Assignment 2
Report

Submitted by :
Punit Kumar Jha
NETID: punit2
Graduate Student
Department of Chemistry

October 12, 2017

1 Introduction:

1.1 Objective

This assignment requires us to optimize a dense matrix-matrix multiplication routine and understand its performance behavior. We shall also learn more about PAPI (**P**erformance **A**pplication **P**rogramming **I**nterface) to measure the cache misses and to analyze the performance based on the profiling output.

1.2 PAPI Documentation

The PAPI documentation was read and I learnt about its architecture[1]. The figure below shows PAPI's architecture. PAPI has two layers of architecture:

- The Framework layer – this consists of the API (low level and high level) and machine independent support functions.
- The Component Layer – defines and exports a machine independent interface to the machine dependent functions and data structures. These functions are defined in the components, which may use kernel extensions, operating system calls, or assembly language to access the hardware performance counters on a variety of subsystems.

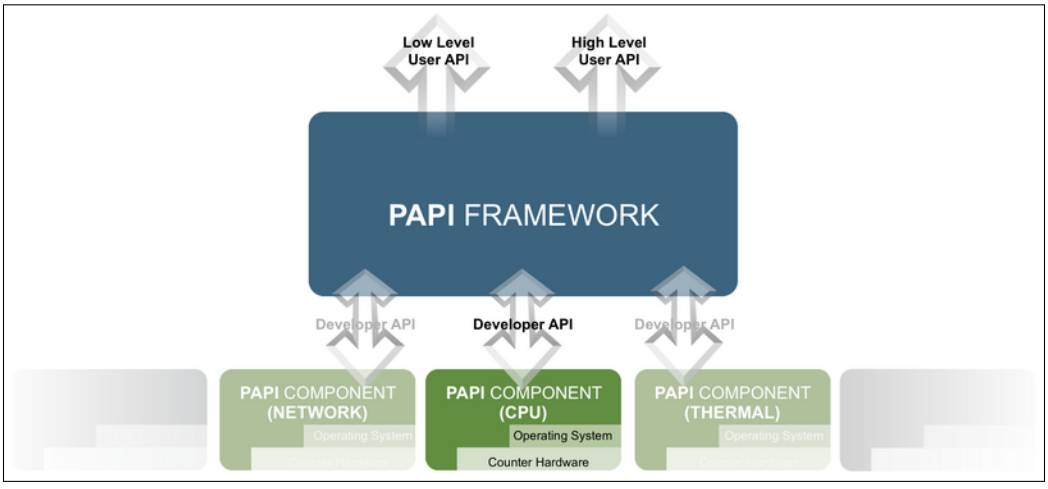

Figure 1: Internal design of the PAPI architecture

2 PART A- Setting up the environment

The basic implementation of the matrix multiply was provided to us in the `basic_seq.c`. The tarball was uploaded to the cluster and the modules were loaded. After extracting the source code and compiling, two binaries were obtained as `cc_without_papi_1000` and `cc_with_papi_1000`. The run script was then submitted on the cluster and the output file was produced as `out-JOBID.txt`.

- **What does the overall binary do ?**
- The overall binaries that are created – `cc_without_papi_1000` and `cc_with_papi_1000` are the executables and they are used to run the program. One of them uses PAPI and the other one does not.
- **In which files is the PAPI profiling implemented ? What are the necessary steps for profiling ?**
- The `src` folder of the tarball provided to us with the `main.c` file. This was the main function which linked the 3 different matrix multiplication implementation in the files `basic_seq.c`, `basic_seq_b.c` and `cache_aware_seq.c`, with the PAPI implementations in the files `papi_helpers.c` and `papi_support.c`. The files `papi_helpers.h` and `papi_support.h` contain the list of functions defined in the `.c` files.
  I went thorough the files and found the following functions defined in the `papi_support.c` files:
  - `void papi_init()` - this function initializes the PAPI system. It uses the `PAPI_library_init()` function to initialize the PAPI library and `PAPI_create_eventset()` function to initilize the event set that are to be profiled..
  - `void papi_prepare_counter(int eventcode)` - This function is used to initialize various counters that one might need. It uses `PAPI_add_event()` function.
  - `void papi_start()`- starts the PAPI counter. It uses `PAPI_start()` function.
  - `papi_stop_and_report()` - this function stops the counter and prints the results. The functions used within it are `PAPI_stop()` and `PAPI_reset()`

