Programming Intel® Xeon Phi™

An Overview

Anup Zope

Mississippi State University

20 March 2018

Outline

Background

Single Core to Multi-Core to Many-Core

- Single core
  - processor computational capacity improved through
    - ★ instruction pipelining
    - ★ out-of-order engine
    - ★ sophisticated and larger cache
    - ★ frequency scaling
  - Major computational capacity improvement was due to frequency scaling.
  - But faced limitations due to added power consumption from frequency scaling.
  - This motivated the shift to multi-core processors.
- Multi-core
  - Computational capacity improvement is due to multiple cores.
  - Sophisticated cores give good serial performance.
  - Additionally, parallelism provides higher aggregate computational throughput.
- Many-core
  - Computational capacity improvement is due to large number of cores.
  - When large number of cores are requires, they need to be simple due to chip area limitations.

Parallel Programming Paradigms

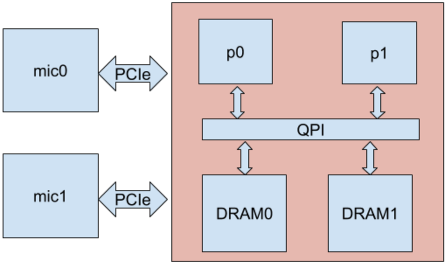
- Distributed memory computing
  - Multiple processes, each with separate memory space, located on the same and/or multiple computers.
  - A process is fundamental work unit.
  - a.k.a. MPI programming in HPC community.
  - Suitable when working set size exceeds a single computer DRAM capacity.
- Shared memory computing
  - Single process, with multiple threads that share memory space of the process.
  - A thread is fundamental work unit.
  - a.k.a. multihreading.
  - Suitable when working set fits in a single computer DRAM.

Xeon Phi Architecture

Shadow Node

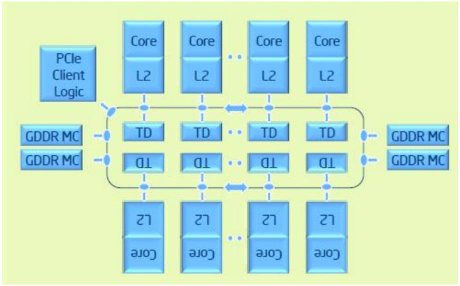
A node in HPC^2 shadow cluster

- Xeon (Ivy Bridge) as host
  - 1 chip per node
  - 2 processors on one chip
  - 10 cores per processor
  - 1 thread per core
  - NUMA architecture
  - 2.8 GHz
- Xeon Phi (Knight's Corner) as coprocessor
  - connected to host CPU over PCIe
  - 2 coprocessors per node
  - 60 cores per coprocessor
  - 4 threads per core



Xeon Phi^{TM} Architecture and Operating System

- 60 cores, 4 threads per core
- 32 KB L1I, 32 KB L1D shared by 4 threads
- L2 cache
  - 512 KB per core
  - Interconnected by ring
  - 30 MB Effective L2
  - Distributed tag directory for coherency
- SIMD capability
  - 512 bits vector units
  - 16 floats and 8 doubles per SIMD instruction
- 8 GB DRAM
- Runs Linux 2.6.38.8 with MPSS 3.4.1



Programming Xeon Phi^{TM}

Programming Xeon Phi™

- KNCNI instruction set:
  - Not backward compatible
  - Hence, unportable binaries
- Requires special compilation steps
  - using Intel 17 compiler and MPSS 3.4.1
- Two programming models
  - Offload model
    - ★ application runs on host with parts of it offloaded to Phi
    - ★ heterogeneous binary
    - ★ incurs the cost of PCI data transfer between host and coprocessor
  - Native model
    - ★ applications runs entirely on Phi
    - ★ no special code modification
    - ★ appropriate for performance measurement

Native Mode: Compilation

- Login to `shadow-login` or `shadow-devel`

```
ssh username@shadow-devel.hpc.msstate.edu
```

- Setup intel 17 compiler

```
swsetup intel-17
```

- Remove following paths from `LD_LIBRARY_PATH`
  - `/usr/lib64`
  - `/lib64`
  - `/lib`
  - `/usr/lib`
- Add following path to `PATH` for `micnativeloadex`

```
PATH=/cm/local/apps/intel-mic/3.4.1/bin:$PATH
```

