Arm® Allinea Studio / Arm Forge Trials Package

Version 1.0

Documentation

101533_0100_00_en

Arm® Allinea Studio / Arm® Forge Trials Package

Documentation

Copyright © 2019 Arm Limited or its affiliates. All rights reserved.

Release Information

Document History

Issue	Date	Confidentiality	Change
0100-00	06 November 2019	Non-Confidential	New document for v1.0

Confidentiality Status

Product Status

The information in this document is Final, that is for a developed product.

Web Address

www.arm.com

Contents

Arm® Allinea Studio / Arm Forge Trials Package Documentation

List of Figures
Arm® Allinea Studio / Arm Forge Trials Package Documentation

Preface

This preface introduces the *Arm® Allinea Studio / Arm Forge Trials Package Documentation*.

It contains the following:

- *About this book* on page 7.

About this book

This book describes how to get started with the Arm® Allinea Studio/Arm® Forge Trials package. It introduces you to the Arm High Performance Computing (HPC) tools, and shows you how to use them to compile, run, analyze, and optimize some example code.

Using this book

This book is organized into the following chapters:

Chapter 1 Package and Requirements
Describes the package contents and the software requirements it needs.

Chapter 2 Tutorial
This tutorial takes you through the steps to compile, run, fix the bug, analyze, and optimize the trial examples.

Chapter 3 Support and Further Resources
Describes the resources that are available for further support.

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

bold
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace` (underlined: mono)
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
Denotes language keywords when used outside example code.

`<and>`
Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Arm Allinea Studio / Arm Forge Trials Package Documentation*.
- The number 101533_0100_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

- *Arm® Developer*.
- *Arm® Information Center*.
- *Arm® Technical Support Knowledge Articles*.
- *Technical Support*.
- *Arm® Glossary*.

Chapter 1
Package and Requirements

Describes the package contents and the software requirements it needs.

It contains the following sections:

1.1 Package Contents

Lists the components of the Arm Allinea Studio/Arm Forge Trials package.

The package contains:

- README.txt

  The README file:
  - Lists the components of the Arm Allinea Studio/Arm Forge Trials package.
  - States the user support statement.
  - States the Non-Confidential Proprietary Notice.
- **license_terms**

  Contains the legal notices:
  - license_agreement.txt
  - redistributables.txt
  - supplementary_terms.txt
  - third_party_licenses.txt
- **/doc/**

  Contains HTML (**/html/index.html**) and PDF versions of the trials package documentation.
- **/src/**
  - **/C/**

    Contains the C example source file and Makefile:
    - `mmult.c`
    - `Makefile`
  - **/F90/**

    Contains the Fortran example source file and Makefile:
    - `mmult.F90`
    - `Makefile`
  - **/Py/**

    Contains the Python example file (with associated C and Fortran source files and Makefiles) and Makefile:
    - `Makefile`
    - `mmult.py`
    - **/C/**
      - `mmult.c`
      - `Makefile`
    - **F90**
      - `mmult.F90`
      - `Makefile`

1.2 Software requirements

Describes the software requirements for the Arm Allinea Studio/Arm Forge Trials package.

Note

This package is for Linux platforms only.

To build and run the application, ensure that:

- You have *downloaded* and *installed* the tools you are trialing.
- You place the trial license in the correct location for the tools:
  - For Arm Compiler for Linux: If you install to the default location, copy the trial license into the `/opt/arm/licenses` directory. Alternatively, if you choose an installation location, copy the trial license into the `licenses` directory of your chosen installation directory.
  - For Arm Forge and Arm Performance Reports: Copy the trial license into the `licenses` directory where the tool is installed.
- An Message Passing Interface (MPI) environment is configured. If you are unsure if you have an MPI environment configured, ask your System Administrator.

  If you do not have an MPI environment configured, and you are using the Arm Compiler for Linux, see the build recipes on how to build *OpenMPI*, *MPICH*, and *MVAPICH*.
- You have installed a Basic Linear Algebra Subprogram (BLAS) library.

  On Arm-based platforms, you can use the *Arm Performance Libraries* that ship with the Arm Compiler for Linux.
- You have loaded the environment module for the tools you are using.
- (Python example only) You have installed Python and its associated libraries:
  - Python 3.6 is installed.
  - *MPI4Py 3.0.2*, *NumPy 1.16.4*, and *SciPy 1.3.0* modules are installed.

  Note

  If you are using the Arm Compiler for Linux, see the build recipe on how to build *Numpy*.

Chapter 2
Tutorial

This tutorial takes you through the steps to compile, run, fix the bug, analyze, and optimize the trial examples.

It contains the following sections:

2.1 The algorithm

Describes the algorithm used in the Arm Allinea Studio/Arm Forge Trials package tutorial.

The application in this tutorial performs the following calculation:

```
C = A * B + C
```

A, B, and C are double-precision square matrices. In this first version, the algorithm is written in parallel using MPI and includes one master process and several worker processes. The application performs the following tasks:

- The master process initializes matrices A, B, and C.
- The master process sends the entire matrix B, along with slices of A and C, to the worker processes.
- The master and worker processes perform the matrix multiplication function on the domain that has been given to the processes, and each process computes a slice of C.
- The master process retrieves all slices of C and puts the results into matrix C.
- The master process writes the results of C in a file.