  The following functions defined in the `papi_helpers.c` file:
  - `void papi_error_to_string()` - this function returns a string description of the given error code. This function has a list of different PAPI errors that are listed such as - `PAPI_EINVAL`, `PAPI_ENOMEN` etc.
  - `void handle_error()` - this function prints error message and exits.
  - `void check_error()` - this function is used to wrap the calls to PAPI libraries.
  - `double current_time()` - this function returns the current time using the gettimeofday() function of C++.

  Finally the `main.c` files links and uses all the above mention functions to measure the events of interest using PAPI. In our problem we have to measure the L2 data cache misses and accesses these events/counters are defined in the file as

```
#ifndef NOPAPI
papi_prepare_counter(PAPI_L2_DCM);
papi_prepare_counter(PAPI_L2_DCA);
#endif
```

- **What are the events that are being profiled by default ?**

- Events are occurrences of specific signals related to a processors function. Hardware performance counters exist as a small set of registers that count events, such as cache misses and floating point operations while the program executes on the processor. Each processor has a number of events that are native to that architecture.

  - **Native Events** –Native event codes and names are platform dependent, so native codes for one platform are not likely to work for any other platform. To determine the native events for your platform, see the native event lists for the various platforms in the processor architecture manual. The command `papi_native_avail` provides insight into the names of the native events for a specific platform.
  - **Preset Events** -Preset events, also called predefined events, are a common set of events which are considered relevant and useful for application performance optimization. These events are found in many CPUs that provide performance counters and give access to the memory hierarchy, cache coherence protocol events, cycle and instruction counts, functional unit, and pipeline status. For example, Total Cycles (in user mode) – `PAPI_TOT_CYC`. PAPI also supports presets events that may be derived from the underlying hardware metrics. Eg. - Total L1 Cache Misses (`PAPI_L1_TCM`) might be the sum of L1 Data Misses and L1 Instruction Misses on a given platform. A preset can be either directly available as a single counter, derived using a combination of counters, or unavailable on any particular platform. The command `papi_avail` provides insight into the names of the native events for a specific platform.

- **Is there any performance difference between the two compiled binaries ?**

- Printed below is the output file that was produced when a N=1000 and M=1000 matrix multiplication was carried out using all three versions, i.e,
  Crude - simple matrix multiplication
  Basic - matrix multiplication using a temporary variable
  Cache Aware - matrix multiplication using tiling
  we see that there is not much performance difference between the two complied binaries except for the extra PAPI events reported by one binary.

```
Running code without PAPI...
Running basic sequential version to save the results.
Running the basic version.
Execution time: 4.7724872
Running your better basic version.
Execution time: 3.5855770
GOOD -- Test passed.
Running your cache-aware version.
Execution time: 3.5257850
GOOD -- Test passed.
Running code with PAPI...
Intializing PAPI.
Running basic sequential version to save the results.
Prepared counter: 'PAPI_L2_DCM'.
Prepared counter: 'PAPI_L2_DCA'.
Running the basic version.
Starting PAPI counters.
Execution time: 4.7697971
Event 1: 125181617
```

```
Event 2: 1126171611
Resetting PAPI.
Running your better basic version.
Starting PAPI counters.
Execution time: 3.5903232
Event 1: 125158537 --cache misses
Event 2: 1126985285 --cache access
Resetting PAPI.
GOOD -- Test passed.
Running your cache-aware version.
Starting PAPI counters.
Execution time: 3.5301442
Event 1: 113252738
Event 2: 128046573
Resetting PAPI.
GOOD -- Test passed.
```

3 PART B- Simple Improvement

Since the basic version was inefficient due to unnecessary accesses to the C array a better version using a temporary variable was implemented. The code is presented below.

```
void basic_better_MM()
  {
      int i, j, k;
      for (i = 0; i < N; ++i) {
          for (j = 0; j < N; ++j) {
              double temp=0;
              for (k = 0; k < M; ++k) {
                   temp+=A[i][k]*B[k][j];
              }
              C[i][j]=temp;
          }
      }
  }
```

