PAPER • **OPEN ACCESS**

Performance evaluation of throughput computing workloads using multi-core processors and graphics processors

Performance evaluation of throughput computing workloads using multi-core processors and graphics processors

Gaurav P Dave, N Sureshkumar and S S Blessy Trencia Lincy
School of Computer Engineering, VIT University, Vellore-632014, India

E-mail: sureshkumar.n@vit.ac.in

Abstract. Current trend in processor manufacturing focuses on multi-core architectures rather than increasing the clock speed for performance improvement. Graphic processors have become as commodity hardware for providing fast co-processing in computer systems. Developments in IoT, social networking web applications, big data created huge demand for data processing activities and such kind of throughput intensive applications inherently contains data level parallelism which is more suited for SIMD architecture based GPU. This paper reviews the architectural aspects of multi/many core processors and graphics processors. Different case studies are taken to compare performance of throughput computing applications using shared memory programming in OpenMP and CUDA API based programming.

1. Introduction

Architectural advancements in microprocessors created improvement in performance of HPC applications by increasing processing elements on processor die. This in contrast to traditional way of increasing clock rate; created huge impact on the software development community. [1] With this approach we can divide the processors in two groups: the first is multi-core model where a few cores are integrated in single processor (e.g. Intel Core- i7 contains four physical cores) and second is many-core model where large number of core/ processing elements are integrated (e.g. Nvidia GeForce GTX 1080 contains 2560 single precision CUDA cores). The power of massively parallel computer systems can be utilized in two ways: automatic parallelism, parallel programming. Both ways differs in peak performance achievable for an application and effort required to implement parallelism in the code. In automatic parallelism ILP (Instruction Level Parallelism) and parallel compilers plays major role by taking advantage of hardware features of micro architecture.

The scope of the paper is to review architectural aspects of multi/many core processors and graphics processors. Different case studies are taken to compare performance of throughput computing applications using shared memory programming in OpenMP and CUDA API based programming.

The rest part of the paper is divided into following parts: Section: 2 shows throughput computing applications and usage in different domains. Section: 3 provide major performance evaluation metrics for parallel programs. Section: 4 provide review on HPC hardware and software. Section: 5 describes case study on CpenMP and CUDA programming and analysis using profiling. Section:6 concludes the paper. And Section: 7 shows proposed future work.

2. Throughput computing workloads

With increase in digital data from various sources like IoT, social networking sites, multimedia, big data, scientific data, medical data etc. has created processing and storage requirements at large scale. Such kind of massive data requires state of art storage, indexing, processing and retrieving requirements. One of attractive feature of such applications is "inherent data level parallelism" due to which data can be processed in independent order on different processing elements. "Data level parallelism" and "processing deadlines" plays major role in "throughput computing applications" as in simple terms throughput is how much data can be processed in given time period.

In [2], authors provide evaluation of selected throughput computing application kernels based on CPU and GPU implementations. These kernels are part of application code having inherent parallelism. They can be classified based on (1) computation and memory access (2) nature of memory access to take advantage of data parallelism in terms of SIMD (3) coarse grained and fine grained parallelism to identify synchronization requirements. Summary of these kernels are given in table:1. Authors using benchmarks and optimized code for CPU and GPU shows that the performance for CPU can be obtained at par with GPU implementations by carefully using the architectural features of multi-core CPUs. Author's shows performance comparison based on following criteria which is briefly given below:

1. Memory bandwidth:

The effect of data transfer to/from external memory in performance of kernel depends on two areas: (1) how much computation is provided in kernel so that it can use the memory transactions (2) whether the kernel has working set that is appropriate for size of storage elements (i.e. cache or buffers)

2. Computation in terms of FLOPS(Floating Point Operations Per Second):

It depends on performance of single thread and achieved TLP using multiple cores or DLP using vector (SIMD) units.

3. Impact of cache memory:

Working set characteristics determine the performance of kernel. Cache is useful for hiding memory latency of kernel by storing kernel's working set on cache fully or partially.

4. Level of data parallelism in terms of gather/scatter:

If the kernel is not bandwidth bound then it is advantageous with increasing DLP. However for achieving best performance the data layout should be aligned with the width of SIMD units.

5. Reduction of threads and synchronization among threads:

Throughput computing applications uses TLP and DLP for achieving best performance. Creation, manipulation, joining of threads or synchronization among threads involve overhead so reduction of joining and synchronization plays vital role.

6. Specific fixed functions

Certain kernels involve operations which are supported in hardware of CPU or GPU. Such kernels take advantage of such features.

