LLNL-CONF-543971

Index Sets and Vectorization

J. A. Keasler

April 2, 2012

Emerging Technologies in HPC Application Development
Livermore, CA, United States
March 19, 2012 through March 21, 2012

Index Sets and Vectorization

LLNL Emerging Technologies in HPC Application Development Workshop

March 19-21, 2012

Jeff Keasler and Rich Hornung

Lawrence Livermore National Laboratory

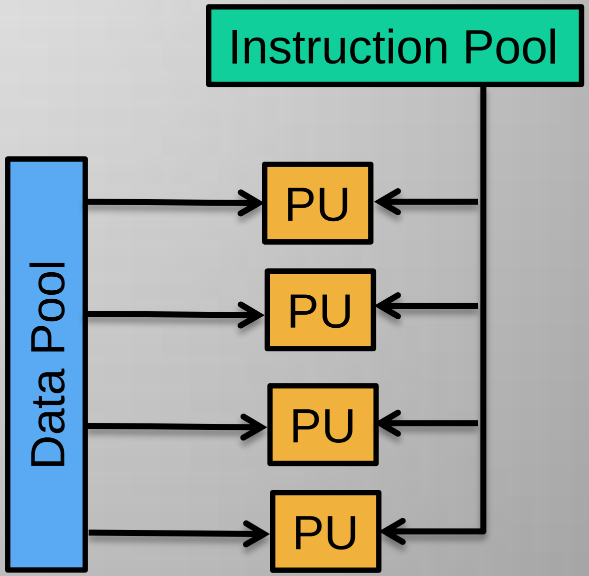
LLNL-PRES-xxxxxx

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

What is vectorization?

- Data parallelism (SIMD, SIMT, etc.): extension of ISA enabling the same instruction to be performed on multiple data items simultaneously
- Many/most CPUs support vectorization in some form
- Hardware core vector lengths

SSE		4 SP
SSE2	2 DP,	4 SP
AVX	4 DP,	8 SP
MIC	8 DP,	16 SP
BG/P	2 DP	
BG/Q	4 DP	



PU = processing unit

Vectorization is difficult to enable, but can yield large efficiency gains

- Extra programmer effort is required
  - Not all algorithms can be vectorized (regular algorithm structure & fine-grain parallelism must be used)
  - Most CPUs have data alignment restrictions for load/store operations (obey or risk incorrect code)
  - Special directives are often needed to enable vectorization
  - Vector instructions are architecture-specific

+ Vectorization is the best way to optimize for power and performance due to reduced clock cycles
  + When data is organized properly, a vector load instruction (i.e. movaps) can replace 'normal' load instructions (i.e. movsd)
  + Vector operations can potentially have a smaller footprint in the instruction cache when fewer instructions need to be executed

Simple operation example a[i] = b[i]*b[i], SSE2, aligned

- Vectorized (DP 2 load, 2 store, 2 flop, 6 cycle)

```
loop: movaps    (%rbx,%rsi,8), %xmm0
      mulpd     %xmm0, %xmm0
      movaps    %xmm0, (%rdx,%rsi,8)
      addq      $2, %rsi
      cmpq      %rax, %rsi
      jb        loop
```

- Unvectorized(DP 1 load, 1 store, 1 flop, 6 cycle)

```
loop: movsd     (%rbx,%rax,8), %xmm0
      mulsd     %xmm0, %xmm0
      movsd     %xmm0, (%rdx,%rax,8)
      incq      %rax
      cmpq      %rcx, %rax
      jb        loop
```

