ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)
Volume 5, Issue 4, April 2016

Performance Analysis of Jacobi solver

Swapnil Kulkarni

Abstract— **This paper deals with PDE solvers by Jacobi method. The strategy to overcome limitation due to the size of the cache is discussed. The loop optimization used in this case to achieve better performance is called loop blocking. The 1D and 2D blocking methods for Jacobi solver is the focus of this work.**

Index Terms—**Data reuse, Operational Intensity, Roofline line.**

I. Introduction

The performance of any parallel scientific application depends on: 1) numerical methods involved in the application 2) complexity of the numerical algorithms and 3) exploiting capability of the underlying hardware by the application. The first two are related to the understanding of the numerical method, its implementation and the inherent parallelism. The last one requires the programmer to have the knowledge of the computer architecture in order to benefit the most. Putting all together to ensure better scaling of the application is the heart of scientific computing. With the emergence of multicore architecture, there is focus on the understanding of the way data is fetched from the memory. The emphasis is on data locality and data reuse to enhance parallelism to overcome the latency across different hierarchical storage (memory and cache). This calls for an understanding of the data access for performing the actual computation that is required of the scientific application. This work focuses on roofline model to understand the behavior of Jacobi kernel (PDE solver) on the underlying architecture

II. BACKGROUND

A. Peak performance

The peak performance is what the system is theoretically able to deliver. But the system will not give the performance above the peak performance; it is just upper the limit of performance. It varies according to different characteristics of the system. The sustained performance can be defined as the performance which a program delivers over the entire run time. It is the performance which the system will deliver practice.

Manuscript received April, 2016*.*

Swapnil Kulkarni*, Department of Computer Science and Engineering, Vel Tech Dr. RR & Dr. SR Technical University Avadi, TamilNadu, India 7709577907, Chennai, India.*

The gap between peak performance and sustained performance are due to cycles lost due to data access (latency). Latency is the time taken from the issue of a memory request to the time the data is available to the processor. Consider the example of Intel Sandy Bridge in which data transfer latency due to data transfer from main memory to L3 cache, from L3 to L2, from L2-L1, from L1 to register are 26-31 cycles,175-350 cycles, 12 cycles, and 4 cycles respectively. So cache hides the latency. Caches are small and fast memory elements between the processor and DRAM. This memory acts as a low-latency high-bandwidth storage. Writing cache friendly programming will improve the cache hit ratio which will discuss in section.

B. Cache Concepts

Cache Hit: When program needs a data a first look for the data in the current cache block, if present then we have what is called a cache hit .

Cache Miss: When program needs data which is not present in current cache block then we have cache miss. Here we have to bring the data in cache.

3Cs of cache: 1) Compulsory: On the first access to a block; the block must be brought into the cache; also called cold start misses, or first reference misses. 2) Capacity: Occur because blocks are being discarded from cache because cache cannot contain all blocks needed for program execution. 3) Conflict: In the case of set associative or direct mapped block placement strategies, conflict misses occur when several blocks are mapped to the same set or block frame.

The most commonly used replacement strategies for today's caches are random and least recently used (LRU). The random replacement strategy chooses a random cache line from the set of eligible cache lines to be replaced. The LRU strategy replaces the block which has not been accessed for the longest time [7].

C. Locality of reference

There are two types of locality: temporal and spatial locality. Temporal locality: A memory location is referenced once is likely to be referenced again multiple times in the near future. The repeated access of data during loop iteration is an example of temporal locality. Spatial locality: If a memory location is referenced once, then the program is likely to reference a nearby memory location in the near future [7]. Data are accessed in blocks of cache line. The access of elements of matrix which are stored in row-major order is an example of spatial locality.