Kernels are classified based on computation, memory and synchronization requirements. Various high performance libraries are provided in CUDA toolkit. Open source and proprietary libraries are also available for Nvidia CUDA enabled GPUs.

Table 1 Throughput computing kernels

Description of kernels	Characteristics	Parallelism (SIMD)*	Where TLP possible?	Usage	Nvidia CUDA library
SGEMM (Single precision	Performs $O(n^3)$ computations and $O(n^2)$ data accesses. Compute to	RL	Processing of 2D tiles created by row of	Used in linear algebra	cuBLAS, cuBLAS-XT, NVBLAS,

General matrix-multiplication)	data access ratio is O(n), so compute intensive nature.		matrix-1 and column of matrix-2		EM Photonics CULA Tools, ArrayFire
Monte Carlo (MC)	Heuristic method for simulating the nature of system like PI value approximations, finance stock price predictions. Simulation can be broadly divided into random number generator, data set generation, function evaluation and statistical aggregation. It is compute intensive.	RL	In random number generation and statistical aggregation	Simulating systems	cuRAND-For random number generation Thrust library can be used to accelerate MC on GPU.
Convolution (Conv)	It is commonly used for image filtering. It consist of multiply-add operation and data access for nearby neighbours. Each pixel is independently processed so both SIMD and TLP can be used. If multidimensional convolution is used then cache blocks play important role. It is compute intensive and for small filters bandwidth bound.	RL	Due to independent processing of each pixel, TLP can be used across pixels.	For analysis of images	NVIDIA Performance Primitives library (NPP)
Fast Fourier Transformation (FFT)	Used to transform signals from time domain to frequency domain and vice versa. Compared to DFT which requires $O(n^2)$ operations, FFT requires O(nlogn) operations.FFT is compute intensive or bandwidth bound depending on size of the signals.	RL	In smaller FFTs	Signal processing	cuFFT
Single-Precision A·X Plus Y(SAXPY)	It is group of scalar multiplication and vector addition in Basic Linear Algebra Subprograms (BLAS). For large vectors the operation is bandwidth bound.	RL	Across the vectors X and Y as both are independent.	Used in linear algebra	cuBLAS, cuBLAS-XT, NVBLAS, EM Photonics CULA Tools, ArrayFire
Lattice Boltzmann	Used for simulating fluid and runs efficiently on	RL	Across the cells of lattice	Computational	cuSOLVER, AmgX

method (LBM)	massively parallel computers. Compute and data are O(n) leading low compute to bandwidth ratio. It is bandwidth bound.			Fluid Dynamics(CFD)	
Constraint Solver (Solv)	Synchronization bound	G/S	Across constraints	Rigid body physics	-
Sparse matrix vector multiplication (SpMV)	Bandwidth bound for matrices with larger dimensions	G	Across non-zero elements	For solving sparse metrices	cuSPARSE
Gilbert–Johnson–Keerthi distance algorithm (GJK)	Used to find distance between two convex sets. Compute intensive.	G/S	Across objects	Collision Detection	Image processing libraries ArryFire
Sorting	Radix sort is taken into consideration. Compute intensive	G/S	Across elements	Database	ArryFire
Ray casting (RC)	Used to visualize 3-D datasets like medical images, CT scan data etc. Memory intensive operations are performed where first level working set may contain 4-8 MB and last level working set may contain over 500 MB to several GB	G	Across rays	Volume Rendering	NVIDIA IndeX
Searching	Compute intensive for small tree, bandwidth intensive at bottom of tree for large tree. In memory searching is faster if depth of tree is less than last cache size in CPU performance analysis. For GPU searching is compute intensive and run time search is proportional to depth of tree	G/S	Across queries	Database	-

Histogram (Hist)	Image processing algorithm for separating the pixels based on parameters and aggregating in different bins. Reduction or synchronization intensive	Conflict detection support is required for SIMD implemen tation	Across pixels	Image Analysis	Image processing libraries ArryFire, Nvidia Performance primitives
Bilateral (Bilat)	Used as non-linear filter in image processing for edge preserving smoothing provisions. Compute intensive	RL	Across pixels	Image Analysis	Image processing libraries like ArryFire

***Parallelism: RL-Regular, G-Gather, S-Scatter**

Kernels are classified based on computation, memory and synchronization requirements. Various high performance libraries are provided in CUDA toolkit. Open source and proprietary libraries are also available for Nvidia CUDA enabled GPUs.

In [3], authors divide the workloads based on "RMS- Recognition, Mining and Synthesis" and categorise throughput computing applications using similarity patterns based on algorithmic structure, mathematical model and data structures. Authors show the importance of workload convergence on architectural design of general purpose computing platforms and impact of computing devices on user's experience and developer's methodology and productivity.