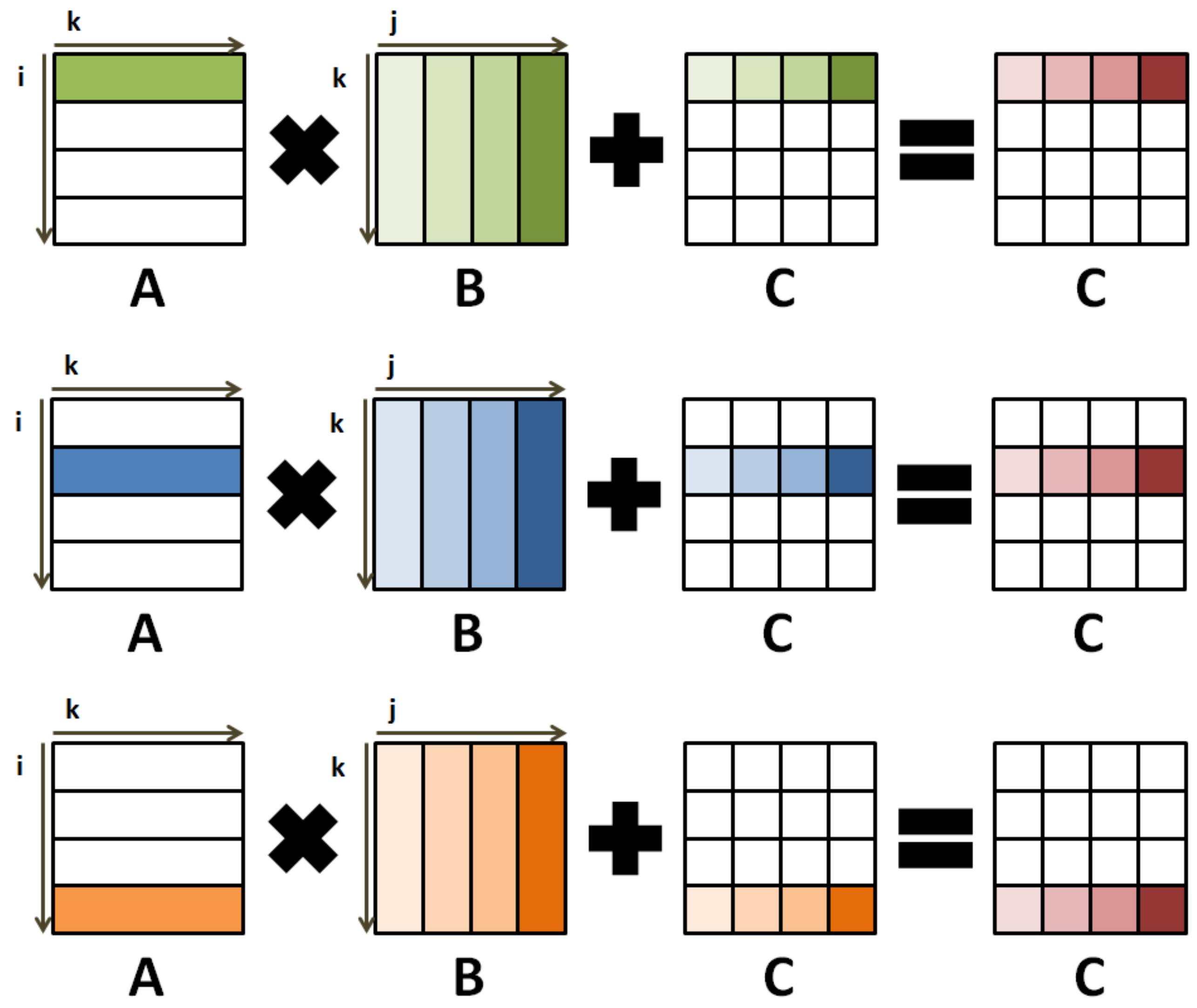


Figure 2-1 Description of the parallel algorithm

The figure shows how all processes contribute in parallel to the creation of the result in matrix C.

This package includes C/C++, Fortran, and Python versions of this application.

2.2 Compile and Run the Code

Instructions to compile and run the example code used in the Arm Allinea Studio/Arm Forge Trials tutorial.

Prerequisites

You must install all the necessary tools as described in *Software requirements* on page 1-11

Procedure

1. Compile the code. To compile all versions of the application, run `make` in the `src` directory:

   ```
   cd src
   make
   ```

   The compiler command and compilation options can be set in file:make.def. The `Makefile` in the `src` directory uses these settings.

   Note

   Alternatively, the C/C++ and Fortran source files can be compiled separately in the `C` and `F90` directories.

   However, the Python example relies on C and F90 kernels that must be compiled using the `Makefile` provided in the `Py` directory.

2. Run the application. To run the C, Fortran, or Python application, navigate to the `C`, `F90`, or `Py` directory that contains the compiled binary. To run the application using eight processes, use:
   - For the `C` and `F90` versions:

   ```
   mpirun -n 8 ./mmult
   ```

   - For the `Py` version:

   ```
   mpirun -n 8 python ./mmult.py
   ```

   Additional arguments can be added to change the size of the matrices, for example 512x512:
   - For the `C` and `F90` versions:

   ```
   mpirun -n 8 ./mmult 512
   ```

   - For the `Py` version:

   ```
   mpirun -n 8 python ./mmult -s 512
   ```

   Note

   By default, the `Py` version runs the C kernel. To run the F90 kernel, use:

   ```
   mpirun -n 8 python ./mmult -k F90
   ```

The application crashes with a ‘Segmentation fault’. This is expected behavior. Each example source file provided in this trials package, has an intentional bug.

Next Steps

To fix the bug, you must debug the code. *Fix the bug* on page 2-15 shows you how to debug the code using Arm DDT, and then how to fix the bug.

2.3 Fix the bug

This topic describes how to identify and fix the bug in each of the example source files with Arm DDT.

Prerequisites

- You must install all the necessary tools as described in *Software requirements* on page 1-11.
- You must complete the instructions in *Compile and Run the Code* on page 2-14.

Procedure

1. Recompile the application with the `-g` debugging flag.

   To make debugging simpler, Arm recommends that you compile without using compiler optimizations. To disable optimizations, add the `-O0` option to the `CFLAGS` variable. In the following examples, the options used on the command line by the `Makefile` can be set in `src/make.def`.

   ```
   CFLAGS = -O0 -g
   ```

   Remove the initial executable with:

   ```
   make clean
   ```

   And recompile with:

   ```
   make
   ```

   Note

   In Fortran, the compiler might display a warning. To display more information at runtime, compile using the “-fcheck=bounds” flag with GCC, or “-Mbounds” using the Arm Compiler for Linux.

2. Navigate to the `C`, `F90`, or `Py` directory containing the application for the version you want to debug.
3. Debug the application with `ddt mpirun` using express launch. For example:

