Vectoriation using Intel composer XE 2013 for Linux

- **实验需求**
  - Intel® Composer XE 2013 Initial Release
  - At least 2nd generation Intel® Core™ processor (with Intel® AVX)
  - For stable measurement it is recommended to turn off Intel® HyperThreading, Intel SpeedStep® and Intel® Turbo Boost
- **实验目标**
  - ✓ 选择正确的指令集扩展开关进行编译
  - ✓ 生成和分析矢量化和优化报告
  - ✓ 处理内存别名和对齐问题，可能会阻止矢量化
  - ✓ 理解跨过程优化对于获得最佳矢量化结果的好处
  - ✓ 通过使用#pragma/directive SIMD来改进矢量化
  - ✓ 使用Guided Auto Parallelism (GAP)——特别是用于矢量化

C/C++实验

- **实验1.1 无法矢量化：获取矢量化的诊断信息**
  1. 进入**matvector/c**目录
  2. 使用参数**-no-vec**关掉由编译器提供的level2的矢量化优化

     ```
     $ icc –ansi-alias -O2 –no-vec multiply.c driver.c -o matvector
     $ ./matvector
     ```

     记录执行时间：48.045
  3. 打开AVX的矢量化：

     ```
     $ icc –ansi-alias -O2 –xavx multiply.c driver.c -o matvector
     $ ./matvector
     ```

     记录执行时间：45.395
  4. 在**multiply.c** 中我们预设了pattern来制约性能，如何发现和修正呢?
  5. 通过矢量化报告来获取信息：

     ```
     $ icc –ansi-alias -O2 –xavx multiply.c driver.c -o matvector –vec-report3
     ```

     提供的信息就包含了哪些loops是否被矢量化，供开发者优化

C/C++实验

- **实验1.2 内部loop的矢量化：**
  1. 关掉整个编译单元中所有函数参数的别名

     $ icc –ansi-alias -O2 –xavx multiply.c driver.c -o matvector -fargument-noalias

     $ ./matvector

     记录执行时间：44.195

  2. 可以为每个loop加上**#pragma ivdep**关键字，说明无循环依赖

     $ icc –ansi-alias -O2 –xavx multiply.c driver.c -o matvector

     $ ./matvector

     记录执行时间：21.044

  3. 也可以使用**-restrict**选项：

     $ icc –ansi-alias -O2 –xavx multiply.c driver.c -o matvector –restrict

     $ ./matvector

     记录执行时间：20.547

C/C++实验

- **实验1.3 对齐改进：通过对齐数据提升性能**
  1. 使用__attribute__((aligned(…))) 将driver.c中的a,b,c三个数组对齐

     ```
     $ icc –ansi-alias -O2 –xavx multiply.c driver.c -o matvector
     $ ./matvector
     ```

     记录执行时间：20.715
  2. 使用__assume_aligned(…)告知multiply.c数组做了对齐处理

     命令同上

     记录执行时间：19.103
  3. 也可以对row填充来适配AVX向量的容量(Solution: solutions/padding)

     命令同上

     记录执行时间：11.388
  4. 使用#pragma vector aligned确保multiply.c中row元素数量匹配SIMD向量的容量

     命令同上

     记录执行时间：**11.028**

C/C++实验

- **实验2 使用过程间优化(IPO)**
  1. 打开square_charge/c 目录
  2. 不使用IPO编译

```
$ icc -ansi-alias -O3 -xavx rabs.c square_charge.c trap.c twod.c -o sq
$ ./sq
```

  3. 使用IPO编译

```
$ icc -ansi-alias -O3 -xavx rabs.c square_charge.c trap.c twod.c -o sq
-ipo
$ ./sq
```

  4. 使用-opt-report-phase ipo_inl 来理解编译器如何优化

C/C++实验

- **实验3 Pragma SIMD**
  1. 打开simd/c 目录
  2. 不做修改直接编译运行

     ```
     $ icc -ansi-alias -O2 -xavx main.c simd.c -o simd
     $ ./simd
     ```
  3. 通过报告来寻找指令耗时长的原因

     ```
     $ icc -ansi-alias -O2 -xavx main.c simd.c -o simd -vec-report3
     ```

     使用#pragma simd 进行向量化(参考solutions/simd)

     命令同上
  4. 参考solutions/vectorlength对simd.c修改

     命令同上
  5. 参考solutions/reduction检查是否是因为reduction引起的错误

     命令同上
  6. 参考solutions/linear检查是否因为指针线性增加引起的错误

     命令同上

C/C++实验

- **实验4 Guided Auto Parallelism(GAP)**
  1. 打开gap/c 目录
  2. 检查gap.c是否有向量化可以优化的地方
  3. 使用GAP来获取编译器的报告

     $ icc -ansi-alias -std=c99 -O2 -xavx -c gap.c –guide -parallel
  4. 有3种GAP策略

     $ icc -ansi-alias -std=c99 -O2 -xavx -c gap.c –parallel –guide-par (自动并行化)

