



arm

Hands on with Arm Tools

GoingArm, HiPEAC 2018

Outline

- Outcomes of the hands-on
- Introduction to Arm tools
 - HPC Compiler
 - Performance Libraries
 - Performance analysis tools
- Introduction to hands on-material

Outcomes

- In this session you will:
 - Work on single-node Arm hardware (Cavium ThunderX)
 - Use the Arm compiler to generate a binary
 - Use flags to generate different binaries and compiler advice
 - Use the Arm Performance Libraries for optimal performance
 - Observe performance data in the Arm MAP performance analysis tool

Arm HPC Compiler

- C / C++ and Fortran support
 - armclang
 - armclang++
 - armflang
- Based on LLVM
- Optimized for HPC use cases
- Just like any other compiler – easy to pick up in this session

Arm Performance Libraries

- Optimized BLAS, FFT, etc for many-core nodes
- Specialized support for Arm instruction set architectures
- Specialized support for specific hardware

Arm Performance Analysis

- Multi-architecture support for debugging and performance analysis
- Scalable (many thousands of nodes), low-overhead and easy to use
- Timeline of application behavior

Hands-on Material

Tutorial Problem

- Matrix-matrix multiplication in C/C++:
 - Focus not on understanding the problem, but how to use the Arm toolchain
 - Naïve to optimized performance
- Program flow
 - Initialize random data
 - Perform multiply
- Use of compiler for single core and multi-core application
- Understanding of application performance

Matrix-Matrix Multiply: Version 1

- Naïve implementation works on single core
- No consideration of underlying hardware

```
for (i= 0; i < n; ++i){
  for (j= 0; j < l; ++j){
    for (k= 0; k < m; ++k){
      matC[i][j] += matA[i][k] * matB[k][j];
    }
  }
}
```

- How bad is performance? Easier to evaluate when compared to other code

Matrix-Matrix Multiply: Version 2

- Add blocking to be able to take advantage of cache

```
for (kk= 0; kk < m; kk+= blockSize){
    for(jj= 0; jj < l; jj+= blockSize){
        for(i= 0; i < n; i+= 2){
            for(j= jj; j < min(l, jj + blockSize); j+= 2){
                for(k =kk; k < min(m, kk + blockSize); ++k){
                    matC[i][j] += matA[i][k] * matB[k][j];
                    matC[i][j+1] += matA[i][k] * matB[k][j+1];
                    matC[i+1][j] += matA[i+1][k] * matB[k][j];
                    matC[i+1][j+1] += matA[i+1][k] * matB[k][j+1];
                }
            }
        }
    }
}
```

- Have only added one level of blocking, which can take advantage of a single level of cache only
- Also little bit of loop unrolling to allow re-use of registers

Matrix-Matrix Multiply: Version 2

- Add blocking to be able to take advantage of cache

```
for (kk= 0; kk < m; kk+= blockSize){
    for(jj= 0; jj < l; jj+= blockSize){
        for(i= 0; i < n; i+= 2){
            for(j= jj; j < min(l, jj + blockSize); j+= 2){
                for(k =kk; k < min(m, kk + blockSize); ++k){
                    matC[i][j] += matA[i][k] * matB[k][j];
                    matC[i][j+1] += matA[i][k] * matB[k][j+1];
                    matC[i+1][j] += matA[i+1][k] * matB[k][j];
                    matC[i+1][j+1] += matA[i+1][k] * matB[k][j+1];
                }
            }
        }
    }
}
```

- Performance improvement of greater than 2x at 1024x1024 with 128 block size
- Still tens of seconds to perform a small multiply – bring out the big guns

Matrix-Matrix Multiply: Version 3

- Use Arm Performance Libraries
 - Addition of `-larmpl` flag to build command
 - Standard BLAS interface

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, n, m, l, 1, dataA,  
            m, dataB, l, 1, dataC, m);
```

- Approximately 17x speed-up over blocked multiply and 40x over naïve implementation
- Still almost at a second run time at small 1024x1024 matrix

Matrix-Matrix Multiply: Version 4

- Have 96 physical cores available on the system – make use of them
- OpenMP directives added to loops
- Link to parallel version of the Arm Performance Libraries (-larmpl_mp)