"Random number generation(RNG)" is one of basic requirement for throughput computing applications like Monte Carlo methods. For accelerating RNG using GPU; different methods and requirements are reviewed in [4].

Brain data processing is one of the toughest challenge for identifying brain activities and functions. Massively parallel systems formed using "GPGPUs" can be used for gaining performance using three approaches: (1) decomposing the "electroencephalogram(EEG)" series (2) changing synchronization measures for multivariate EEG (3) reducing the dimensions for large scale "parallel factor analysis".[5]

3. Performance evaluation

For evaluating parallel applications one of major parameter is scalability. Given workload is scalable if by increasing number of processing elements the application can use the available pool of processors and improve the performance.

Speedup(S):

Speedup(S) of parallel solution by comparing with sequential solution of given application is ratio of running time of sequential solution (T_1) to running time of parallel solution on N processors (T_N) as shown in equation:1.

Parallel efficiency(E):

Parallel efficiency(E) can be obtained by dividing speedup(S) by total number of processing elements as shown in equation:2.

$$S(N) = T_1 / T_N \qquad (1)$$

$$E(N) = S(N) / N = T1 / (N_1 \ {*}T_N) \qquad (2)$$

Ideal speedup or linear speedup is N and that can be achieved using N processing elements if $T_N = T_1/N$ which leads to "parallel efficiency" 1.0. The linear speedup is theoretical and cannot be achieved in practical situation as some portion of any application is serially executable which cannot be parallelize.

Speedup metrics can be classified as:

1. Strong scaling or fixed size speedup
2. Weak scaling or fixed time speedup
3. Memory bounded speedup

Strong scaling speedup is calculated using Amdahl's law which states that "achievable speedup of given workload(w) is identified using portion of program's code which can be executed in serial manner only". If we take N as infinite number of computing resources then we get upper bound of speedup as 1/(Serial_fraction).

$$S(N) = T_1(w) / T_N(w)$$
$$= T_1(w) / (\text{Serial\_fraction} * T(w) + ((1 - \text{Serial\_fraction}) * T_1(w))/N)$$
$$= 1 / (\text{Serial\_fraction} + (1 - (\text{Serial\_fraction})/N) \qquad (3)$$

Weak scaling speedup is calculated using Gustafson's law which states in optimistic way that "as size of problem increases the serial portion of code is not always appropriate to assume. The parallel computing resources can be deployed to solve large problem in time bound manner. Workload can be divided into sequential (w_s) and parallel (w_p) parts. Here it is assumed that parallel part scales in linear way with increase of processors."

$$S(N) = T_1(w_s + N * w_p) / T_1(w_s + w_p)$$
$$S(N) = (\text{Serial\_fraction} * T_1(w) + N * (1 - \text{Serial\_fraction}) * T_1(w)) / T_1(w)$$
$$= N + (1 - N) * \text{Serial\_fraction}$$

Memory bounded speedup is higher than weak scaled speedup for applications where problem size increases more linearly with increase in number of processing elements because of constraints of memory i.e. arithmetic calculations increase faster than requirements of memory.[6]

4. HPC hardware and software

4.1 HPC hardware

Differences between multi-core and modern graphics processing units:

Multi-core processors

Each core (generally <10) works as an independent processing element having complete instruction set and out-of-order execution or dynamic scheduling (i.e. each core can execute different tasks and instructions are processed according to which operands are available at given instance of time rather than program order). Multi-core processors can be homogeneous (i.e. contains similar cores) or heterogeneous (i.e. contains different kind of cores). "Hyperthreading" may be supported (i.e. each core can execute two hardware threads at same time). Generally they are used to maximize speed of execution for sequential programs). Design is optimized for sequential code performance by using ILP (Instruction Level Parallelism). Large cache memory for data and instruction are provided to hide instruction and data latency.

Graphics processing units

GPUs contain hundreds of SIMD cores having limited instruction set and heavily multithread, in-order execution and single-instruction issue processors. A single SIMD instruction can execute multiple operations on different operands so to take advantage of SIMD instructions our code should be in form of vectors means independent instructions of similar type has to be identified and these selected instructions are replaced by SIMD instructions. Control and instruction cache are shared among other cores. Design is optimized for multimedia intensive applications. Maximum chip area and power used for floating point calculation so design focus is on providing large number of numeric computation engines. Large memory bandwidth is provided for moving data to GPU as well as small cache memory is provided to control bandwidth requirements for applications.

