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
Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.
Index Sets and Vectorization

LLNL Emerging Technologies in HPC Application Development Workshop
March 19-21, 2012

Jeff Keasler and Rich Hornung
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

\[ a[i] = b[i] \times b[i], \text{ SSE2, unaligned} \]

- Vectorized loop:

  ```assembly
  movsdx (%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 %xmm1, 16(%rsi,%rdi,8)
  movhpd %xmm1, 16(%rdx,%rdi,8)
  mulpd %xmm1, %xmm1
  movsd %xmm1, 24(%rsi,%rdi,8)
  movhpd %xmm1, 24(%rdx,%rdi,8)
  movsd %xmm2, 32(%rsi,%rdi,8)
  movhpd %xmm2, 32(%rdx,%rdi,8)
  mulpd %xmm2, %xmm2
  movsd %xmm2, 32(%rdx,%rdi,8)
  movhpd %xmm2, 40(%rdx,%rdi,8)
  movsd %xmm3, 48(%rsi,%rdi,8)
  movhpd %xmm3, 48(%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

\[ a[i] = b[i] * b[i], \text{ AVX, unaligned} \]

- Vectorized loop:
  - \text{vmovupd} \ 32(%r8,%rcx,8), %ymm1
  - \text{vmovupd} \ (%r8,%rcx,8), %ymm0
  - \text{vmulpd} \ %ymm1, %ymm1, %ymm3
  - \text{vmulpd} \ %ymm0, %ymm0, %ymm2
  - \text{vmovupd} \ %xmm2, (%rdi,%rcx,8)
  - \text{vmovupd} \ %xmm3, 32(%rdi,%rcx,8)
  - \text{vextractf128} \ $1, %ymm2, 16(%rdi,%rcx,8)
  - \text{vextractf128} \ $1, %ymm3, 48(%rdi,%rcx,8)
  - \text{vmovupd} \ 96(%r8,%rcx,8), %ymm5
  - \text{vmovupd} \ 64(%r8,%rcx,8), %ymm4
  - \text{vmulpd} \ %ymm5, %ymm5, %ymm7
  - \text{vmulpd} \ %ymm4, %ymm4, %ymm6
  - \text{vmovupd} \ %xmm6, 64(%rdi,%rcx,8)
  - \text{vmovupd} \ %xmm7, 96(%rdi,%rcx,8)
  - \text{vextractf128} \ $1, %ymm6, 80(%rdi,%rcx,8)
  - \text{vextractf128} \ $1, %ymm7, 112(%rdi,%rcx,8)
  - \text{addq} \ $16, %rcx
  - \text{cmpeq} \ %rdx, %rcx
  - \text{jb} \ \text{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
\( a[i] = b[i] \times b[i], \text{ AVX, aligned} \)

- Vectorized (16 load, 16 store, 16 flop, 15 cycle) loop:
  
  ```
  vmovupd (%rbx,%rsi,8), %ymm0
  vmulpd %ymm0, %ymm0, %ymm1
  vmovupd %ymm1, (%rdx,%rsi,8)
  vmulpd %ymm2, %ymm2, %ymm3
  vmovupd %ymm3, 32(%rbx,%rsi,8), %ymm2
  vmulpd %ymm4, %ymm4, %ymm5
  vmovupd %ymm5, 64(%rbx,%rsi,8), %ymm4
  vmulpd %ymm6, %ymm6, %ymm7
  vmovupd %ymm7, 96(%rdx,%rsi,8)
  addq $16, %rsi
  cmpq %rax, %rsi
  jb loop
  ```

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

likely bandwidth limited with this code density per core
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