```
#pragma omp parallel for private(i, j, jj, k, kk)
for (kk= 0; kk < m; kk+= blockSize){
  for(jj= 0; jj < l; jj+= blockSize){
    for(i= 0; i < n; i+= 2){
      for(j= jj; j < min(l, jj + blockSize); j+= 2){
        for(k =kk; k < min(m, kk + blockSize); ++k){
          matC[i][j] += matA[i][k] * matB[k][j];
          matC[i][j+1] += matA[i][k] * matB[k][j+1];
          matC[i+1][j] += matA[i+1][k] * matB[k][j];
          matC[i+1][j+1] += matA[i+1][k] * matB[k][j+1];
        }
      }
    }
  }
}
```

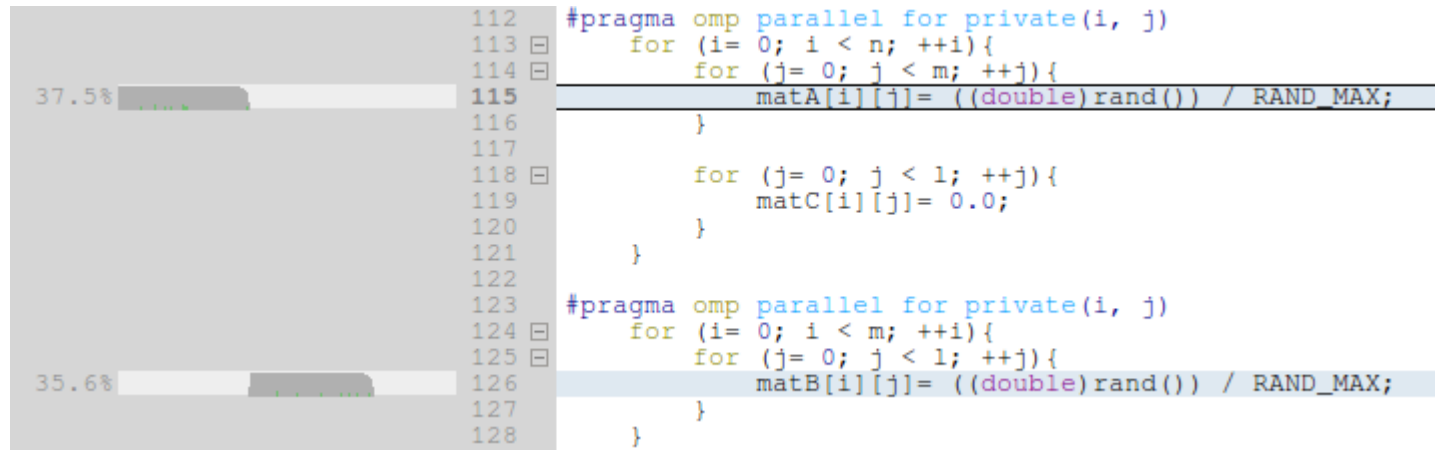
Matrix-Matrix Multiply: Version 4

- Have 96 physical cores available on the system – make use of them
- Small matrix size means not all threads can be used with simple OpenMP directive
- Arm Performance library utilizes all threads – approximately 100x faster than blocking approach

```
#pragma omp parallel for private(i, j, jj, k, kk)
for (kk= 0; kk < m; kk+= blockSize){
  for(jj= 0; jj < l; jj+= blockSize){
    for(i= 0; i < n; i+= 2){
      for(j= jj; j < min(l, jj + blockSize); j+= 2){
        for(k =kk; k < min(m, kk + blockSize); ++k){
          matC[i][j] += matA[i][k] * matB[k][j];
          matC[i][j+1] += matA[i][k] * matB[k][j+1];
          matC[i+1][j] += matA[i+1][k] * matB[k][j];
          matC[i+1][j+1] += matA[i+1][k] * matB[k][j+1];
        }
      }
    }
  }
}
```

Matrix-Matrix Multiply: Version 4 (Interlude)

- Time to populate matrices with random data much higher when using OpenMP
- Use Arm Forge (performance analysis toolset) to investigate



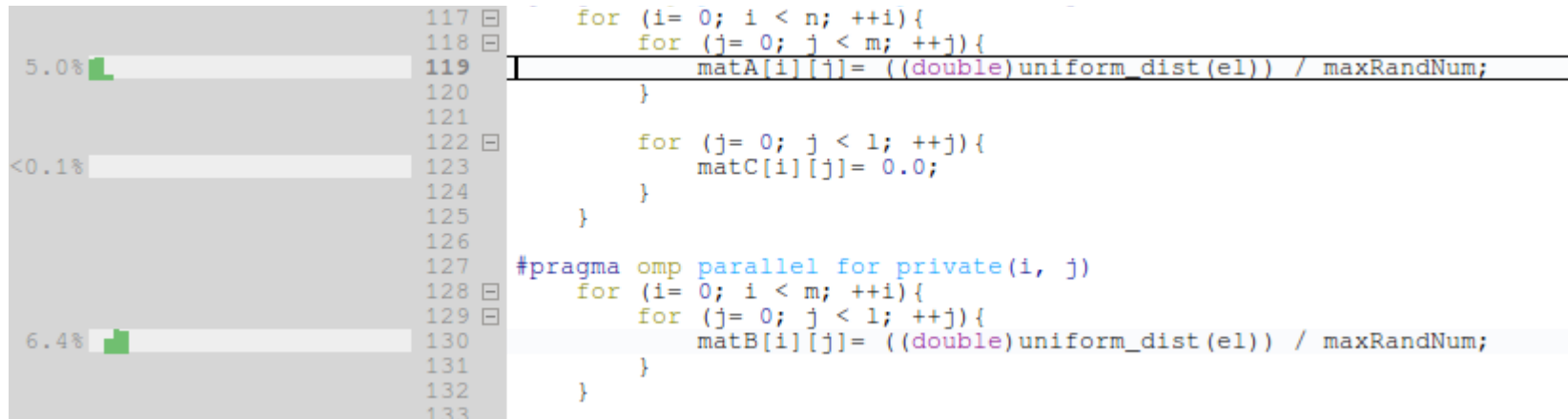
The screenshot shows a performance analysis tool interface with two code snippets and their corresponding performance metrics. The top snippet, lines 112-117, is a parallel loop for populating matrix A, with a performance metric of 37.5%. The bottom snippet, lines 123-128, is a parallel loop for populating matrix B, with a performance metric of 35.6%. Both snippets use the `rand()` function for random data generation.