   ```
   ddt mpirun -n 8 ./mmult
   ddt mpirun -n 8 python ./mmult.py
   ```

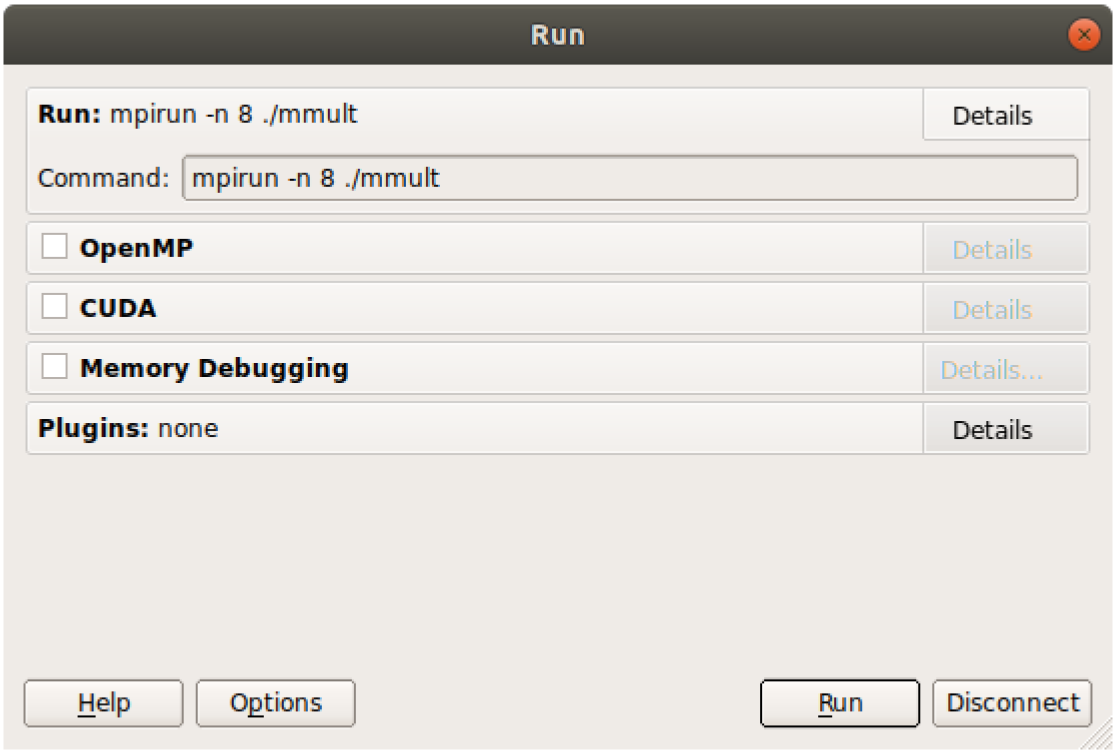
   If your MPI environment does not support express launch, run the `ddt` command:

   ```
   ddt -n 8 ./mmult
   ddt -n 8 python ./mmult.py
   ```

   Note

   `-n 8` tells the debugger to debug using 8 processes. However, the application crashes at any scale.

   Arm DDT starts and displays the **Run** window. If you run the application on a remote system, ensure that you enable X forwarding. To learn more about using X forwarding with Arm Forge, see the *Connecting to a remote system* topic in the Arm Forge user guide.



Run: mpirun -n 8 ./mmult

Command: mpirun -n 8 ./mmult

☐ OpenMP

☐ CUDA

☐ Memory Debugging

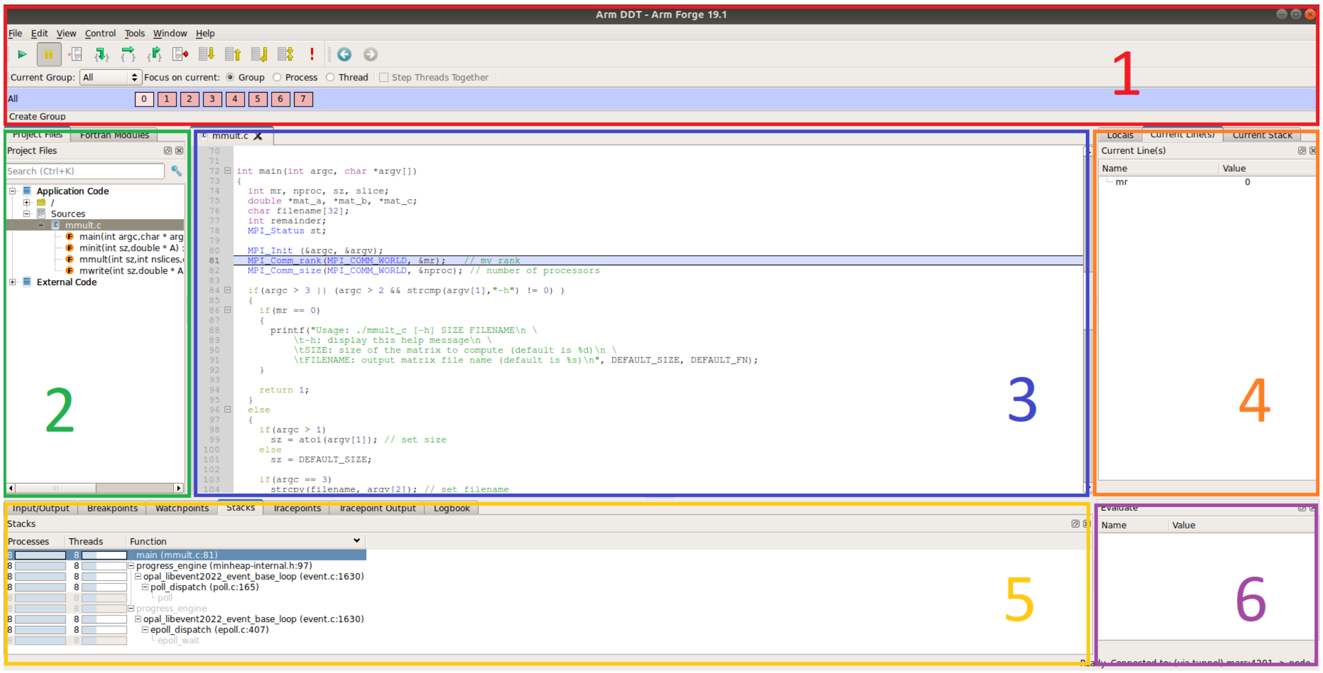
Plugins: none

Figure 2-2 Arm DDT run window

This window displays the debugging session settings.

4. To continue, click **Run**.

   This displays the main debugger window.