- Compile using `-mmic` switch

```
icpc -mmic <other flags> sample.cpp
```

[1] Intel C++ 17 User Guide:
https://software.intel.com/en-us/intel-cplusplus-compiler-17.0-user-and-reference-guide

Native Mode: Execution

- Find dependencies and their path

```
micnativeloadex <binary> -l
```

- Execute the binary
  - Manually:

```
ssh mic0
export LD_LIBRARY_PATH=/lib:/lib64:/usr/lib64:/usr/
    local/intel-2017/compilers_and_libraries_2017
    .0.098/linux/compiler/lib/mic
./a.out
```

  - Using `micnativeloadex`:

```
export SINK_LD_LIBRARTY_PATH=/lib:/lib64:/usr/lib64
    :/usr/local/intel-2017/
    compilers_and_libraries_2017.0.098/linux/compiler
    /lib/mic
micnativeloadex ./a.out
```

[1]See: https://software.intel.com/en-us/articles/building-a-native-application-for-intel-xeon-phi-coprocessors/

Offload Mode

- a.k.a. heterogeneous programming model
- computationally intensive, highly parallel sections of the code need to be marked as offload regions
- decision to execute the offload regions on the coprocessor is made at runtime
    - if MIC device is missing, the offload sections run entirely on CPU
    - there is option to enforce failure if the coprocessor is unavailable
    - requires data copying between the host and device

Offload Mode: Marking Offload Code

- Offload regions

```
#pragma offload target(mic:target_id) \
 in(all_Vals : length(MAXSZ)) \
 inout(numEs) out(E_vals : length(MAXSZ))
for (k=0; k < MAXSZ; k++) {
  if ( all_Vals[k] % 2 == 0 ) {
    E_vals[numEs] = all_Vals[k];
    numEs++;
  }
}
```

Offload Mode: Marking Offload Code

- Offload functions and variables

```
__attribute__ (( target (mic))) int global = 55;

__attribute__ (( target (mic)))
int foo() {
  return ++global;
}

main() {
  int i;
  #pragma offload target(mic) in(global) out(i, global)
  {
    i = foo();
  }
  printf("global␣=␣%d,␣i␣=␣%d␣(should␣be␣the␣same)\n", global,
       i);
}
```

Offload Mode: Marking Offload Code

- Offload multiple functions and variables

```
#pragma offload_attribute (push,target(mic))
int global = 55;

int foo() {
  return ++global;
}
#pragma offload_attribute (pop)
```

Offload Mode: Managing Memory Allocation

- Automatic allocation and deallocation (default)

```
#pragma offload target(mic) in(p:length(100))
```

- Controlled allocation and deallocation

```
#pragma offload target(mic) in(p:length(100) alloc_if(1)
    free_if(0))
// allocate memory of p on entry to the offload
// do not free p when exiting offload

...

#pragma offload target(mic) in(p:length(100) alloc_if(0)
    free_if(0))
// reuse p on coprocessor, allocated in previous offload
// do not free p when exiting offload

...

#pragma offload target(mic) in(p:length(100) alloc_if(0)
    free_if(1))
// reuse p on coprocessor, allocated in earlier offload
// deallocate memory of p when exiting offload
```

Offload Mode: Target Specific Code

Offload compilation takes place in two passes - CPU compilation and MIC compilation. __MIC__ macro is defined in the MIC compilation pass.

```
#pragma offload_attribute(push,target(mic))
class MyClass {
#ifdef __MIC__
  // MIC specific definition of MyClass
#else
  // CPU specific definition of MyClass
#endif
};

void foo() {
#ifdef __MIC__
  // MIC specific implementation of foo()
#else
  // CPU specific implementation of foo()
#endif
}
#pragma offload_attribute(pop)
```

Offload Mode: Offload Specific Code

`__INTEL_OFFLOAD` macro is defined when compiling using `-qoffload` (on by default) and not defined when compiling using `-qno-offload`

```
__attribute__((target(mic))) void print() {
#ifdef __INTEL_OFFLOAD
  #ifdef __MIC__
    printf("Using␣offload␣compiler␣:␣␣Hello␣from␣the␣coprocessor\n"
        );
    fflush(0);
  #else
    printf("Using␣offload␣compiler␣:␣␣Hello␣from␣the␣CPU\n");
  #endif
#else
  printf("Using␣host␣compiler␣:␣␣Hello␣from␣the␣CPU\n");
#endif
}
```

Debugging

Debugging Xeon Phi^{TM}Application

- Two ways
  - using `gdb` on command line (native only)
  - using Eclipse IDE (native and offload)
- I will cover debugging of native applications remotely using `gdb-mic`.
- See https://software.intel.com/en-us/articles/debugging-intel-xeon-phi-applications-on-linux-host for more information.