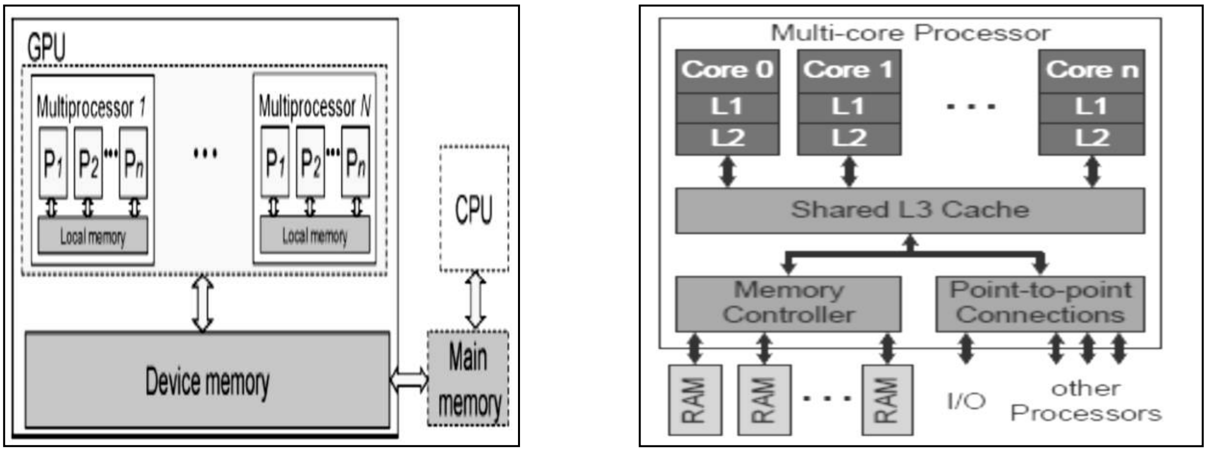


Figure 1 Simplified architecture of multi-core processor and GPU

4.2 Software
As surveyed in [7], parallel programming models can be classified as:

4.2.1. Pure Parallel Programming Models

4.2.1.1. POSIX threads
Thread is known as light weight process and it has program counter (PC) and stack memory. The "Pthreads, or Portable Operating System Interface (POSIX) Threads" is a collection of C types and procedures. "Pthread" is developed using a header file "pthread.h" and library which is useful for "fork/join" based parallel programming. This model provides dynamic memory allocation using heap memory and shared global variables. For giving exclusive access of shared variables to threads mutex and semaphore constructs are supported. Author suggests that this model is not suitable for HPC applications due to scalability issues(i.e. number of threads created in program are independent of number of available processors in the system).

4.2.1.2. OpenMP (Open Multiprocessing)
"OpenMP" is used for shared memory parallel programming. It is a "multithreading interface" to create HPC applications by providing abstract compiler directives in C, C++ or Fortran languages to create threads, provide synchronization and manage memory in shared memory environment. It follows block structured fork/join model for creating parallel regions in the programs. Each parallel regions is executed by the created threads as single individual task. This approach is also known as "work sharing". So parallel loop based structures, "Single Program Multiple Data (SPMD)" and fork/join structures are benefited using "OpenMP". Apart from higher level abstraction in terms of compiler directive, this model supports "application specific synchronization primitives" which ease the programmers burden for managing synchronization explicitly in programs; unlike in "Pthreads".

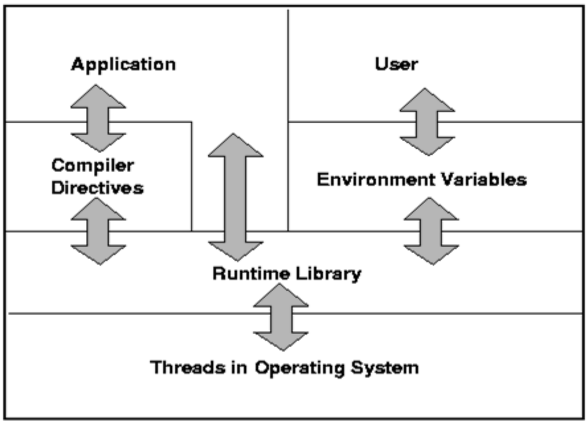


Figure 2 OpenMP architecture

4.2.1.3. Message passing

"Message passing" programming model is used to create distributed memory parallel programs. While comparing with "OpenMP"; communication among different processes is carried out by passing messages rather than sharing variables. "Message Passing Interface (MPI)" is de facto standard for creating large scale programs for HPC applications using distributed memory architecture. It supports "SPMD" and "master/worker" programming patterns. "MPI" is good for portable applications and "task parallel" programs which consist of dynamic data structures and unstructured calculations. Nowadays "MPI" supports combination of "message passing" and "multithreading" with inherent thread safety levels.