Table 1: Time taken for multiplication by the Crude implementation:

			Time taken	
	N value	M value	With PAPI	NO PAPI
1.	N=100	M=100	0.0045631	0.0044260
2.	N=100	M=1000		
3.	N=1000	M=100		
4.	N=1000	M=1000	4.7791979	4.7811692
5.	N=2000	M=2000	63.3099399	63.7706439
6.	N=3000	M=3000	178.6862819	178.9405808

Table 2: L2 Data Cache Access and Misses Crude implementation:

	N value	M value	Data Cache Access	Data Cache Misses
1.	N=100	M=100	126514	804
2.	N=100	M=1000		
3.	N=1000	M=100		
4.	N=1000	M=1000	1126056044	125181301
5.	N=2000	M=2000	8556049395	4126088874
6.	N=3000	M=3000	30443525692	3442558871

Table 3: Time taken for multiplication by the basic improvement implementation:

			Time taken	
	N value	M value	With PAPI	NO PAPI
1.	N=100	M=100	0.0034800	0.0035141
2.	N=100	M=1000		
3.	N=1000	M=100		
4.	N=1000	M=1000	3.4414601	3.6110849
5.	N=2000	M=2000	42.0729799	42.1395299
6.	N=3000	M=3000	129.6551239	129.6125619

Table 4: L2 Data Cache Access and Misses basic improvement implementation:

	N value	M value	Data Cache Access	Data Cache Misses
1.	N=100	M=100	126378	714
2.	N=100	M=1000		
3.	N=1000	M=100		
4.	N=1000	M=1000	1127480702	125151938
5.	N=2000	M=2000	8560710341	4127705751
6.	N=3000	M=3000	30458238058	3438190494

3.0.1 Observations on Simple Improvemnent Part - B

- We note that using a temporary variable decreases the time of multiplication. We also note that the cache misses and accesses are significantly decreased as well (exact numbers are shown in the table above).
- Each temp is accessed $\approx$ NM times and has a constant reuse distance of $\mathcal{O}1$.
- temp is kept in a register - so number of loads are reduced.
- This transformation is usually done by the complier itself.

4 PART C- Cache-aware tiling implementation

Using the better implemented version in Part B , the tiled version was implemented. The code is shown below. It can be seen from the code below that every dimension was tiled. Different experiments were run using different matrix sizes and the results are shown below:

```
void cache_aware_MM()
  {
          int i,j,k,ii,jj,kk;
          int n=5;
          int T=200;
          for( ii=0; ii<n; ii++)
          {
                  for( jj=0; jj<n ; jj++)
                  {
                          for( kk=0; kk<n ; kk++)
                          {
                                  for(i=ii*T; i<(ii+1)*T; i++)
                                  {
                                          for(j=jj*T; j<(jj+1)*T; j++)
                                          {
                                                  double temp=C[i][j];
                                                  for(k=kk*T; k<(kk+1)*T; k++)
                                                  {
                                                          temp+=A[i][k]*B[k][j];
                                                  }
                                                  C[i][j]=temp;
                                          }
                                  }
                          }
                  }
          }
    }
```

Table 5: Time taken for multiplication by the cache aware implementation using different tile lengths for N=M=1000:

			Time taken	
N=M=1000 Serial No.	T value	n value	With PAPI	NO PAPI
1.	200	5	3.5301442	3.5257850
2.	500	2	3.5691800	3.5667129
3.	250	4	3.5296502	3.5297809
4.	100	10	3.4414601	3.4301271
5.	50	20	3.5682349	3.6233051

Table 6: L2 Data Cache Access and Misses cache aware implementation:

	N value	M value	Data Cache Access	Data Cache Misses
1.	N=100	M=100	17317	702
2.	N=100	M=1000		
3.	N=1000	M=100		
4.	N=1000	M=1000	131152476	2810691
5.	N=2000	M=2000	1054665319	22043963
6.	N=3000	M=3000	3551211229	77537256

Table 7: Time taken for multiplication by the cache aware implementation:

			Time taken	
	N value	M value	With PAPI	NO PAPI
1.	N=100	M=100	T=10 n=10 time=0.0038178	T=10 n=10 time=0.0038218
2.	N=100	M=1000	T= n= time=	T= n= time=
3.	N=1000	M=100	T= n= time=	T= n= time=
4.	N=1000	M=1000	T=100 n=10 time= 3.4414601	T=100 n=10 time=3.4301271
5.	N=2000	M=2000	T=100 n=20 time=27.6909170	T=100 n=20 time=27.5702009
6.	N=3000	M=3000	T=100 n=30 time= 93.4036939	T=100 n=30 time=293.0013258

4.0.1 Observations on Cache Aware tiling implementation

- We note that compared to the basic improved version, the tiled version is faster – takes significantly less time.
- This version also has significantly less cache access and misses (except when the tile size for N= 1000 M=1000 is T= 50 and n=20).
- The best tile size for N= 1000 M=1000 is T= 100 and n=10.
- The improvement is performance is related to the length of the tile. The best performance is for the case where the whole matrix B is cache. The longer the tile the better the pipelining and prefectching.
- Tiling in three dimensions – computes n^3 products of $N \times N$ matrices, where n is the number of tiles.
- If the cache size is $\approx N^2$ then we have $\approx 3n^3N^2 = \frac{N^3}{\sqrt{cachesize}}$ cache misses
- Tilling is three dimension is a little better that tiling in two dimensions but usually not worth it.

4.1 Modifying PAPI to measure the L3 Cache misses rate

Table 8: L3 Data Cache Access and Misses cache aware implementation:

	N value	M value	Data Cache Access	Data Cache Misses
1.	N=3000	M=3000	172157010	9448272

4.1.1 Observations on L3 Cache misses rate

- To arrive at the above results first of all `papi_avail` command was used. This listed the preset PAPI event that are on the system.
- Then the `main.c` file was modified and the following lines were added to it:

```
#ifndef NOPAPI
papi_prepare_counter(PAPI_L3_TCM);
papi_prepare_counter(PAPI_L3_DCA);
#endif
```

5 PART D- Compiler Optimization

Table 9: Comparison of time taken for multiplication by different implementations at -O3 level of optimization. For the results below T=100 was used for all tilings and n=10, 20 or 30 was chosen depending on the size of the matrix.

			-O3 optimization time					
			Crude		Basic		Cache Aware	
	N value	M value	With PAPI	No PAPI	With PAPI	No PAPI	With PAPI	No PAPI
1.	N=100	M=100	0.0003612	0.0003531	0.0003181	0.0003340	0.0005121	0.0004919
2.	N=1000	M=1000	0.2817860	0.2885530	0.5981081	0.6018870	0.3455169	0.3448529
3.	N=2000	M=2000	2.3121450	2.3758669	8.7332199	7.9068670	2.8418751	2.8423440
4.	N=3000	M=3000	8.6348660	8.6550109	43.6038430	43.7560091	12.5859420	12.6247201

Table 10: Comparison of L2 data cache misses and accesses accross different implementations at -O3 level of optimization. For the results below T=100 was used for all tilings and n=10, 20 or 30 was chosen depending on the size of the matrix.

			-O3 optimization time					
			Crude		Basic		Cache Aware	
	N value	M value	Access	Misses	Access	Misses	Access	Misses
1.	N=100	M=100	Access	Misses	Access	Misses	Access	Misses
2.	N=1000	M=1000	3149065	124914427	146794128	559434909	5015029	132121086
3.	N=2000	M=2000	983552487	45565255	4288455775	2692143155	1065841511	77781243
4.	N=3000	M=3000	3749893051	81274720	16827278230	6912341037	5460835277	490296600

5.0.1 Observations on Complier Optimizations Part -D

- The loops are vectorized so the multiplication time is significantly reduced.
- Loop unrolling are done by the compiler – as stated in the compiler reports that were generated and hence the time and cache misses are significantly reduced by the -O3 level of optimzation.

6 References

[1] http://icl.cs.utk.edu/projects/papi/wiki/PAPIC:Overview.