Current Group: All | Focus on current: [✓ Group] Process Thread | ☐ Step Threads Together

All 0 1 2 3 4 5 6 7

Create Group

```
int main(int argc, char *argv[])
{
  int mr, nproc, sz, slice;
  double *mat_a, *mat_b, *mat_c;
  char filename[32];
  int remainder;
  MPI_Status st;

  MPI_Init (&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &mr);    // my rank
  MPI_Comm_size(MPI_COMM_WORLD, &nproc); // number of processors

  if(argc > 3 || (argc > 2 && strcmp(argv[1],"-h") != 0) )
  {
    if(mr == 0)
    {
      printf("Usage: ./mmult_c [-h] SIZE FILENAME\n \
          \t-h: display this help message\n \
          \tSIZE: size of the matrix to compute (default is %d)\n \
          \tFILENAME: output matrix file name (default is %s)\n", DEFAULT_SIZE, DEFAULT_FN);
    }

    return 1;
  }
  else
  {
    if(argc > 1)
      sz = atoi(argv[1]); // set size
    else
      sz = DEFAULT_SIZE;

    if(argc == 3)
      strcpy(filename, argv[2]); // set filename
```

Current Line(s)

Name	Value
mr	0

Stacks

```
Processes  Threads  Function
8          8        main (mmult.c:81)
8          8        progress_engine (minheap-internal.h:97)
8          8          opal_libevent2022_event_base_loop (event.c:1630)
8          8            poll_dispatch (poll.c:165)
8          8              poll
8          8        progress_engine
8          8          opal_libevent2022_event_base_loop (event.c:1630)
8          8            epoll_dispatch (epoll.c:407)
8          8              epoll_wait
```

Evaluate

Name	Value

Ready. Connected to: (via tunnel) mars:4201 -> node

Figure 2-3 Arm DDT main window

The interface displays the following:

- Section 1: Commands to play, pause or stop the application and select the processes or threads to inspect
- Section 2: Project source files
- Section 3: Source code view
- Section 4: Current line and program variables
- Section 5: Stack view and breakpoint, tracepoint, and watchpoint settings
- Section 6: Evaluate window for arbitrary expressions

———— **Note** ————

At this stage, the `C` and `F90` versions display the source code of the application.

If Arm DDT does not display the source code, recompile the source file and ensure that the `-g` debugging flag is included on the compile line. For complete instructions on how to compile the examples, see *Compile and Run the Code* on page 2-14.

If you still experience a problem, contact Arm support.

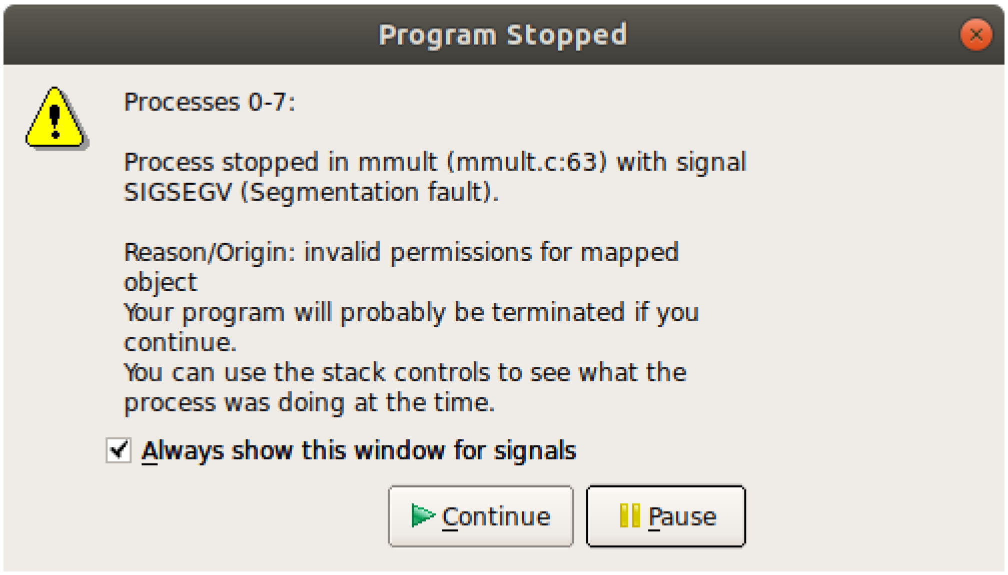
For the `Py` version, no source code displays when the debugger attaches. This is expected when running in the Python interpreter. The C or F90 source code of the kernel displays when the application crashes, if you compile the kernels with `-g`.

————————————

5. To visualize where the application crashes, click **Play** (in the top-left corner):

Figure 2-4 Arm DDT Play button

The debugger stops where the application crashes.


Program Stopped

Processes 0-7:

Process stopped in mmult (mmult.c:63) with signal SIGSEGV (Segmentation fault).

Reason/Origin: invalid permissions for mapped object
Your program will probably be terminated if you continue.
You can use the stack controls to see what the process was doing at the time.

[✓] Always show this window for signals

Figure 2-5 Arm DDT segmentation fault message

6. Click **Pause**.

The source code viewer highlights the line of code where the crash occurs.

In C:

```
res += A[i*sz+k]*B[k*sz*j];
```

In F90:

```
res=A(k,i)*B(j,k+res)
```

This problem is caused by an error in the expression to compute the index of array B, and this results in an out of bound memory access. The bug can be fixed with:

In C:

```
res += A[i*sz+k]*B[k*sz+j];
```

In F90:

```
res=A(k,i)*B(j,k)+res
```

Note

To automatically detect an out of bound memory access with Arm DDT, select the **Memory debugging** box in the **Run** window. Navigate to memory debugging **Details** and enable **Add guard pages to detect out of bound heap accesses**.

For more information, see *Memory debugging* in the Arm Forge user guide.

7. Save the source file: select **File > Save Source File**.
8. Recompile the source file: select **File > Build**.

   By default, **Build** runs `make` in the current directory.

   Note

   To change the **Build** options, select **Configure Build** and set the configuration settings.

9. To run the executable with the fix, select **File > Restart session**.

   You are prompted to restart the application. Click **Yes**.
10. Play the application in the debugger again.

    The application runs without any issues until every process in the program has terminated, and outputs the following when running 8 processes:

```
0: Size of the matrices: 64x64
3: Receiving matrices...
6: Receiving matrices...
2: Receiving matrices...
4: Receiving matrices...
7: Receiving matrices...
1: Receiving matrices...
5: Receiving matrices...
0: Initializing matrices...
0: Sending matrices...
1: Processing...
2: Processing...
3: Processing...
4: Processing...
5: Processing...
6: Processing...
7: Processing...
0: Processing...
1: Sending result matrix...
3: Sending result matrix...
5: Sending result matrix...
7: Sending result matrix...
2: Sending result matrix...
4: Sending result matrix...
6: Sending result matrix...
0: Receiving result matrix...
0: Writing results...
0: Done.
```

When fixed, the application writes the results in the working directory to a file called `res_C.mat`, `res_F90.mat`, or `res_Py.mat` (depending on the version you used). Ensure that your working directory is writable.