4.3. Heterogeneous parallel programming

Due to emergence of "general purpose graphics processing units (GPGPUs)" which provides traditional GPUs as accelerator for general purpose computations using different "APIs".

4.3.1. CUDA

"Compute Unified Device Architecture(CUDA)" API is created by NVIDIA for creating applipcations which can take advantage of massively parallel graphics processors to perform general purpose computations. "CUDA" model provides high level abstractions and compilers directives for C, C++ and Fortran languages. The terminology is briefly provided here. For this model, CPU known as host as other device (i.e. GPUs, signal processors, application specific SoCs etc.) are interfaced and managed by CPU. As seen in architecture of HPC hardware, "GPU" consist of collection of "streaming multiprocessors (SMs)" which executes a large number of threads in parallel manner. The threads are logically organized into two level hierarchy of grid and blocks. Each thread has a unique "ThreadID". The block may contain one of three kind of hierarchy i.e. one dimensional, two dimensional or three dimensional for data structures like vector, matrix and complex number or volume respectively. This model is well suited for "SPMD" pattern which can take advantage of SIMD based GPUs. Thread creation and management is done by CUDA implicitly whereas distribution of work among threads is provided by the programmer in terms of number of blocks per grid and number of threads per block. Block of threads creates a workgroup which is executed on SMs. "Global function" with "CUDA" primitives is created with appropriate thread hierarchy and synchronization. As shown in Fig. [3] global and shared memory are provided. Constant memory is "read only" for device(GPU) and shared memory is accessed by all threads in the block.

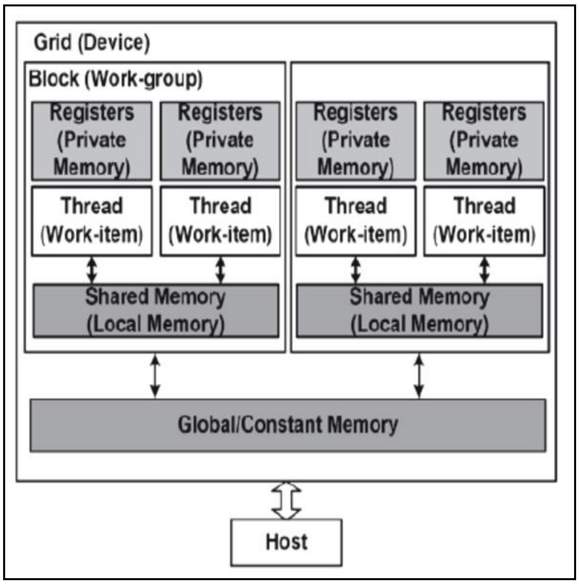

Grid (Device)
Block (Work-group)
Registers (Private Memory)
Registers (Private Memory)
Registers (Private Memory)
Registers (Private Memory)
Thread (Work-item)
Thread (Work-item)
Thread (Work-item)
Thread (Work-item)
Shared Memory (Local Memory)
Shared Memory (Local Memory)
Global/Constant Memory
Host

Figure 3 "CUDA" architecture and memory model

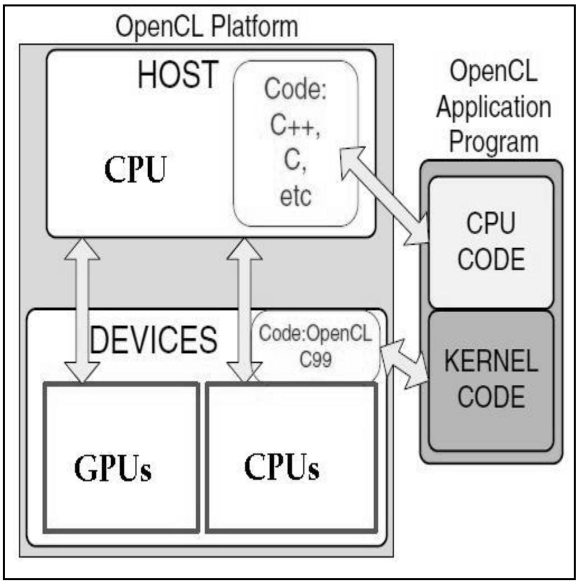


Figure 4. "OpenCL" simplified Model

4.3.2. OpenCL

As shown in Fig. 6, it is used for general purpose heterogeneous parallel computing. While comparing with "CUDA", it supports heterogeneous devices of different vendors and its open royalty free standard.

