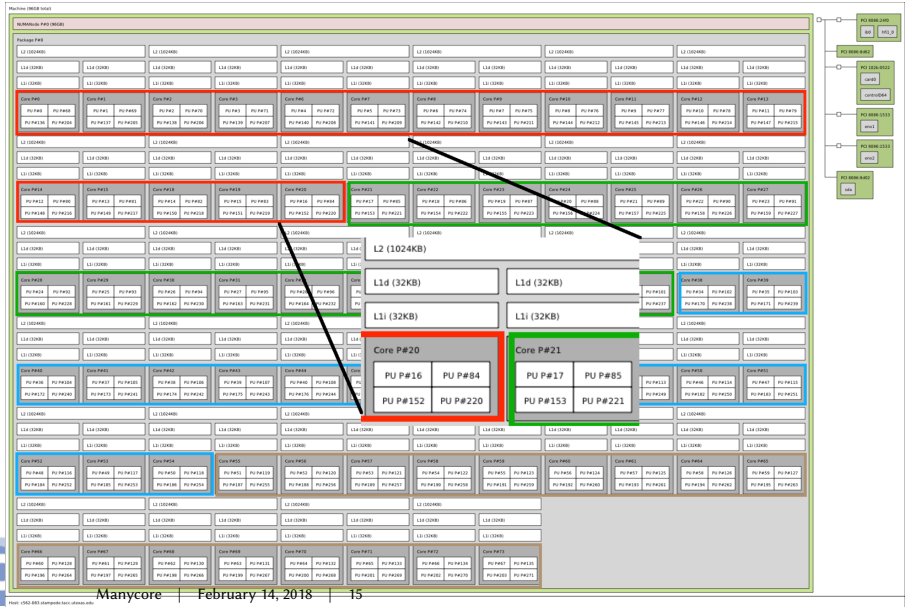
Hardware Threading

Logical Core #

# Multicore Processor Layout: `lstopo`



# Manycore Processor Layout: `lstopo`



# Multitasking

## Basic Concepts

- Linux works on *tasks*, tasks are units of execution
  - *processes* are tasks with private memory/data by default (e.g. programs)
  - *threads* are tasks with shared memory/data by default (e.g. functions)
- Multiprocessing: processes executing in parallel
- Multithreading: threads executing in parallel
- Core & memory *affinity*: associate tasks to cores and memory
- Contention for shared resources: limited memory/cache BW and limited cores
- Dependencies: parallel tasks may be interdependent
  - restricts concurrency

# How-to run Tasks in Parallel

## Many ways in Linux

- Run multiple processes simultaneously from command line or within script
  - Example: `./program1 & ./program2 & ... ./programN`
  - & places processes in background and allows others to start
- fork/multiprocessing programming
  - Start multiple processes intra-node
- pthreads/OpenMP programming
  - Starts multiple threads intra-node
- MPI
  - Starts multiple processes inter-/intra-node
- Which cores do the tasks run on?

# Scheduling

## Linux has a *Scheduler*

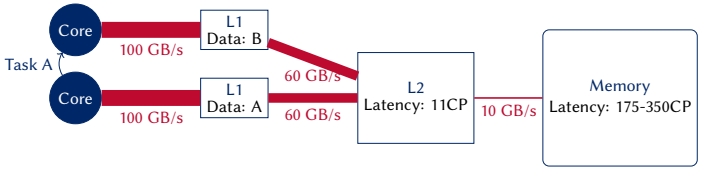
- Linux Scheduler treats processes and threads the same: tasks
- Tries to give every task a "fair" share of processor's time
- Each core has its own runqueue
- Runnable tasks are in a core's runqueue
  1. task at head of queue is run
  2. runs either for max time ($\sim T_{wait}/\#tasks$), finishes, or is interrupted
  3. A new task runs on core (context switch)
  4. Old task inserted back into runqueue if unfinished
- tasks are started on and moved ("float") around cores by load balancer

# Load Balancing

## Linux Scheduler has a Load Balancer that runs periodically

- Load balancer tries to maximize processor & cache utilization
- Tasks are assigned to least busy cores
- Tasks can be moved from more busy to less busy cores
- Load balancer tries to keep tasks close to data though
  - Maintains *data locality*