D. *Roofline Model*

The roofline model gives useful information for application developer in assessing the performance envelope for the application. The performance of an application depends on the amount of computation as well as the data access (read/write) to enable the computation. This can be captured in a quantitative way through operational intensity. The operational intensity is specified as ratio of the number of floating-point operations in the program (kernel) to the number of memory access to fetch the data for computation. It is also known as arithmetic intensity [1]. To know about roofline model we must know about peak performance of the system and its memory bandwidth. It gives useful information about the application or numerical kernel whether it is compute bound or memory bound. The peak floating point performance can be found using hardware specification. The roofline model ties together the floating point performance, memory performance and operational intensity. The memory performance can be measured using the stream benchmark. By running the stream benchmark we get best memory performance which include prefetching and data alignment. The roofline model [1] is a plot usually given on a log-log scale with operational intensity (Flops/byte) on the x-axis versus the floating point performance (Flops/sec) on the y-axis.

III. PDE SOLVER

Partial differential equation (PDE) solvers are employed by a large faction of scientific application in diverse areas such as heat diffusion, electromagnetics, and fluid dynamics. These applications are often implemented using iterative finite difference method. The PDE solvers are a class of iterative kernels which update array elements according to some fixed pattern called stencil.

Laplace equation is second order partial differential equation named after Pierre-Simon Laplace who first studied its properties. In two dimensions, U(x, y) satisfies the partial differential equation,

$\partial^2U/\partial x^2+\partial^2U/\partial y^2=0$,
with the boundary conditions,
$U(x,0) = 0$, $U(x,1) = \sin(2\pi x)$,
$U(0,y) = 0$, $U(1,y) = -\sin(\pi y)$

Discretize the domain, M x M grid with $\Delta x=\Delta y=1/(M+1)$
$x=i\Delta x$,, i € [0,M+1], y=j, Δy j € [0,M+1]. Using finite differences method, the differential equation now has the form,

$$U_{i,j} = (U_{i-1,j}+U_{i+1,j}+U_{i,j-1}+U_{i,j+1})/4 \text{ for all i,,j.}$$

The solution $U_{i;\,j}$ at any point (i; j) in the discretized domain is obtained from its neighboring points. This is known as 5-point stencil which is shown in Fig.1. This is the discrete form of the Laplace equation which will be used to obtain the solution iteratively using the given boundary conditions (values of U(x; y) on the boundaries) [8]. The $U_{i;\,j}$ evaluated in the previous iteration are used to calculate the solution for the next iteration. The solution obtained by this approach is called Gauss Jacobi method (PDE solver).

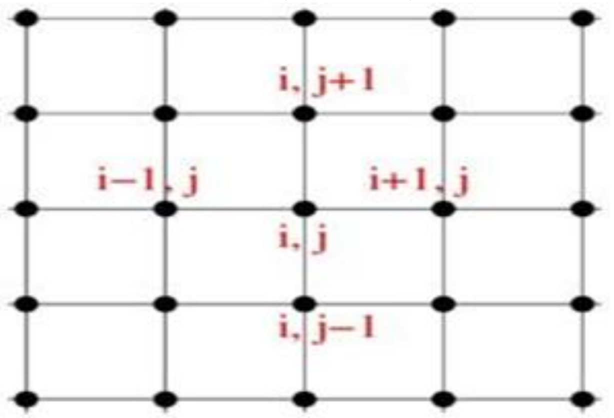

Figure 1: Computational grid (with 5-point stencil)

The Fig.2 shows the numerical solution to Laplace equation by this method.

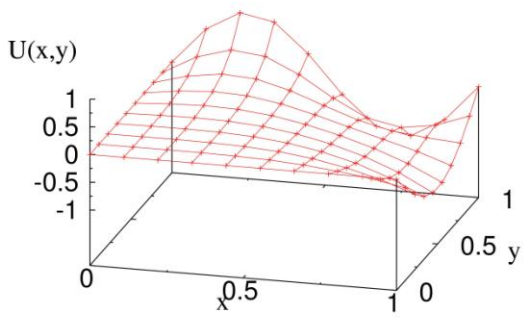

Figure 2: Solution to Laplace Equation

IV. COMPUTATION AND IMPLEMENTATION

A. *Operational Intensity of Jacobi kernel*

The computation of the solution at one point on the grid (using the Jacobi Kernel) involves 4 floating point operations (3 additions + 1 scaling), thus the workload is $W = 4n^2$ for the computation on the entire grid of size nxn. The number of memory access for computing solution at one point on the grid is 5 (4 read + 1 write), total memory access for the computation on the entire grid is $Q = 5n^2$. Thus the operational intensity for Jacobi kernel I = W (n) = Q(n) = 0:8