4.3.3. DirectCompute and TPL(Task Parallel Library) by Microsoft

Microsoft supports GPU programming using DirectCompute. "TPL" provides compiler directive and programming constructs to take advantage of multi-core processors in "MS .NET" environment.

4.3.4. AMD ATI Stream

It is used by AMD multi-core GPUs. It gives support for data and task based parallel programming.

4.3.5. Intel Array Building blocks(ArBB)

It provides general purpose "vector parallel programming" for mathematical and data intensive calculations. It is made of standard C++ library and implicitly uses Intel's Thread Building Blocks (TBB).

5. Implementation

We have used multi-core processor Intel(R) Core(TM) i7-2670QM CPU @ 2.20GHz. Maximum single core frequency is 3.10 GHz using Intel Turbo Boost Technology. It has 4 Cores or ALUs. Intel Hyper-Threading Technology is supported so two threads can be executed per core hence total 8 logical processors are available in the system. 8 GB of RAM is provided in the system. Nvidia GeForce GT 540M graphics card is installed.

5.1 Profiling

For understanding working of CUDA and OpenMP based application behaviour on the given hardware and software; profiling is done using of visual profiler of NVIDIA Nsight in Visual Studio 2015. Such kind of profiling using benchmarks provides better optimization criteria for throughput computing workloads as they tend to be application domain specific and contain similarity patterns or design patterns implicitly. With the available hardware and software we are showing basic steps to differentiate two kind of performance analysis tools: (1) profiling which generates statistical report based on executed instruction in specific time interval and (2) tracing which store all the events in program execution with reference to timestamp and sort the events by comparing initial time.

```
/*Sample code for using OpenMP and CUDA: cuda_openmp.cu (Source Reference: NVIDIA toolkit
documentation)*/
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <omp.h>
#include <stdio.h>
#include <helper_cuda.h>
#include <stdlib.h>
#include <cuda.h>
using namespace std;

// a simple kernel for multiplying
__global__ void kernelMultiplyConstant(int *g_a, const int b)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  g_a[idx] *= b+12345;
  printf("%d ,", g_a[idx]);
}

int main(int argc, char *argv[])
{
  int num_gpus;   // number of CUDA GPUs
  cudaGetDeviceCount(&num_gpus);//identify GPU in system
  if (num_gpus < 1)
  {
    printf("no CUDA capable devices were detected\n");
    return 1;
  }
  // display CPU and GPU configuration
  printf("number of host CPUs:\t%d\n", omp_get_num_procs());
  printf("number of CUDA devices:\t%d\n", num_gpus);
  for (int i = 0; i < num_gpus; i++)
  {
    cudaDeviceProp dprop;
    cudaGetDeviceProperties(&dprop, i);
    printf("   %d: %s\n", i, dprop.name);
  }
  int n = num_gpus * 8192 *100;
  int nbytes = n * sizeof(int);
  int *a = 0;
  int b = 99;
  a = (int *)malloc(nbytes);
  if (0 == a)
  {
    printf("couldn't allocate CPU memory\n");
    return 1;
  }
```

```
   for (int i = 0; i < n; i++)
      a[i] = i;
   omp_set_num_threads(num_gpus*8);// create 8 as many CPU threads as there are CUDA devices
   #pragma omp parallel
   {
      int cpu_thread_id = omp_get_thread_num();
      int num_cpu_threads = omp_get_num_threads();
      // set and check the CUDA device for this CPU thread
      int gpu_id = -1;
      cudaSetDevice(cpu_thread_id % 2);   // "% 2" allows more CPU threads than GPU devices
      cudaGetDevice(&gpu_id);
      printf("CPU thread %d (of %d) uses CUDA device %d\n", cpu_thread_id, num_cpu_threads,
gpu_id);
      int *d_a = 0;
      int *sub_a = a + cpu_thread_id * n / num_cpu_threads;
      int nbytes_per_kernel = nbytes / num_cpu_threads;
      dim3 gpu_threads(128);  // 128 threads per block are allocated
      dim3 gpu_blocks(n / (gpu_threads.x * num_cpu_threads));
      cudaMalloc((void **)&d_a, nbytes_per_kernel);
      cudaMemset(d_a, 0, nbytes_per_kernel);
      cudaMemcpy(d_a,sub_a,nbytes_per_kernel, cudaMemcpyHostToDevice);
      kernelMultiplyConstant<<<gpu_blocks, gpu_threads>>>(d_a, b);
            cudaDeviceSynchronize();
            cudaMemcpy(sub_a,d_a,nbytes_per_kernel, cudaMemcpyDeviceToHost);

            cudaFree(d_a);
    }
    if (a)
    free(a); // free CPU memory

    getchar();
    return 0;
}
```