     $ icc -ansi-alias -std=c99 -O2 -xavx -c gap.c –guide-vec (自动矢量化)

     $ icc -ansi-alias -std=c99 -O2 -xavx -c gap.c –guide-data-trans (数据转换)
  5. 增加选项 “**--guide-opts=...**” 将 GAP 限制到选定的函数，重新运行并将分析限制在函数**transform(…)**或**mult(…)**

Fortran实验

- **实验1.1 无法矢量化：获取矢量化的诊断信息**
  1. 进入matvector/fortran目录
  2. 使用参数-no-vec关掉由编译器提供的level2的矢量化优化

     ```
     $ ifort -O2 –no-vec driver.f90 multiply.f90 -o matvector
     $ ./matvector
     ```

     记录执行时间：___
  3. 打开AVX的矢量化：

     ```
     $ ifort -O2 –xavx driver.f90 multiply.f90 -o matvector
     $ ./matvector
     ```

     记录执行时间：___
  4. 在multiply.f90 中我们预设了pattern来制约性能，如何发现和修正呢?
  5. 通过矢量化报告来获取信息：

     ```
     $ ifort -O2 –xavx driver.f90 multiply.f90 -o matvector –vec-report3
     ```

     提供的信息就包含了哪些loops是否被矢量化，供开发者优化

Fortran实验

- **实验1.2 内部loop的矢量化：**
  1. 关掉通过在整个编译单元中所有函数参数的别名(-fno-alias & -fno-fnalias)

     ```
     $ ifort -O2 -xavx driver.f90 multiply.f90 -o matvector -fno-alias
     $ ./matvector
     ```

     记录执行时间：___
  2. 进入子目录array_version尝试将数组直接传递：

     ```
     $ ifort -O2 -xavx driver_array.f90 multiply_array.f90 -o matvector -vec-report3
     $ ./matvector
     ```

     记录执行时间：___
  3. 作为下一个活动的准备，生成汇编代码 (-S)。你可以看到编译器在函数开头生成了多个版本和测试。这些是由于数组元素的未知对齐引起的。

Fortran实验

- **实验1.3 对齐改进：通过对齐数据提升性能**
  1. 参考solutions/align 将driver_array.f90中的a,b,c三个数组对齐

     ```
     $ ifort -O2 -xavx driver_array.f90 multiply_array.f90 -o matvector
     $ ./matvector
     ```

     记录执行时间：___
  2. 参考solutions/assume_aligned告知multiply_array.f90数组做了对齐处理

     命令同上

     记录执行时间：___
  3. 也可以对row填充来适配AVX向量的容量(Solution: solutions/padding)

     命令同上

     记录执行时间：___
  4. 确保multiply_array.f90中row元素数量匹配SIMD向量的容量

     命令同上

     记录执行时间：___

Fortran实验

- **实验2 使用跨进程优化(IPO)**
  1. 打开square_charge/fortran 目录
  2. 不使用IPO编译

     ```
     $ ifort -O3 -xavx rabs.f90 square_charge.f90 trap.f90 twod.f90 -o sq
     $ ./sq
     ```

     记录执行时间：___
  3. 使用IPO编译

     ```
     $ ifort -O3 -xavx rabs.f90 square_charge.f90 trap.f90 twod.f90 -o sq –ipo
     $ ./sq
     ```

     记录执行时间：___
  4. 使用-opt-report-phase ipo_inl 来理解编译器如何优化

Fortran实验

- **实验3 Directive SIMD**
  1. 打开simd/fortran 目录
  2. 不做修改直接编译运行

     ```
     $ ifort -O2 -xavx main.f90 simd.f90 -o simd
     $ ./simd
     ```
  3. 通过报告来寻找指令耗时长的原因

     ```
     $ ifort -O2 -xavx main.f90 simd.f90 -o simd -vec-report3
     ```

     使用!DIR$ SIMD进行向量化(参考solutions/simd)

     命令同上
  4. 参考solutions/vectorlength对simd.f90修改

     命令同上
  5. 参考solutions/reduction检查是否是因为reduction引起的错误

     命令同上
  6. 参考solutions/linear检查是否因为指针线性增加引起的错误

     命令同上

Fortran实验

- 实验4 Guided Auto Parallelism(GAP)
  1. 打开gap/fortran 目录
  2. 检查gap.f90是否有向量化可以优化的地方
  3. 使用GAP来获取编译器的报告

     ```
     $ ifort -O2 -xavx -c gap.f90 –guide -parallel
     ```

  4. 有3种GAP策略

     ```
     $ ifort -O2 -xavx -c gap.f90 –parallel–guide-par (自动并行化)
     $ ifort -O2 -xavx -c gap.f90 –guide-vec (自动矢量化)
     $ ifort -O2 -xavx -c gap.f90 –guide-data-trans (数据转换)
     ```

  5. 增加选项 “--guide-opts=...” 将 GAP 限制到选定的函数，重新运行并将分析限制在函数transform(…)或mult(…)