Debugging Xeon Phi™ Native Application Remotely

This debugging is performed from host.

- Compile native debug binary using
  - -g option: generates debug symbols
  - -O0: disables all optimization
- Start and configure debugger

```
gdb-mic
target extended-remote | ssh -T mic0 OMP_NUM_THREADS=120
     LD_LIBRARY_PATH=/lib:/lib64:/usr/lib64:/usr/local/
    intel-2017/compilers_and_libraries_2017.0.098/linux/
    compiler/lib/mic /usr/bin/gdbserver --multi -
set sysroot /opt/mpss/3.4.1/sysroots/k1om-mpss-linux
file <path to debug binary>
set remote exec-file <path to debug binary>
set args <command line arguments of debug binary>
start
```

  Then use usual gdb commands to perform debugging.

- Look at `/opt/mpss/3.4.1/sysroots/x86_64-mpsssdk-linux/usr/share/doc/gdb-7.6.50/GDB.pdf` for more information.

Multithreaded Programming

Xeon Phi Parallel Programming

- MPI can be used for distributed programming on Xeon Phi, but it is limited to only the native mode.
- Efficient utilization of Xeon Phi is usually obtained from multithreading in both the offload and native mode.
- This requires workloads that are highly parallel.

Threading Technologies

- pthreads
  - raw interface to threads on POSIX systems
  - lacks sophisticated scheduler
  - requires extensive manual work to use in production level code
- OpenMP (http://www.openmp.org/specifications/)
  - supported by many compilers including Intel 17 compiler
  - supported by C/C++ and Fortran languages
  - variety of parallel constructs suitable for specific situations
- Intel Thread Build Blocks (TBB) (https://www.threadingbuildingblocks.org/)
  - allows logical expression of parallelism
  - automatically maps parallel tasks to threads
  - can coexist with other threading technologies
- Intel Cilk Plus (https://www.cilkplus.org/)
  - extension of C/C++ to support task and data parallelism
  - sophisticated work-stealing scheduler for automatic load balancing
  - allows expression of task parallelism with serial semantics
- Other technologies: OpenACC, OpenCL

OpenMP: Introduction

Advantages

- platform independence since it is a standard adapted by compilers
- much of boilerplate code is hidden behind pragmas
- maintains thread pool that eliminates the cost of frequent thread creation and destruction
- has scheduler that automatically performs task scheduling
- gives access to raw threads as well as allows abstract expression of parallelism

OpenMP: Programming Model and Memory Model

Programming model

- fork-join model - master thread spawns a team of threads
- incremental adaptation of parallelism into programs
- arranged in three parts
  - programming API (use `#include <omp.h>`)
  - `#pragma` directives
  - environment variables

Memory Model

- All threads share memory of the process they belong to.
- Concurrent access to the shared data.
- Each thread has its own private data that is not shared by other threads.

OpenMP: Parallel for Loop

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

Compilation

- For Intel 17 compiler, use `-qopenmp` flag to compile.
- OpenMP pragmas's are recognized only when `-qopenmp` is used.

Execution

- How many threads?
  - By default, thread team contains the number of threads equal to the number of hyperthreads on the processor, unless...
    - ★ `OMP_NUM_THREADS` is set, or
    - ★ `omp_set_num_threads(nthreads)` is called, or
    - ★ `num_threads` clause is specified in the pragma.

OpenMP: Parallel for Loop

- Shared/private data: By default, all variables are shared, unless explicitly marked as private

```
#pragma omp parallel for shared(a,b,c,n) private(b)
for(int i = 0; i < n; ++i) {
  d[i] = a[i] + b*c[i];
  ++b;
}
```

- Conditional parallelization: if `n > threshold` is false, the loop is executed serially.

```
#pragma omp parallel for if(n > threshold)
for(int i = 0; i < n; ++i) {
  d[i] = a[i] + b*c[i];
}
```

- Implicit barrier at the end of the parallel for, unless `nowait clause`.