5.2 Description

The functions which are executed on GPU are known as kernel function. This is explicitly mention to differentiate between kernel workloads and parallel function of CUDA program. kernelMultiplyConstant() is kernel function. Thread hierarchy is created in kernel. Every thread created in for executing on GPU has same code with unique thread-id. So portion of array is divided among the threads according to workload distribution from main() function. cudaMalloc() is used for allocating memory on GPU's RAM.cudaMemset() is used to initialize the memory block. Kernel function is called form main using <<< block, threads per block>>> directive of CUDA. For doing calculations using GPU array is copied from host(i.e. CPU's RAM) to device (i.e. GPU's RAM) using cudaMemcpy() function. After performing required operation on array elements in GPU, the array is copied back to CPU's RAM for further calculations.

omp_set_num_threads() is used to create CPU threads using OpenMP library. #pragma omp parallel directive spawns number of thread and replicate code among them.

Fig. 5, 6 briefly shows the summery of profiling and tracing of the program.

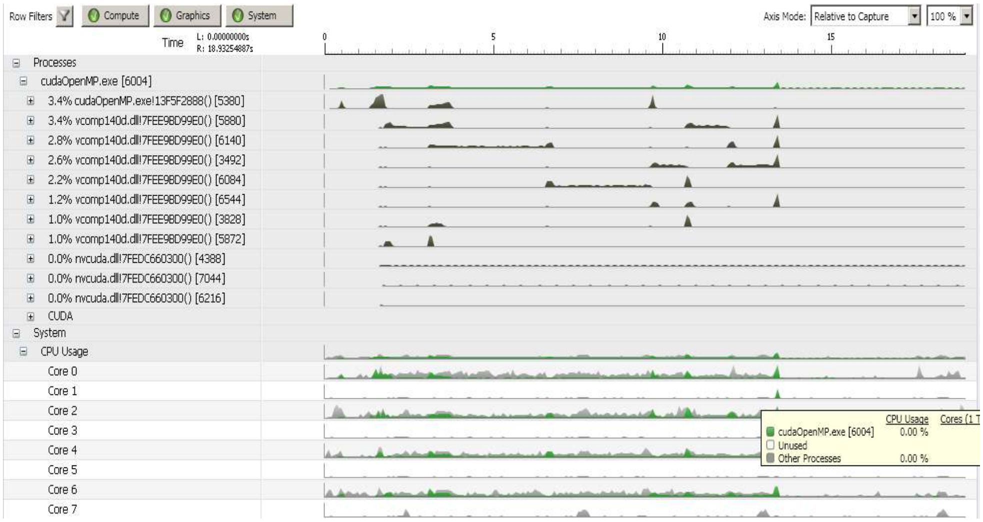
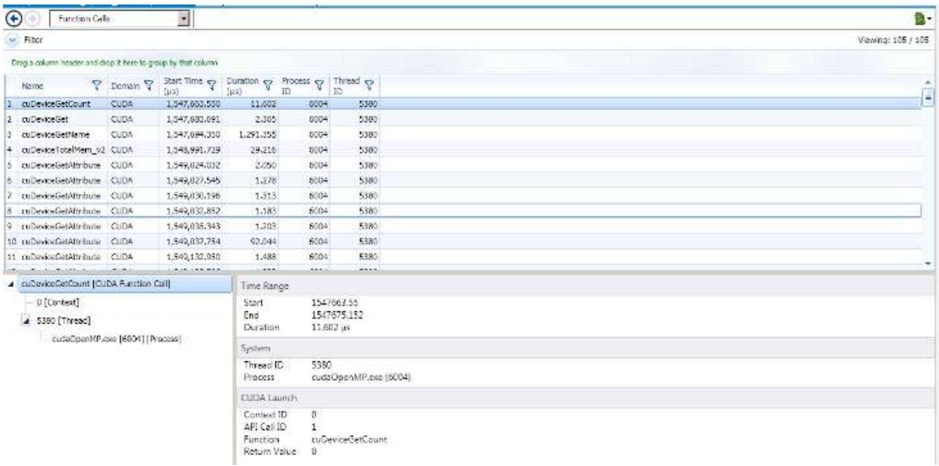