Simple operation example a[i] = b[i]*b[i], SSE2, unaligned

- Vectorized loop:

```
movsd    (%rsi,%rdi,8), %xmm0
movhpd   8(%rsi,%rdi,8), %xmm0
mulpd    %xmm0, %xmm0
movsd    %xmm0, (%rdx,%rdi,8)
movhpd   %xmm0, 8(%rdx,%rdi,8)
movsd    16(%rsi,%rdi,8), %xmm1
movhpd   24(%rsi,%rdi,8), %xmm1
mulpd    %xmm1, %xmm1
movsd    %xmm1, 16(%rdx,%rdi,8)
movhpd   %xmm1,24(%rdx,%rdi,8)
movsd    32(%rsi,%rdi,8), %xmm2
movhpd   40(%rsi,%rdi,8), %xmm2
mulpd    %xmm2, %xmm2
movsd    %xmm2, 32(%rdx,%rdi,8)
movhpd   %xmm2, 40(%rdx,%rdi,8)
movsd    48(%rsi,%rdi,8), %xmm3
movhpd   56(%rsi,%rdi,8), %xmm3
mulpd    %xmm3, %xmm3
movsd    %xmm3, 48(%rdx,%rdi,8)
movhpd   %xmm3, 56(%rdx,%rdi,8)
addq     $8, %rdi
cmpq     %rax, %rdi
jb loop
```

- This is the ***typical*** vectorized code produced by the compiler for SSE2
- Executes four instructions per operation instead of three
- Above code does not show generated prologue and epilogue code
- Does not show additional loop versions for other alignment cases
- This has a large footprint in the instruction cache

Simple operation example a[i] = b[i]*b[i], AVX, unaligned

- Vectorized

```
loop:
vmovupd   32(%r8,%rcx,8), %ymm1
vmovupd   (%r8,%rcx,8), %ymm0
vmulpd    %ymm1, %ymm1, %ymm3
vmulpd    %ymm0, %ymm0, %ymm2
vmovupd   %xmm2, (%rdi,%rcx,8)
vmovupd   %xmm3, 32(%rdi,%rcx,8)
vextractf128 $1, %ymm2, 16(%rdi,%rcx,8)
vextractf128 $1, %ymm3, 48(%rdi,%rcx,8)
vmovupd   96(%r8,%rcx,8), %ymm5
vmovupd   64(%r8,%rcx,8), %ymm4
vmulpd       %ymm5, %ymm5, %ymm7
vmulpd       %ymm4, %ymm4, %ymm6
vmovupd      %xmm6, 64(%rdi,%rcx,8)
vmovupd      %xmm7, 96(%rdi,%rcx,8)
vextractf128 $1, %ymm6, 80(%rdi,%rcx,8)
vextractf128 $1, %ymm7, 112(%rdi,%rcx,8)
addq         $16, %rcx
cmpq         %rdx, %rcx
jb           loop
```

•This is the typical vectorized code produced by the compiler for AVX
•Similar issues to SSE2, but operations work on longer vectors
• Above code does not show generated prologue and epilogue code
• Does not show additional loop versions for other alignment cases
• This has a large footprint in the instruction cache

Simple operation example a[i] = b[i]*b[i], AVX, aligned

- Vectorized (16 load, 16 store, 16 flop, 15 cycle)

```
loop:
vmovupd    (%rbx,%rsi,8), %ymm0
vmulpd     %ymm0, %ymm0, %ymm1
vmovupd    %ymm1, (%rdx,%rsi,8)
vmovupd    32(%rbx,%rsi,8), %ymm2
vmulpd     %ymm2, %ymm2, %ymm3
vmovupd    %ymm3, 32(%rdx,%rsi,8)
vmovupd    64(%rbx,%rsi,8), %ymm4
vmulpd     %ymm4, %ymm4, %ymm5
vmovupd    %ymm5, 64(%rdx,%rsi,8)
vmovupd    96(%rbx,%rsi,8), %ymm6
vmulpd     %ymm6, %ymm6, %ymm7
vmovupd    %ymm7, 96(%rdx,%rsi,8)
addq       $16, %rsi
cmpq       %rax, %rsi
jb         loop
```

likely bandwidth limited with this code density per core

- This could be the default code generated with the proper programming model: no prologue, no epilogue, no loop versioning

Hybrid index sets insulate users from architecture specific details

- We have applied hybrid index sets to achieve optimal vectorization
- We can extend this concept to handle other programming models
- Finally, we can do even better!
  - Current traversal implementations contain detailed compiler directives
  - We have proven the compiler is capable of good vectorization but not conveniently exposed
  - We would like to start a dialogue with compiler vendors to better expose vectorization for our needs

Lawrence Livermore National Laboratory