Note

To run Arm DDT in non-interactive mode, use `ddt --offline mpirun`:

```
ddt --offline mpirun -n 8 ./mmult
```

The debugger runs in the background of the application and outputs a debugging report:

```
firefox mmult_8p_1n_YYYY-MM-DD_HH-MM.txt
```

YYY-MM-DD_HH-MM corresponds to a timestamp of the report creation date. For more information, see *Offline debugging* in the Arm Forge user guide.

Related references

Chapter 3 Support and Further Resources on page 3-29

2.4 Analyze the behavior

Describes how to analyze the behavior of the example code used in the Arm Allinea Studio/Arm Forge Trials tutorial, and how to check if there is any performance issue with Arm Performance Reports.

Prerequisites

- You must install all the necessary tools as described in *Software requirements* on page 1-11
- You must complete the instructions in *Compile and Run the Code* on page 2-14 and *Fix the bug* on page 2-15.

Procedure

1. Run the application with eight processes on a large test case, for example 3072x3072 matrices:

   ```
   perf-report mpirun -n 8 ./mmult 3072
   ```

   or

   ```
   perf-report mpirun -n 8 python ./mmult.py -s 3072
   ```

   If your MPI environment does not support express launch, run the following command instead:

   ```
   perf-report -n 8 ./mmult 3072
   ```

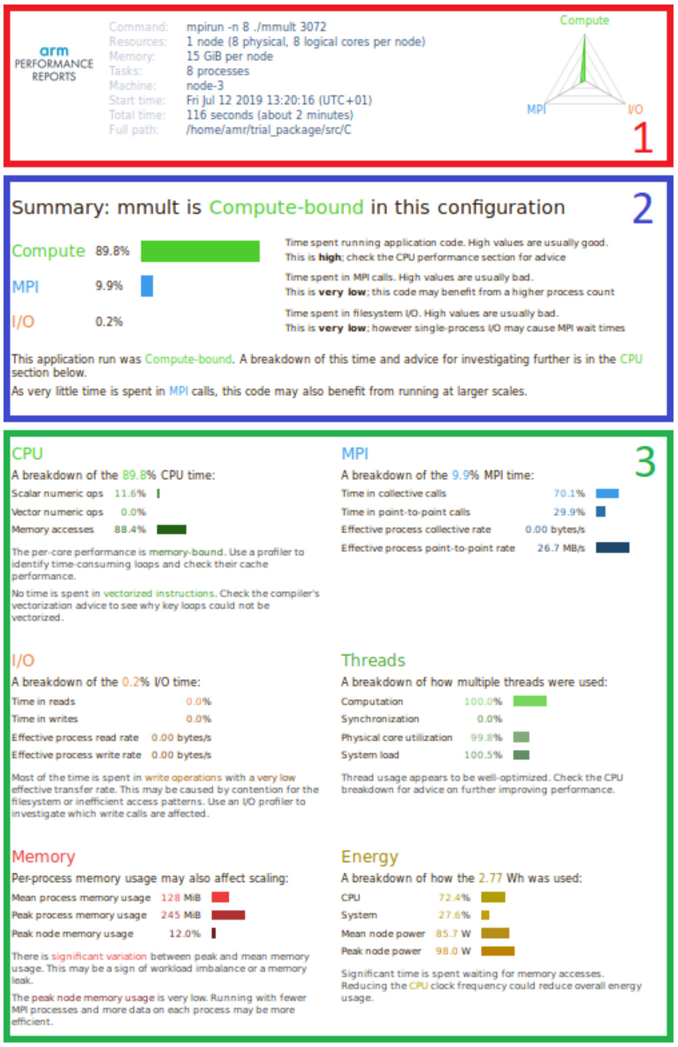
   or

   ```
   perf-report -n 8 python ./mmult.py -s 3072
   ```

   When the execution terminates, Arm Performance Reports creates two files:
   - `mmult_8p_1n_YYYY-MM-DD_HH-MM.txt`
   - `mmult_8p_1n_YYYY-MM-DD_HH-MM.html`

   `YYY-MM-DD_HH-MM` corresponds to a timestamp of the report creation date. The two files contain the same data, in two different formats.
2. To visualize the results, open the HTML file in your web browser (either locally or remotely if you have X forwarding enabled). For example, to use Firefox:

   ```
   firefox mmult_8p_1n_YYYY-MM-DD_HH-MM.txt
   ```



```
Command:     mpirun -n 8 ./mmult 3072
Resources:   1 node (8 physical, 8 logical cores per node)
Memory:      15 GiB per node
Tasks:       8 processes
Machine:     node-3
Start time:  Fri Jul 12 2019 13:20:16 (UTC+01)
Total time:  116 seconds (about 2 minutes)
Full path:   /home/amr/trial_package/src/C
```

Summary: mmult is Compute-bound in this configuration

Compute	89.8%	🟩	Time spent running application code. High values are usually good. This is **high**; check the CPU performance section for advice
MPI	9.9%	🟦	Time spent in MPI calls. High values are usually bad. This is **very low**; this code may benefit from a higher process count
I/O	0.2%		Time spent in filesystem I/O. High values are usually bad. This is **very low**; however single-process I/O may cause MPI wait times

This application run was Compute-bound. A breakdown of this time and advice for investigating further is in the CPU section below.
As very little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU

A breakdown of the 89.8% CPU time:

Scalar numeric ops	11.6%	▮
Vector numeric ops	0.0%	
Memory accesses	88.4%	▬

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

No time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

MPI

A breakdown of the 9.9% MPI time:

Time in collective calls	70.1%	▬
Time in point-to-point calls	29.9%	▮
Effective process collective rate	0.00 bytes/s	
Effective process point-to-point rate	26.7 MB/s	▬

I/O

A breakdown of the 0.2% I/O time:

Time in reads	0.0%
Time in writes	0.0%
Effective process read rate	0.00 bytes/s
Effective process write rate	0.00 bytes/s

Most of the time is spent in write operations with a very low effective transfer rate. This may be caused by contention for the filesystem or inefficient access patterns. Use an I/O profiler to investigate which write calls are affected.

Threads

A breakdown of how multiple threads were used:

Computation	100.0%	▬
Synchronization	0.0%	
Physical core utilization	99.8%	▮
System load	100.5%	▮

Thread usage appears to be well-optimized. Check the CPU breakdown for advice on further improving performance.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	128 MiB	▬
Peak process memory usage	245 MiB	▬
Peak node memory usage	12.0%	▮

There is significant variation between peak and mean memory usage. This may be a sign of workload imbalance or a memory leak.

The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.