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

OpenMP: Parallel Region and Worksharing Constructs

- Marking parallel region:

```
#pragma omp parallel [clauses...]
{
  // worksharing constructs
}
```

- Specifying worksharing constructs:
  - For loop:

```
#pragma omp for
for(int i = 0; i < n; ++i) {
  ...
} // implied barrier unless nowait
```

  - Sections:

```
#pragma omp sections
{
  #pragma omp section
    <code block1>
  #pragma omp section
    <code block2>
  ...
} // implied barrier unless nowait
```

OpenMP: Parallel Region and Worksharing Constructs

- Specifying worksharing constructs
  - Single:

```
#pragma omp single
{
  // executed by only one thread
} // implied barrier unless nowait
```

  - Master:

```
#pragma omp master
{
  // executed by only the master thread - thread 0
} // NO implied barrier
```

Note:

- Woksharing constructs must be enclosed in parallel region.
- It must be encountered by all threads.
- Only the parallel region launches new threads, work sharing construct just specifies the parallel work.

OpenMP: Parallel region/Worksharing Clauses

- `private(list...)`:
  - Variables in list are private to each thread.
  - They are uninitialized on entry to the parallel region.
- `firstprivate(list...)`:
  - Variables in list are private to each thread.
  - They are initialized from original object before the parallel region.
- `lastprivate(list...)`:
  - Variables in list are private to each thread.
  - The original object before the parallel region is updated by a thread that executes lexically last section/loop iteration.

OpenMP: Critical, Atomics and Reductions

Consider the example:

```
#pragma omp parallel for
for(int i = 0; i < n; ++i) {
  sum += a[i]; // leads to data races unless protected using
      critical or atomic
}
```

The statement `sum += a[i]` leads to data races unless protected using `critical` or `atomic` as follows.

- Using critical section:

  ```
  #pragma omp parallel for
  for(int i = 0; i < n; ++i) {
    #pragma omp critical
    {
      sum += a[i];
    }
  }
  ```

  - Pros: Can execute multiple statements in thread safe manner.
  - Cons: Expensive.

OpenMP: Critical, Atomics and Reductions

- Using atomic:

```
#pragma omp parallel for
for(int i = 0; i < n; ++i) {
  #pragma omp atomic
  sum += a[i];
}
```

  - Pros: Lightweight
  - Cons: Can execute only a single statement with update, read, write or capture semantics to a variable.

If the purpose is to perform reduction use OpenMP reductions instead.

```
#pragma omp parallel for reduction(+:sum)
for(int i = 0; i < n; ++i) {
  sum += a[i];
}
```

OpenMP: False Sharing

Reduction can also be performed as follows.

```
float * thread_sum = new float[nthreads];
for(int i = 0; i < nthreads; ++i)
  thread_sum[i] = 0.0;

#pragma omp parallel num_threads(nthreads)
{
  int t = omp_get_thread_num();
  // calculate iteration space - [start, end) - for this thread
  for(int i = start; i < end; ++i)
    thread_sum[t] += a[i];
}

float sum = 0.0;
for(int i = 0; i < nthreads; ++i)
  sum += thread_sum[i];

delete[] thread_sum;
```

But DON'T use this approach. Why?

OpenMP: Tasks

- Introduced in OpenMP 3.0
- Allows dynamic task graph creation, which is executed in parallel by in-built OpenMP scheduler.

Consider the example of serial quicksort:

```
void quickSort(int arr[], int low, int high) {
  if(low < high) {
    pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);  // Before pi
    quickSort(arr, pi + 1, high); // After pi
  }
}
```

- `partition()` is serial, but subsequent `quickSort()`s are data independent.
- This forms a task graph.
- OpenMP strategy: create tasks for the independent `quickSort()`s.

OpenMP: Tasks

Parallel quicksort using OpenMP:

```
void quickSort(int arr[], int low, int high) {
  if(low < high) {
    pi = partition(arr, low, high);
    #pragma omp task firstprivate(arr,low,pi)
    quickSort(arr, low, pi - 1);  // Before pi
    #pragma omp task firstprivate(arr,high,pi)
    quickSort(arr, pi + 1, high); // After pi
  }
}
```

- Created tasks for each `quickSort()`.
- Note that a, `low`, `high` and `pi` are `firstprivate` since subsequent `quickSort()` need them initialized from parent `quickSort()`.
- We also need a driver code to start the quicksort in parallel.

```
#pragma omp parallel shared(a, nelements)
{
  #pragma omp single nowait
  { quickSort(a, 0, nelements-1); }
}
```