```
112 #pragma omp parallel for private(i, j)
113     for (i= 0; i < n; ++i){
114         for (j= 0; j < m; ++j){
115             matA[i][j]= ((double)rand()) / RAND_MAX;
116         }
117
118     for (j= 0; j < l; ++j){
119         matC[i][j]= 0.0;
120     }
121 }
122
123 #pragma omp parallel for private(i, j)
124     for (i= 0; i < m; ++i){
125         for (j= 0; j < l; ++j){
126             matB[i][j]= ((double)rand()) / RAND_MAX;
127         }
128     }
```

- Lots of time spent waiting on the kernel in glibc rand function
 - Kernel lock being held means that parallel performance can't be obtained

Matrix-Matrix Multiply: Version 5 (Interlude)

- Update random function to be thread safe implementation (e.g. C++ stdlib functions)



The image shows a code editor with C++ code for matrix generation. On the left, a performance profiler shows three bars: a green bar at 5.0%, a grey bar at <0.1%, and another green bar at 6.4%. The code includes nested loops for initializing matrices A, C, and B. Matrix A is initialized with random values, matrix C is initialized to zero, and matrix B is initialized with random values. A parallel pragma is used for the initialization of matrix B.

```
117 for (i= 0; i < n; ++i){
118     for (j= 0; j < m; ++j){
119         matA[i][j]= ((double)uniform_dist(e1)) / maxRandNum;
120     }
121 }
122 for (j= 0; j < l; ++j){
123     matC[i][j]= 0.0;
124 }
125 }
126
127 #pragma omp parallel for private(i, j)
128 for (i= 0; i < m; ++i){
129     for (j= 0; j < l; ++j){
130         matB[i][j]= ((double)uniform_dist(e1)) / maxRandNum;
131     }
132 }
133 }
```

- Can make use of all cores in parallel

Matrix-Matrix Multiply: Version 6

- Allow compiler to make better decisions on loop parallelism

```
#pragma omp parallel for collapse(2) private(i, j, jj, k, kk)
for (kk= 0; kk < m; kk+= blockSize){
  for(jj= 0; jj < l; jj+= blockSize){
    for(i= 0; i < n; i+= 2){
      for(j= jj; j < min(l, jj + blockSize); j+= 2){
        for(k =kk; k < min(m, kk + blockSize); ++k){
          matC[i][j] += matA[i][k] * matB[k][j];
          matC[i][j+1] += matA[i][k] * matB[k][j+1];
          matC[i+1][j] += matA[i+1][k] * matB[k][j];
          matC[i+1][j+1] += matA[i+1][k] * matB[k][j+1];
        }
      }
    }
  }
}
```

- Allows better use of threads – 8x speed-up over not using collapse directive
- Still greater than 10x slower than Arm Performance Libraries

Matrix-Matrix Multiply: Version 7

- Lots of improvements to be made in the simple implementation of matrix-matrix multiply
 - Consideration of all levels of cache
 - Hardware specific block size
 - Use of vectorisation
- Compiler can auto-vectorize. Does it in this case?
 - Add `-Rpass-analysis=loop-vectorize` for report on vectorization

```
mat_mult_7.cpp:48:21: remark: loop not vectorized: cannot identify array bounds [-Rpass-analysis=loop-vectorize]
                    for(k =kk; k < min(m, kk + blockSize); ++k){
                    ^
mat_mult_7.cpp:47:17: remark: loop not vectorized: could not determine number of loop iterations [-Rpass-analysis=loop-vectorize]
                    for(j= jj; j < min(l, jj + blockSize); j+= 2){
                    ^
```

Further Exercises

- Have a play around with the code provided – can you get vector instructions generated? <https://goo.gl/NN7KsC>
- Consider compiler flags – do your favourite GNU and Intel flags exist for Arm?
- <https://goo.gl/orhZEu> and <https://goo.gl/FKAjwP>
- Consider the performance profiles generated by the Arm Forge tools – what do they tell you about the performance of the executables? <https://goo.gl/PaeA2N>
- Links to extra material in the Tutorial Walkthrough – Final Summary

Hands on - The instructions

Instructions available on Google Docs (<https://goo.gl/mQaMb3>)



Thank You!

Danke!

Merci!

谢谢!

ありがとう!

Gracias!

Kiitos!

감사합니다

धन्यवाद

arm

arm

The Arm trademarks featured in this presentation are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners.

www.arm.com/company/policies/trademarks