Energy

A breakdown of how the 2.77 Wh was used:

CPU	72.4%	▬
System	27.6%	▮
Mean node power	85.7 W	▬
Peak node power	98.0 W	▬

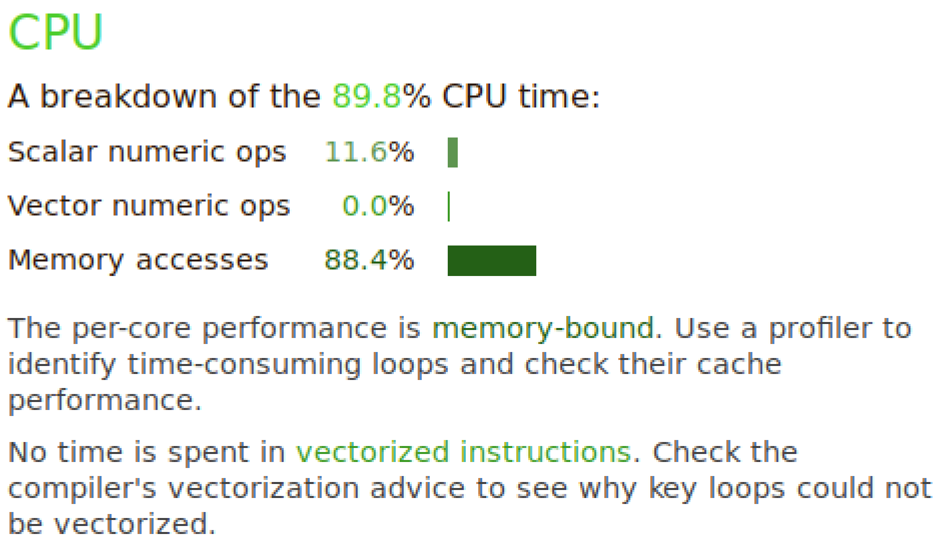
Significant time is spent waiting for memory accesses. Reducing the CPU clock frequency could reduce overall energy usage.

Figure 2-6 Arm Performance Report HTML results

The report shows different sections:

- Section 1: Describes the system settings (including the number of physical and logical cores), the job configuration (including the number of processes and number of nodes) and the execution time.
- Section 2: Shows the amount of time spent in computations (CPU), communications (MPI), and IO.
- Section 3: Shows a breakdown of:
  - — The CPU, MPI, and IO time.
  - — How multiple threads were used.
  - — How the energy was used.

The details of the report will be different and relevant to your system configuration, but the report should indicate that the application is CPU bound. The CPU breakdown section gives more information about the type of instruction run:



CPU

A breakdown of the 89.8% CPU time:

Scalar numeric ops	11.6%	▌
Vector numeric ops	0.0%	│
Memory accesses	88.4%	█

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

No time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

Figure 2-7 Arm Performance Report CPU metrics with no compiler optimizations

The compiler does not perform vectorization. As the report suggests, you can change the behavior by changing your compiler options.

3. To enable the `-Ofast` compiler optimization, edit `src/make.def`:

```
CFLAGS = -Ofast -g
```

Note

On non-x86 architectures, the CPU metrics are different. Instead, the tool reports these metrics:

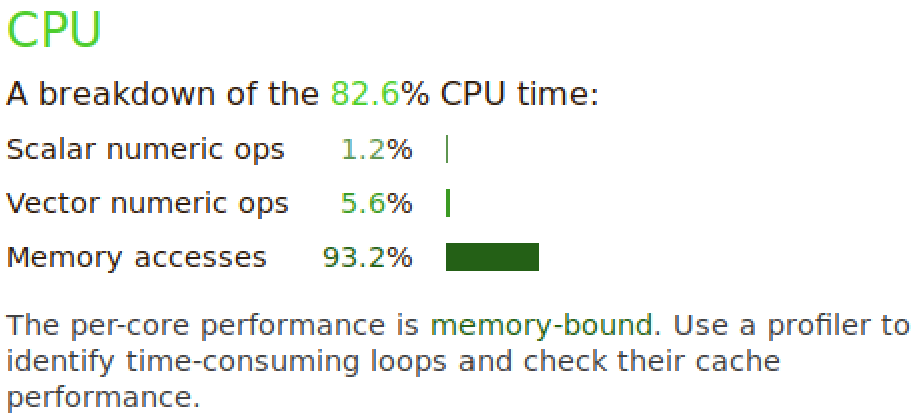
- Cycles per instructions
- Amount of L2 (or L3) cache accesses
- Amount of processor back-end/front-end stalls

Keep these numbers low for better performance.

4. Remove the previous executable, recompile, and run Arm Performance Reports again:

```
make clean
make
perf-report mpirun -n 8 ./mmult 3072
```

The new report shows a performance improvement because the code has been vectorized by the compiler.



CPU

A breakdown of the 82.6% CPU time:

Scalar numeric ops	1.2%	
Vector numeric ops	5.6%	
Memory accesses	93.2%	

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

Figure 2-8 Arm Performance Report CPU metrics with compiler optimizations

2.5 Optimize the code

Describes how to profile and optimize the example code used in the Arm Allinea Studio/Arm Forge Trials tutorial using Arm Performance Reports and Arm MAP. Arm Performance Reports can identify high memory accesses, and Arm MAP can identify the time-consuming loops in the example code.

Prerequisites

- You must install all the necessary tools as described in *Software requirements* on page 1-11
- You must complete the instructions in *Compile and Run the Code* on page 2-14, *Fix the bug* on page 2-15, and *Analyze the behavior* on page 2-20.
- Ensure the code has been compiled with the `-g` debugging flag.

Procedure

1. To profile the code with multiple processes and the 3072x3072 test case, use `map --profile mpirun`. For example:

   ```
   map --profile mpirun -n 8 ./mmult 3072
   ```

   or

   ```
   map --profile mpirun -n 8 python ./mmult.py -s 3072
   ```

   If express launch is not supported for your MPI environment, run `map --profile`:

   ```
   map --profile -n 8 ./mmult 3072
   ```

   ```
   map --profile -n 8 python ./mmult.py -s 3072
   ```

   The `--profile` option runs the profiler in non-interactive mode. When the execution terminates, a profile file is created by Arm MAP:

   ```
   mmult_8p_1n_YYYY-MM-DD_HH-MM.map
   ```

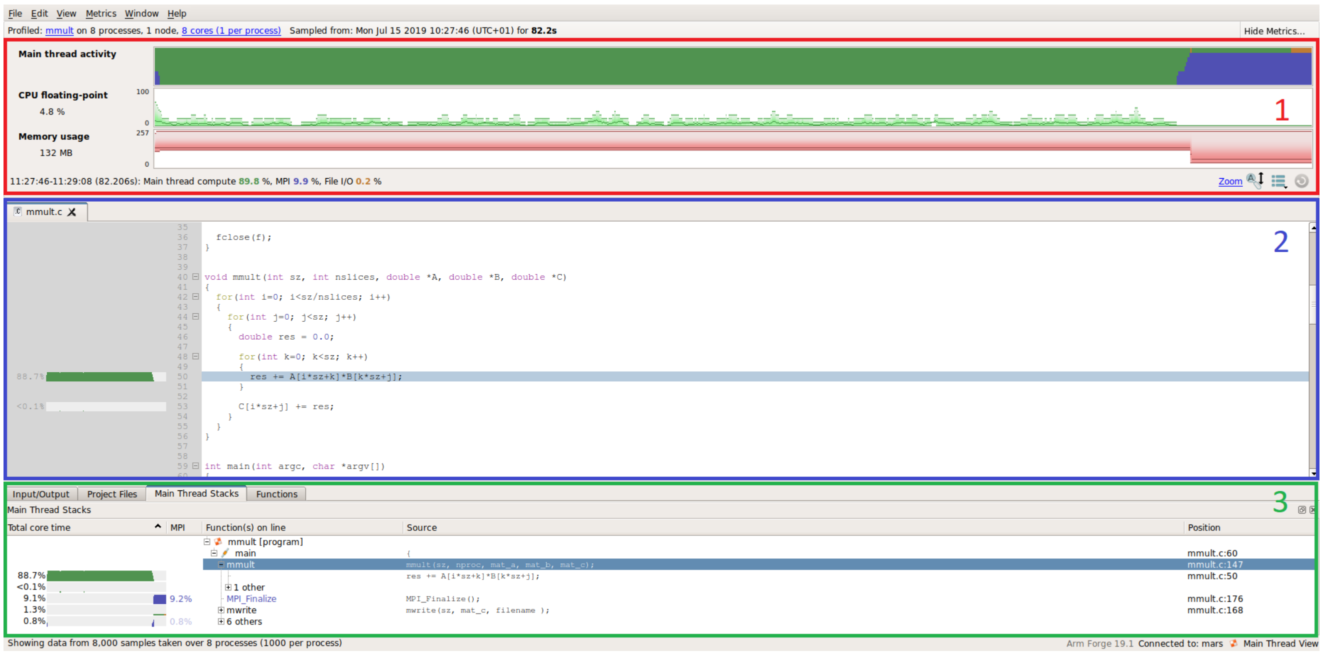
   YYY-MM-DD_HH-MM corresponds to a timestamp of the report creation date.
2. To view the results, run the interactive mode:

   ```
   map mmult_8p_1n_YYYY-MM-DD_HH-MM.map
   ```

   Arm MAP starts.

   Note

   If you are running remotely, make sure X forwarding is enabled.



Profiled: mmult on 8 processes, 1 node, 8 cores (1 per process) Sampled from: Mon Jul 15 2019 10:27:46 (UTC+01) for **82.2s**

11:27:46-11:29:08 (82.206s): Main thread compute 89.8 %, MPI 9.9 %, File I/O 0.2 %

```
  fclose(f);
}


void mmult(int sz, int nslices, double *A, double *B, double *C)
{
  for(int i=0; i<sz/nslices; i++)
  {
    for(int j=0; j<sz; j++)
    {
      double res = 0.0;

      for(int k=0; k<sz; k++)
      {
        res += A[i*sz+k]*B[k*sz+j];        88.7%
      }

      C[i*sz+j] += res;                  <0.1%
    }
  }
}


int main(int argc, char *argv[])
```

Main Thread Stacks

Total core time	MPI	Function(s) on line	Source	Position
		mmult [program]		
		main	{	mmult.c:60
		mmult	mmult(sz, nproc, mat_a, mat_b, mat_c);	mmult.c:147
88.7%			res += A[i*sz+k]*B[k*sz+j];	mmult.c:50
<0.1%		1 other		
9.1%	9.2%	MPI_Finalize	MPI_Finalize();	mmult.c:176
1.3%		mwrite	mwrite(sz, mat_c, filename);	mmult.c:168
0.8%	0.8%	6 others		

Showing data from 8,000 samples taken over 8 processes (1000 per process) Arm Forge 19.1 Connected to: mars Main Thread View

Figure 2-9 Arm MAP main window

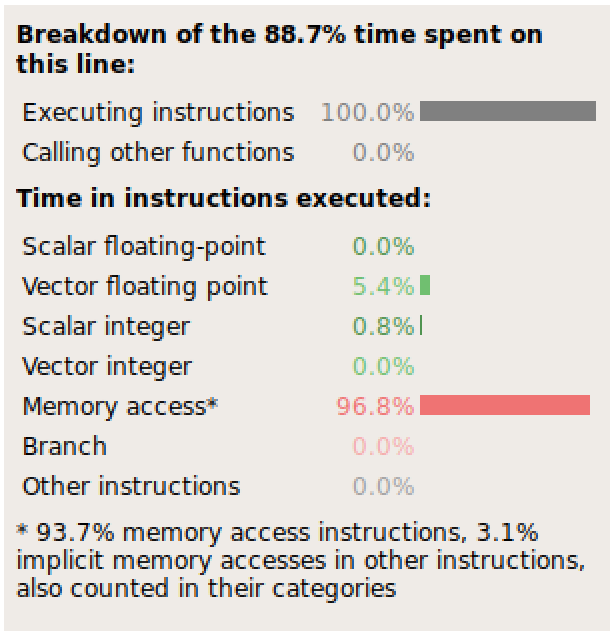
The profile consists of three sections:

- Section 1: The metrics view describes the activity of the processes and threads over time. The colors indicate CPU (green), MPI (blue), or IO (orange) activity. If Python is used, the time spent in the Python interpreter is reported in pink.
- Section 2: The source code view shows annotations about the time spent on each line, and the type of activity performed (CPU, MPI, or IO).
- Section 3: The stack/function view shows aggregated data per call stack or function.