# Load Balancing: Data Locality



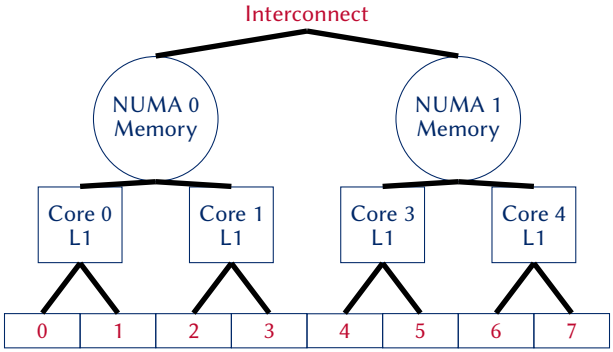

## When task is moved to new core, local cache or memory may not have data

- Data must be transferred to new core somehow: forwarded, over shared cache (slower), through memory (slowest)
- When tasks move to new socket, local memory will not have required data
  - Data must be transferred from memory over socket interconnect (%50 longer latency, limited bandwidth)
  - *First touch policy*: First socket task runs on allocates memory

# Load Balancing

## Load Balancing for data locality

- Sockets with different memory are labeled NUMA nodes in Linux
- Balance tasks across cores that share L1, then L2, then L3, then memory
- As last resort move task to different socket



# CPU & Memory Affinity

## Load balancer is not perfect

- Caches get thrashed due to *context switches*
- Tasks end up on wrong NUMA node

## Tasks can be *pinned* to sets of cores or NUMA nodes

- Each task has a mask of cores it may run on
- A pinned task will only float to allowed cores
- Many tools for this:
  - numactl
  - taskset
  - sched_setaffinity
  - `OMP_PROC_BIND` & `GOMP_AFFINITY` (GNU) & `KMP_AFFINITY` (Intel)
  - MPI implementations have their own affinity tools

# Multiprocessing & Multithreading Performance

## Measuring parallel performance

- *speedup*: runtime of parallel versus serial $\frac{T_s}{T_p}$
- *efficiency*: speedup versus number of tasks $\frac{T_s/\#tasks}{T_p}$
  - Typically less than 1
- *Granularity*: amount of work assigned to each task

# Multitasking Performance

## Affinity and Load Balancer will assign tasks to cores

- Once assigned to a core, how do we divide work among tasks?
- Say we have a problem that we break into 10 parallel parts:
  - we have 10 cores and 10 tasks (each task gets a part of the problem)
  - affinity or balancer places 1 task on each core
  - ensure all parts complete at similar time
  - otherwise some tasks (and cores) are idle
- This is *load balancing* from the programmer's perspective
- Want to minimize idle time for tasks
  - Maximizes usage of cores
- Programmer implements this in some way

# Load Balancing vs Overhead

## Load balancing will now refer to programmer's implementation

- *Granularity* is the amount of work per task
- It is important to balance granularity for parallel efficiency
- Too much granularity can lead to load imbalance
  - Some tasks could be idle for long periods, waiting for others to complete
- Too little granularity can lead to large overhead
  - Parallelizing not free
    - Partitioning problem
    - Assigning work to tasks
    - Communicating data between tasks
    - Synchronizing tasks

# Load Imbalance Example

## A loop with 10 iterations, divided among 9 threads

```
for(j=0;j<10;j++){...}
```

- Each iteration takes 1 thread 1 day
- 9 iterations are done the 1st day by 9 threads
- 1 iteration is done the 2nd day by 1 thread (9 threads are idle for a day :( )
- $efficiency = \frac{T_s/\#tasks}{T_p} = (10/9)/2 = 55\%$
- Negligible communication overhead but poor load balance

# Parallel Overhead Example

## A loop with 10 iterations, divided among 10 threads

```
for ( j =0; j <10; j ++){...}
```

- Setup of parallelization requires 1ms
- Each iteration takes 1 thread 0.1ms
- 10 iterations are done by 10 threads in 1 ms
- Total parallel time is 2ms $\rightarrow$ serial would have run in 1ms!
- *efficiency* $= \frac{T_s/\#tasks}{T_p} = (1/10)/2 = 5\%$
- *efficiency* $< 50\%$ slower than serial

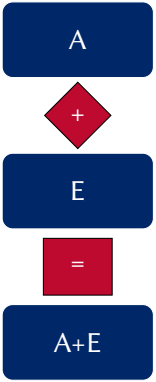
# Vectorization

yet another level of parallelism

# Vectorization

## Basics

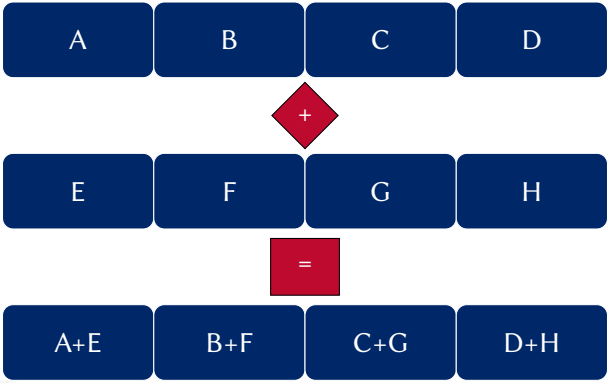
- Cores have registers
- Data must be moved from cache/memory into registers before operating on
- To add 2 integers:
  1. Move first integer A from cache to a register
  2. Move second integer E from cache to a different register
  3. Add integers and place result in yet a different register
  4. Move result to cache/memory
- Frequencies are limited so can only go so fast
- Why not move and add several integers simultaneously?
  - SIMD: Single Instruction Multiple Data