B. *Roofline model for the Intel Sandy Bridge*

The Intel Sandy Bridge is chosen as representative of the modern multicore architecture. This processor has eight cores (numbered 0-7) on a die. Each of these CPUs/cores have access to independent 32 KBytes of level 1 cache and 256 KBytes of level 2 cache respectively. The level 3 cache of 20 MBytes capacity is shared by these eight cores (mapped to socket: 0/1). We can plot the roofline model for the Intel Sandy Bridge. The memory bandwidth measured using stream benchmark is 34.45GB/s, and peak performance for the double precision is 173GF/s The *sloped line* (with unit slope) is the envelope of the points obtained (for different operational intensities) by multiplying the memory bandwidth (GBytes/s) and operational intensity (Flops/Byte) to obtain performance (GFlops/s) for a given operational intensity (kernel). The *horizontal line* indicates the peak performance for double precision operations. These two lines intersect and the *minimum of the performance* about the point of intersection (on the operational intensity axes) gives the roofline envelope. The following Fig 3 about roofline model for the Intel Sandy Bridge processor.

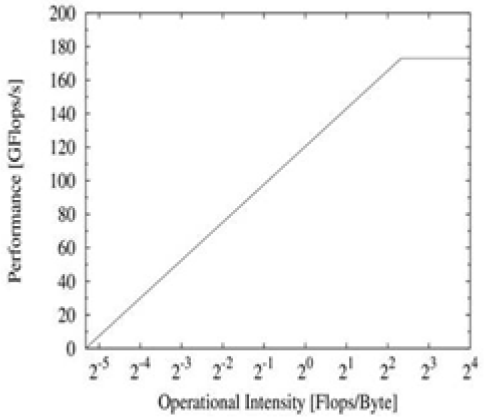

Figure 3: Roofline model Intel Sandy Bridge

The x-axis is operational intensity which is Flops/byte and y-axis is attainable floating point performance which is Flops/sec. The performance of application or kernel can be no higher than the horizontal line since it is hardware limit. The roofline model sets an upper bound on performance of any kernel or application which depends on its operational intensity. If application or kernel according to its operational intensity hits the roof in the slanted part of roof then its performance is ultimately limited by memory bandwidth (memory bound) or if it hits the flat part of roof then its performance is ultimately limited by computational bounded. *Roof line model varies according to the system architectures.*

C. *Data layout*

An illustration of the data layout used in the solver for, say n = 8, is shown in below Fig.4.