Vectorization

Vectorization

- Data parallel programming which supports operating on multiple data items in a single instructions.
- a.k.a. Single Instruction Multiple Data (SIMD) parallelism according to Flynn's taxonomy.
- Most of the modern processors support SIMD parallelism.
- Xeon Phi has 512 bit SIMD units.
  - can perform operations on 16 floats/8 doubles in one instruction
  - vectorization is absolutely essential to gain efficiency on Xeon Phi.
- Vectorization approaches:
  - Autovectorization: This is performed by compiler with or without assistance from programmers.
  - Intrinsics: Programmers control the vectorization using special functions provided by compiler called as intrinsics. Compiler translates intrinsics to assembly instructions.

Autovectorization of Intel C++ 17 Compiler

- Compiler always tries to perform vectorization as long as (-qno-vec) flag is not specified.
- To see which code is vectorized use `-qopt-report[=n]` flag, where `n` specifies the level of detail of the optimization report.

Example:

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

The compiler generates vector instructions for the loop body because,

- the loop is countable - `n` does not change in the loop
- each iteration is data independent
- the loop has single entry and single exit (i.e. no `break` statement)
- the loop body has straight line code
- the loop does not call any other functions
- data of consecutive loops in each array is contiguous

If these conditions are not met, autovectorization produces scalar code.

Assisting Autovectorization

If compiler cannot prove that the loop iterations are independent, it does not perform vectorization of the loop. For example,

```
void linear_add(float * a, float * b, float * c, int n) {
  for(int i = 0; i < n; ++i) a[i] = b[i]+c[i];
}
```

In this case, compiler cannot know whether the memory pointed by the pointers overlaps or not. So the loop is not vectorized. The code needs to be modified as follows to enable vectorization.

```
void linear_add(float * a, float * b, float * c, int n) {
  #pragma ivdep
  for(int i = 0; i < n; ++i) a[i] = b[i]+c[i];
}
```

OR

```
void linear_add(float * restrict a, float * restrict b, float *
    restrict c, int n) {
  for(int i = 0; i < n; ++i) a[i] = b[i]+c[i];
}
```

Other Autovectorization Hints

- `#pragma loop count(n)`: specifies the loop trip count so that compiler can decide whether to vectorize the loop or not based on cost analysis
- `#pragma vector always`: requests that the loop be vectorized irrespective of the cost analysis, if it is safe to do so
- `#pragma vector align`: asserts that the arrays in following loop point to aligned data
- `#pragma novector`: asks compiler not to vectorize following loop
- `#pragma vector nontemporal`: provides hint to the compiler that write only array data in following loop can be written using streaming stores

Explicit Vectorization

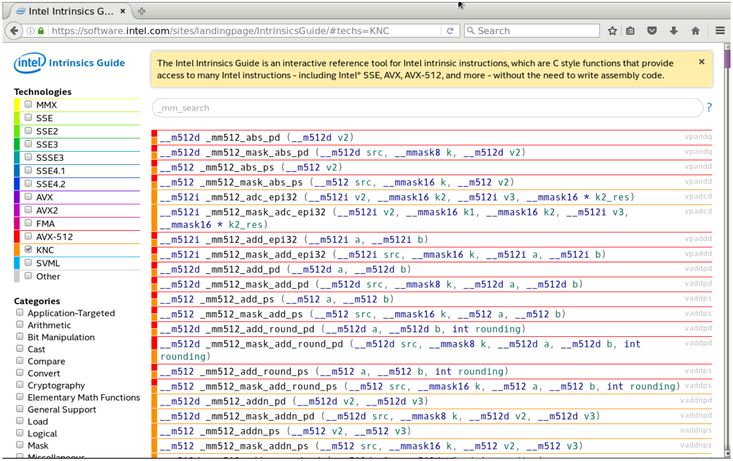
- If the compiler autovectorizer is not sufficient, programmers can use explicit vector instructions.
- Intel compiler provides intrinsic functions that allow access to these instructions.