# Vectorization

## Make registers wider → vector registers

- Same motivations as multicore: more compute at comparable power
- Increase # computations by increasing number of operations per cycle



# Evolution of SIMD Hardware

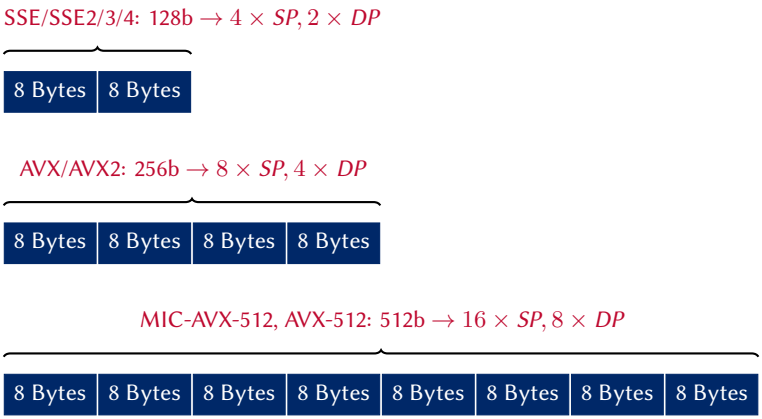
**Vector length has recently increased by a factor of 4!**

| Year | Width | ISA Name | Data (register names) |
|---|---|---|---|
| 1997 | 64 | MMX | Integer |
| 1999 | 128 | SSE1 | SP FP SIMD (xMM0-8) |
| 2001 | 128 | SSE2 | DP FP SIMD(xMM0-8) |
| — | 128 | SSE3, SSE4 | ··· |
| 2010 | 256 | AVX | DP FP SIMD (yMM0-16) |
| 2013 | 256 | AVX2 | ··· |
| 2016 | 512 | MIC-AVX512, AVX512 | DP FP SIMD (zMM0-32) |

*SP = Single Precision Number (32 bits)*
*DP = Double Precision Number (64 bits)*

# Vector Registers

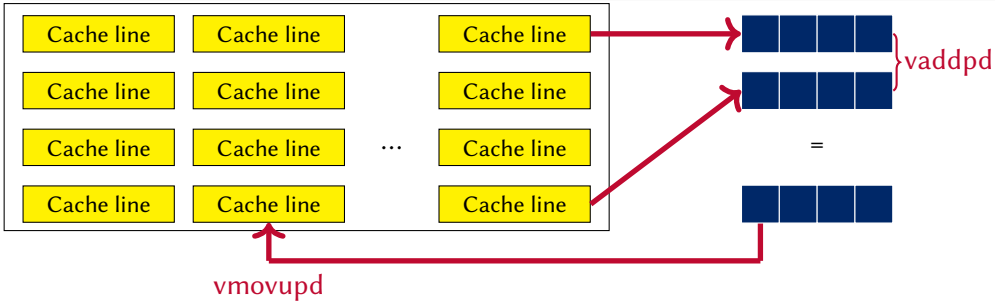
- Different types of processors have different width vector registers
- Register width is in bits
- SP/DP are 32b/64b (4B/8B)
- Correct vector instructions are required to use registers



# Vector Instructions

## Vectorized code uses vector instructions

- Vector instructions act on multiple data elements
- Vector instructions exist for moving data and operating on data



# Vectorization in Code

## for loop *before* vectorization

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

## for loop *after* vectorization (vector length of 4)

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

# Ideal Speed-up due to Vectorization

## Assume Perfectly Vectorized

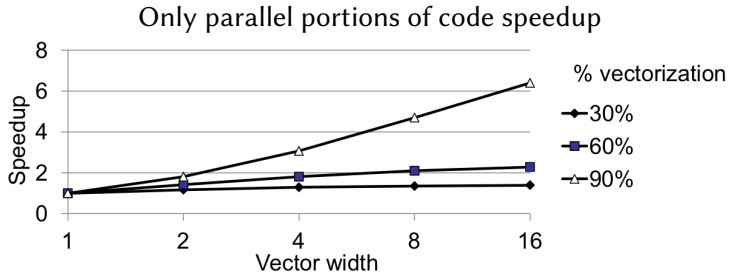
- Speed-up $\propto$ Vector width
  - 128b SSE: $\times 2$DP
  - 256b AVX: $\times 4$DP
  - 512b AVX-512: $\times 8$DP
- Calculate theoretical peak for Stampede KNL
  - $68 cores \times 2 \frac{VPU}{core} \times \frac{8 FLOPs}{VPU\ cycle} \times \frac{1.4\ Gcycles}{s} = 1523.2\ GFLOPS$
  - Fused-Add multiply $\times 2$
  - Theoretical KNL peak $\sim$ 3 TFLOPS