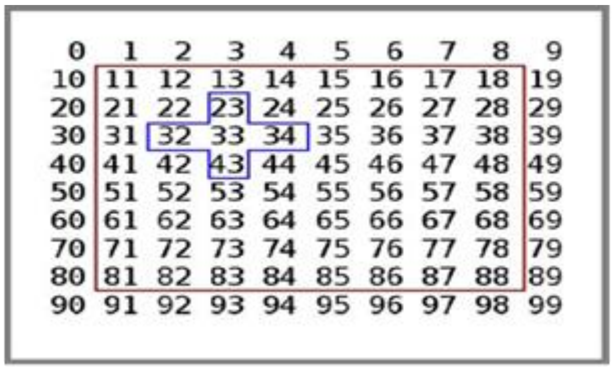

Figure 4: Data layout

The number of grid points in the computational domain is n^2. So the total number of gird points required for solving the Laplace equation is $(n + 2)^2$ along with the points on the boundary. Here discretization along the x and y axis, respectively is 1/(n + 1). The computational grid is bounded on top by grid points along 11- 18 and on the bottom by 18-88 grid points. The left and right bounds of domain are the segments connecting grid points 11- 81 and 18-88. Since the two dimensional grid points are stored in one dimensions, in order to compute the solution at any grid point, which require the left, right, up, down, neighbours, are accessed with offsets -1, +1, (n + 2), (n + 2) respectively. For example, grid point 33 has grid points 32 and 34 has left and right neighbours. The grid points 23 and 43 are up and down neighbours of grid point 33. The following code segment is the implementation of the data layout illustrated above. The offsets for the neighbours are: -1 for the left and +1 for the right neighbours. The offsets for the up and down neighbours are -(n + 2) and +(n + 2) respectively. The code snippet of the implementation of one dimensional data layout described above. The arrays uo and ui store the solution of the current iteration and previous iteration respectively. The following code segment is the implementation of data layout describe above.

```
//Code:
 for (iter=0;iter<itmax;iter++){
       for (i=1;i<=n;i++) {
               for (j=1;j<=n;j++) {
                   b=i*(n+2)+j;
                   uo[b]=0.25*(ui[b-1]+ui[b+1]+ui[b-(n+2)]+
                               ui[b+(n+2)]);
                 }
         }
  }
```

D. *Data reuse*

Data along the 'x' direction are stored in contiguous memory locations, thus ensuring spatial locality of reference. The data scan for compute grid is row by row (stride-1) shown in Fig.5. *The previous row is shown by blue row and current row is shown by red row.* The left, right and top neighbours of grid points of current row were accessed in computing solution at the grid points of previous row and hence available in cache (data reuse). Only bottom the neighbour needs to be fetched from memory.

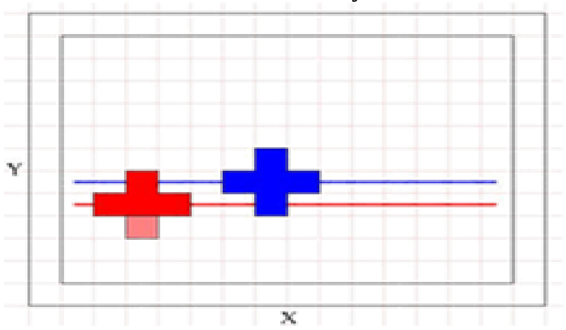

Figure 5: Scanning of the computational grid

The performance of this kernel de-pends on the effective reuse of data in cache. To compute the solutions at grid points along the row (which is the current row), we require the solutions across the three rows; the previous row, the current row, and the next row. As discussed earlier, the Jacobi kernel requires two computational girds (one for read: solution from previous iteration and other one for write: to store the result of current iteration). The L1 cache size requirement should be at-least three rows of the computational grid. For a grid with the inner loop 'i' of extent 'imax', on architecture with cache line size of 64 bytes (8 floats) the limit on imax can be computed for a given L1 cache size. For Intel Sandy Bridge, which has L1 cache size 32KB, imax=32*1024/2*3*8=683. This limiting value is nothing but the number of elements of float type which are stored in L1 cache. This will pose constraint on the problem size to be studied to achieve better performance. Loop tiling ensures better data reuse as well as relaxes constraint of problem size which will discuss in next section.

E. Loop Blocking

Loop blocking [7] is primarily used to improve temporal locality by enhancing the reuse of data in cache and reducing the number of cache capacity misses.. Tiling a single loop replaces it by a pair of loops. The inner loop of the new loop nest traverses a block (tile) of the original iteration space with the same increment as the original loop. The outer loop traverses the original iteration space with an increment equal to the size of the block which is traversed by the inner loop. Thus, the outer loop feeds blocks of the whole iteration space to the inner loop which then executes them step by step.

```
//non-blocked code segment:
for(j=1;j<jmax;j++){
    for(i=1;i<imax;i++){
       u[j][i]=0.25*(u[j][i-1]+u[j][i+1]+u[j-1][i]+u[j+1][i]);
    }
 }
```

Above code segment which is not blocked code in this code segment data are traverse j followed by i in this code. But due to LRU policy data will not stay in cache so we are not getting the temporal locality. The access to the terms u[j][i-1], u[j][i+1] which are left and right neighbours follow spatial locality of reference due to row-major order but the term u[j-1][i] which is part of the previous row can get evicted from cache due to LRU policy due to the limitations of the size of cache. So we cannot achieve the temporal locality.