Row Filters
Compute
Graphics
System
Axis Mode: Relative to Capture
100 %
Time
L: 0.00000000s
R: 18.93254887s
0
5
10
15
Processes
cudaOpenMP.exe [6004]
3.4% cudaOpenMP.exe!13F5F2888() [5380]
3.4% vcomp140d.dll!7FEE9BD99E0() [5880]
2.8% vcomp140d.dll!7FEE9BD99E0() [6140]
2.6% vcomp140d.dll!7FEE9BD99E0() [3492]
2.2% vcomp140d.dll!7FEE9BD99E0() [6084]
1.2% vcomp140d.dll!7FEE9BD99E0() [6544]
1.0% vcomp140d.dll!7FEE9BD99E0() [3828]
1.0% vcomp140d.dll!7FEE9BD99E0() [5872]
0.0% nvcuda.dll!7FEDC660300() [4388]
0.0% nvcuda.dll!7FEDC660300() [7044]
0.0% nvcuda.dll!7FEDC660300() [6216]
CUDA
System
CPU Usage
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
CPU Usage
Cores (1 T
cudaOpenMP.exe [6004] 0.00 %
Unused
Other Processes 0.00 %

Figure 5 Timeline of program execution


Function Calls
Filter
Viewing: 105 / 105
Drag a column header and drop it here to group by that column

	Name	Domain	Start Time (μs)	Duration (μs)	Process ID	Thread ID
1	cuDeviceGetCount	CUDA	1,547,663.550	11.602	6004	5380
2	cuDeviceGet	CUDA	1,547,680.091	2.305	6004	5380
3	cuDeviceGetName	CUDA	1,547,694.350	1,291.355	6004	5380
4	cuDeviceTotalMem_v2	CUDA	1,548,991.729	29.216	6004	5380
5	cuDeviceGetAttribute	CUDA	1,549,024.032	2.050	6004	5380
6	cuDeviceGetAttribute	CUDA	1,549,027.545	1.278	6004	5380
7	cuDeviceGetAttribute	CUDA	1,549,030.196	1.313	6004	5380
8	cuDeviceGetAttribute	CUDA	1,549,032.852	1.183	6004	5380
9	cuDeviceGetAttribute	CUDA	1,549,035.343	1.203	6004	5380
10	cuDeviceGetAttribute	CUDA	1,549,037.754	92.044	6004	5380
11	cuDeviceGetAttribute	CUDA	1,549,132.950	1.488	6004	5380

cuDeviceGetCount [CUDA Function Call]
0 [Context]
5380 [Thread]
cudaOpenMP.exe [6004] [Process]

Time Range
Start 1547663.55
End 1547675.152
Duration 11.602 μs

System
Thread ID 5380
Process cudaOpenMP.exe [6004]

CUDA Launch
Context ID 0
API Call ID 1
Function cuDeviceGetCount
Return Value 0

Figure 6 Function calls with associated threads

Using profiling and tracing data we can find the detailed time analysis and space analysis of program for finding regions which requires optimization.

6. Conclusion

Multi-core processors and graphics processors provide promising way to improve performance of HPC applications and throughput computing workloads. Throughput computing applications are domain specific so optimization according to deployed hardware and software plays major role in improving performance. Scalability is an important factor as performance analysis metric for HPC applications. Parallel programming paradigm creates new challenges in term of debugging and profiling due to massively threaded parallel systems.

7. Future work

Implementation of different throughput computing workloads on HPC hardware and software for finding application specific optimization criteria for multi-core architecture based software development.

References

[1] Sutter H and Larus J 2005 Software and the concurrency revolution *Queue* **3** 54-62

[2] Lee V W, Kim C, Chhugani J, Deisher M, Kim D, Nguyen A D, Satish N and Smelyanskiy M 2010 Debunking the 100X GPU vs . CPU Myth : An Evaluation of Throughput Computing on CPU and GPU 451–460

[3] Chen B Y, Ieee S M, Chhugani J, Dubey P, Ieee F, Hughes C J, Ieee M, Kim D, Kumar S, Ieee M, Lee V W, Nguyen A D, Ieee M, Smelyanskiy M, and Ieee M 2008 Convergence of Recognition , Mining , and Synthesis Workloads and Its Implications **96**

[4] Ecuyer P L, Munger D, Oreshkin B, and Simard R 2017 ScienceDirect Random numbers for parallel computers : Requirements and methods, with emphasis on GPUs Mathematics and Computers in Simulation **135** 3–17

[5] Chen D, Hu Y, Cai C, Zeng K and Li X 2016 Brain big data processing with massively parallel computing technology : challenges and opportunities

[6] Sun X H and Ni L M 1993 Scalable Problems and Memory-Bounded In: *Journal of Parallel and Distributed Computing* **19** 27–37

[7] Diaz-montes J and Nin A 2012 A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era 19–21

[8] Molka D 2017 Performance Analysis of Complex Shared Memory Systems