# Deviations from Ideal (Multitasking too!!!)

## Due to Bandwidth

- cache utilization
- memory latency
- alignment

## Due to Amdahls law: $T_{total} = T_{parallel}/p + T_{serial}$



# How-to Vectorize Your Program

## Approaches

- Assembly (Bad)
  - Error prone, probably won't be that fast, not very portable
- Instrinsics (Better)
  - Less error prone, probably not the fastest possible, still not portable
- Compiler Flags (Good)
  - This is often the best idea!
  - Portable
  - Automatically vectorizes your code
- Link to optimized, vectorized libraries if available (Very good!)
  - Teams of scientists and engineers have put years into optimizing these
  - Portable
  - MKL

# How-to Vectorize Your Program Automatically

## Use compiler flags

- Intel 17.x+:
  - `-O2` (SEE2), `-xavx/-xcore-avx2/-xCore-AVX512/-xmic-avx512/-xCommon-AVX512`
  - `-xhost` (chooses most advanced host has)
  - "fat-binaries" using `-xCORE-AVX2 -axCORE-AVX512,MIC-AVX512`
- GCC and LLVM:
  - `-O3` (SSE2), `-mavx/-mavx2/{-mavx512 -mavx512cd -mavx512er -mavx512pf}/-mfma -mavx512f -mavx512cd`
  - `-march=native/knl/skylake-avx512`

# Vector Aware Programming

1. Most vectorization opportunities are in loops
2. Keep loops simple to help compiler vectorize
3. Evaluate compiler output (tools exist for this)
4. Evaluate performance improvement after vectorization
5. Consider Memory Access Patterns (unit stride is critical)
6. Vector Math Libraries should be used for special functions
7. MKL or equivalent should be used for linear algebra

# 3 General Rules for Optimization

## Applies to Serial, Vectorized, and Multithreaded Code

1. Make compute intensive parts of code clean
   - 1.1 Compiler likes linear, predictable data structures
   - 1.2 Save OO for staging and managing workflow
   - 1.3 Compiler will do most of the work for you
2. Use optimization options (flags)
   - 2.1 tell compiler how hard to work
   - 2.2 tell compiler what instructions are allowed
3. Use math libraries (lapack/blas, MKL)
   - 3.1 These may thread automagically for you
   - 3.2 Teams of engineers have developed these over decades
   - 3.3 Specialized for particular architectures (e.g. KNL)

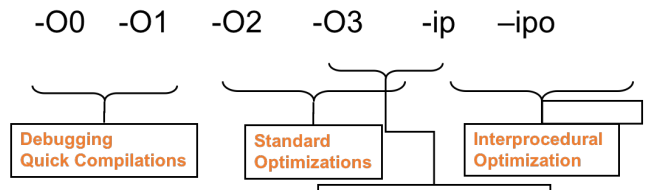
# Compiler Options (Flags)

**Three important categories**

- Optimization Level
- Interprocedural Optimization
- Architecture Specification

It's often a good idea to set at least one option from each category

# Compiler Optimization Levels




-O0: Debug
-O1: Fast compile
-O2: Standard
-O3: More Aggressive

# Interprocedural Optimization Level

## IPO allows compiler to optimize across routine boundaries

- Remove function call overhead
- Reveals serial optimization opportunities
- Reveals vectorization opportunites

| | Intel | GNU |
|---|---|---|
| single file | -ip | -finline-functions |
| multi-file | -ipo | -flto |

# Interprocedural Optimization

## Before Inlining

```
double r2(x,x0,y,y0) {
    return (x-x0)*(x-x0)+(y-y0)*(y-y0)
}

int main(){
...
    double d=r2(a,a0,b,b0);
...
return 0;
}
```

## After Inlining

```
int main(){
...
    double d=(x-x0)*(x-x0)+(y-y0)*(y-y0)
...
return 0;
}
```

Single File = Function definition is in same file as function call

Multi-file = Function definition is in different file from function call

# Architecture Specification

We already discussed these - typically related to vectorization capabilities of processor
Different Architecture have different instructions sets

| Stampede | Lonestar 5 | Stampede KNL |
|---|---|---|
| Sandybridge | Haswell | Knights Landing |
| AVX | AVX2 | MIC-AVX512 |
| | FMA | FMA |
| 4 FLOPs/cycle | 8 FLOPs/cycle. | 16 FLOPs/cycle |

# Questions?