The introduction of loop blocking is illustrated in this section.
1D blocking: The original code is modified as shown below. This scan along 'i' loop is divided into chunks of size 'bi' (block size). The 'j' loop remains unaltered. The outermost loop 'ii' is added to handle the different block. This is 1D blocking. In this code segment we have spatial locality and temporal locality. Here 'i' loop is blocked. The 1D code is modified as shown below.

```
 //1D blocked code segment:
   for(ii=1;ii<imax;ii+=bi){
      for(j=1;j<jmax;j++){
         for(i=ii;i<ii+b;i++){

u[j][i]=0.25*(u[j][i-1]+u[j][i+1]+u[j-1][i]+u[j+1][i]);
          }
      }
   }
```

2D blocking: This scan along 'j', 'i' loops are divided into chunks of size 'bj' and 'bi' respectively. The 'j' and 'i' loops determine the block. The loops 'jj' and 'ii' are added in order to handle the scanning of different blocks. This is 2D blocking. In this code segment we have spatial locality and temporal locality. Here 'j' and 'i' loops are blocked.

```
//2D blocked code segment:
  for(jj=1;jj<jmax;jj+=bj){
     for(ii=1;ii<imax;ii+=bi){
        for(j=jj;j<jj+bj;j++){
           for(i=ii;i<ii+bi;i++){
       u[j][i]=0.25*(u[j][i-1]+u[j][i+1]+u[j-1][i]+u[j+1][i]);
           }
        }
     }
  }
```

V. Conclusion

The performance of Jacobi kernel depends on the effective reuse of data in cache. We can improve the performance of Jacobi kernel (PDE Solver) applying the loop blocking technique which efficiently leads to the reuse of the data. The performance analysis involves scaling studies for different problem sizes.

Acknowledgment

I would like to thank Dr. K. Meena of Vel tech University Chennai for her valuable suggestions and Mrs. Vaishali Maheskar, STO of C-DAC, ACTS, Pune for providing me infrastructure and motivation during the coursework. This work is done as part of M.Tech dissertation work.

REFERENCES

[1] Samuel Webb Williams Andrew Waterman David A. Patterson, "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures". Communication of ACM, Vol. 52, no 4, pp. 65-76, 2009.
[2] Aleksandar Ilic, Freder ico Pratas, and Leonel Sousa, "Cache-aware Roofline model: Upgrading the loft" IEEE *Computer Architecture Letters*, Vol. 1 3, NO. 1, JANUARY-JUNE 201 4.
[3] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G. Spampinato, Markus Puschel," Applying the Roofline Model" Proceedings of IEEE, ISPASS 2014.
[4] Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein, "Quantifying Performance Bottlenecks of Stencil Computations Using the Execution-Cache-Memory Model" Proceedings of the 29th ACM on International Conference on Supercomputing, ICS '15, Pages 207-216 .
[5] G. Hager, J. Treibig, and G. Wellein "Node-level performance engineering" tutorials at International Conference for High Performance Computing, Networking, Storage and Analysis, 2014 (SC'14).
[6] "Intel 64 and IA-32 Architectures Software Developers manual" 2012.
[7] Randal E. Bryant ,David R. O'Hallaron "Computer Systems: A Programmer's Perspective", 2nd ed., pp [559-643] .
[8] Michael T. Health "Scientific computing: An Introductory Survey", 2nd ed., pp [461-463]

Swapnil Kulkarni is a postgraduate student at Vel Tech Dr. RR& Dr. SR Technical University, Avadi, Chennai, Tamil Nadu, India in collaboration with CDAC-ACTS, Pune Maharashtra. He is pursuing his M.Tech in Computer Science Engineering with specialization in High Performance Computing Solutions.