Depending on your system configuration, the details might vary in your results. The profiler indicates that most of the time is spent in one line of the `mmult` function (or when using the Py version, the corresponding calls in the C of F90 version):

- In C: `res += A[i*sz+k]*B[k*sz+j];`
- In F90: `res=A(k,i)*B(j,k)+res`

Select this line of code. The CPU breakdown window appears on the right and shows the following results:



Breakdown of the 88.7% time spent on this line:

Executing instructions	100.0%
Calling other functions	0.0%

Time in instructions executed:

Scalar floating-point	0.0%
Vector floating point	5.4%
Scalar integer	0.8%
Vector integer	0.0%
Memory access*	96.8%
Branch	0.0%
Other instructions	0.0%

* 93.7% memory access instructions, 3.1% implicit memory accesses in other instructions, also counted in their categories

Figure 2-10 Arm MAP line breakdown without optimized memory accesses

The results indicate inefficient memory accesses. The loop nest performs strided accesses to array B. In addition to this, a dependency on intermediate results prevents the compiler vectorizing properly.

Note

On non-x86 architectures, the CPU breakdown is not available. To visualize the high amount of cycles per instructions, L2 (or L3) cache misses, and stalled back-end cycles when the `mmult` function is being executed, instead go to “Metrics” and “CPU instruction”.

---

3. Replace the following code in C:

```
for(int i=0; i<sz/nslices; i++)
{
    for(int j=0; j<sz; j++)
    {
    double res = 0.0;
    for(int k=0; k<sz; k++)
    {
        res += A[i*sz+k]*B[k*sz+j];
    }
    C[i*sz+j] += res;
    }
}
```

with:

```
for(int i=0; i<sz/nslices; i++)
{
    for(int k=0; k<sz; k++)
    {
    for(int j=0; j<sz; j++)
    {
        C[i*sz+j] += A[i*sz+k]*B[k*sz+j];
    }
    }
}
```

And in F90:

```
do i=0,sz/nslices-1
do j=0,sz-1
    res=0.0
    do k=0,sz-1
    res=A(k,i)*B(j,k)+res
    end do
    C(j,i)=res+C(j,i)
```

```
end do
end do
```

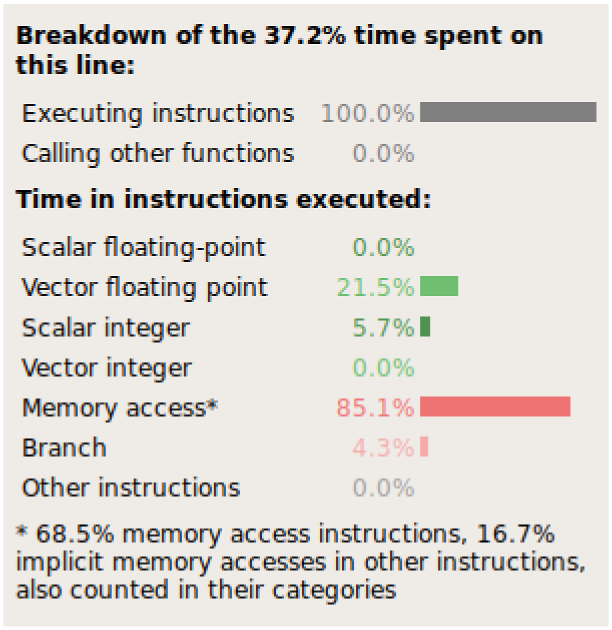
with:

```
do i=0,sz/nslices-1
do k=0,sz-1
    do j=0,sz-1
    C(j,i)=A(k,i)*B(j,k)+C(j,i)
    end do
end do
end do
```

4. Remove the previous executable, recompile, and run Arm MAP again:

```
make clean
make
map --profile -n 8 ./mmult 3072
```

The profiling results show significant performance improvement because of the optimization.



Breakdown of the 37.2% time spent on this line:

Executing instructions	100.0%
Calling other functions	0.0%

Time in instructions executed:

Scalar floating-point	0.0%
Vector floating point	21.5%
Scalar integer	5.7%
Vector integer	0.0%
Memory access*	85.1%
Branch	4.3%
Other instructions	0.0%

* 68.5% memory access instructions, 16.7% implicit memory accesses in other instructions, also counted in their categories

Figure 2-11 Arm MAP line breakdown with optimized memory accesses

Next Steps

To go further and use an optimized version of the matrix multiplication:

- In the C version, call CBLAS instead of `mmult`:

```
#include <cblas.h>
...
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, size/nproc, sz, sz, 1.0, mat_a,
sz, mat_b, sz, 1.0, mat_c, sz);
```

- In the `F90` version, call BLAS instead of `mmult`:

```
call DGEMM('N','N', sz, sz/nproc, sz, 1.0D0, &
           mat_b, sz, &
           mat_a, sz, 1.0D0, &
           mat_c, sz)
```

Make sure you edit `make.def` to include the BLAS header and link to your BLAS library, for instance with OpenBLAS:

```
CFLAGS = -Ofast -g -I/opt/openblas/include
LFLAGS = -L/opt/openblas/lib -lopenblas
```

In the `Py` version, the call to SciPy's DGEMM can be run with the following command:

```
mpirun -n 8 python ./mmult.py -k Py -s 3072
```

Chapter 3
Support and Further Resources

Describes the resources that are available for further support.

It contains the following section:

3.1 Further Resources

Describes the useful resources that are available to support you in using the Arm Allinea Studio tools.

Arm® Forge resources

- *Arm DDT Get started*
- *Arm MAP Get started*
- *Arm Forge User Guide*
- *Arm Forge tutorial videos*

For more information, see the *Arm Forge* web page.

Arm® Performance Reports resources

- *Arm Performance Reports Running with an example program*
- *Arm Performance Reports User Guide*
- *Arm Performance Reports tutorial videos*

For more information, see the *Arm Performance Reports* web page.

Arm® Fortran Compiler resources

- *Arm Fortran Compiler Get started*
- *Arm Fortran Compiler Reference Guide*
- *Troubleshoot your application with Arm Fortran Compiler*

For more information, see the *Arm Fortran Compiler* web page.

Arm® C/C++ Compiler resources

- *Arm C/C++ Compiler Get started*
- *Arm C/C++ Compiler Reference Guide*
- *Troubleshoot your application with Arm C/C++ Compiler*

For more information, see the *Arm C/C++ Compiler* web page.

Arm® Performance Libraries resources

- *Arm Performance Libraries Get Started guide*
- *Arm Performance Libraries Reference Guide*

For more information, see the *Arm Performance Libraries* web page.

Other resources

- *HPC tools documentation*
- *Arm HPC Tools webinars*
- *Application porting and tuning guides*
- *Arm HPC tools*
- *Arm HPC Ecosystem*

If you require support or would like to provide feedback about this trial package, please contact *support-hpc-sw@arm.com* or use our *support request form*.