For example,

```
void linear_add(float * a, float * b, float * c, int n) {
  // a, b and c are assumed aligned on 64 byte boundary since
  // the load and store expect aligned address
  // otherwise segfault occurs
  for(int i = 0; i < n/16; i+=16) {
    __m512 breg = _mm512_load_ps(&b[i]);
    __m512 creg = _mm512_load_ps(&c[i]);
    __m512 res = _mm512_add_ps(breg, creg);
    _mm512_store_ps(&a[i], res);
  }
}
```

Explicit Vectorization

Intel has provided an interactive guide for browsing the intrinsic functions.
See https://software.intel.com/sites/landingpage/IntrinsicsGuide/



Intel Intrinsics G...

https://software.intel.com/sites/landingpage/IntrinsicsGuide/#techs=KNC

Search

intel Intrinsics Guide

The Intel Intrinsics Guide is an interactive reference tool for Intel intrinsic instructions, which are C style functions that provide access to many Intel instructions - including Intel® SSE, AVX, AVX-512, and more - without the need to write assembly code.

Technologies

- ☐ MMX
- ☐ SSE
- ☐ SSE2
- ☐ SSE3
- ☐ SSSE3
- ☐ SSE4.1
- ☐ SSE4.2
- ☐ AVX
- ☐ AVX2
- ☐ FMA
- ☐ AVX-512
- ☑ KNC
- ☐ SVML
- ☐ Other

Categories

- ☐ Application-Targeted
- ☐ Arithmetic
- ☐ Bit Manipulation
- ☐ Cast
- ☐ Compare
- ☐ Convert
- ☐ Cryptography
- ☐ Elementary Math Functions
- ☐ General Support
- ☐ Load
- ☐ Logical
- ☐ Mask

_mm_search

```
__m512d _mm512_abs_pd (__m512d v2)                                                   vpandq
__m512d _mm512_mask_abs_pd (__m512d src, __mmask8 k, __m512d v2)                     vpandq
__m512 _mm512_abs_ps (__m512 v2)                                                     vpandd
__m512 _mm512_mask_abs_ps (__m512 src, __mmask16 k, __m512 v2)                       vpandd
__m512i _mm512_adc_epi32 (__m512i v2, __mmask16 k2, __m512i v3, __mmask16 * k2_res)  vpadcd
__m512i _mm512_mask_adc_epi32 (__m512i v2, __mmask16 k1, __mmask16 k2, __m512i v3,   vpadcd
__mmask16 * k2_res)
__m512i _mm512_add_epi32 (__m512i a, __m512i b)                                      vpaddd
__m512i _mm512_mask_add_epi32 (__m512i src, __mmask16 k, __m512i a, __m512i b)       vpaddd
__m512d _mm512_add_pd (__m512d a, __m512d b)                                         vaddpd
__m512d _mm512_mask_add_pd (__m512d src, __mmask8 k, __m512d a, __m512d b)           vaddpd
__m512 _mm512_add_ps (__m512 a, __m512 b)                                            vaddps
__m512 _mm512_mask_add_ps (__m512 src, __mmask16 k, __m512 a, __m512 b)              vaddps
__m512d _mm512_add_round_pd (__m512d a, __m512d b, int rounding)                     vaddpd
__m512d _mm512_mask_add_round_pd (__m512d src, __mmask8 k, __m512d a, __m512d b, int vaddpd
rounding)
__m512 _mm512_add_round_ps (__m512 a, __m512 b, int rounding)                        vaddps
__m512 _mm512_mask_add_round_ps (__m512 src, __mmask16 k, __m512 a, __m512 b, int rounding) vaddps
__m512d _mm512_addn_pd (__m512d v2, __m512d v3)                                      vaddnpd
__m512d _mm512_mask_addn_pd (__m512d src, __mmask8 k, __m512d v2, __m512d v3)        vaddnpd
__m512 _mm512_addn_ps (__m512 v2, __m512 v3)                                         vaddnps
__m512 _mm512_mask_addn_ps (__m512 src, __mmask16 k, __m512 v2, __m512 v3)           vaddnps
```

Performance Measurement

Performance Measurement

- Performance is usually measured in the amount of time taken to execute a code. But there can be other measures as well such as cache hit/miss rate, power, DRAM traffic.
- Timing measurements can be performed using various means.
  - `omp_get_wtime()` function of OpenMP
  - `clock_gettime()` function available on POSIX compliant systems
  - Intel VTune Amplifier (https://software.intel.com/en-us/intel-vtune-amplifier-xe)
  - GNU gprof (https://sourceware.org/binutils/docs/gprof/)
- Cache hit/miss rate, power etc. can be measured using performance monitoring unit (PMU) available on various processors. They can be accessed using,
  - Performance API (PAPI) (http://icl.cs.utk.edu/papi/)
  - Intel VTune Amplifier