Multimed Tools Appl (2011) 51:801–818
DOI 10.1007/s11042-010-0656-2

Parallel programming for multimedia applications

Hari Kalva · Aleksandar Colic · Adriana Garcia · Borko Furht

Published online: 4 December 2010

Abstract Computing capabilities are continuing to increase with the availability of multi core and many core processors. The wide availability of multi core processors has made parallel programming possible for end user applications running on desktops, workstations, and mobile devices. While parallel hardware has become common, software that exploits parallel capabilities is just beginning to take hold. Multimedia applications, with their data parallel nature and large computing requirements will benefit significantly from parallel programming. In this paper an overview of parallel programming is presented and languages and tools for parallel programming such as OpenMP and CUDA are introduced within the scope of multimedia applications.

Keywords Parallel programming · OpenMP · CUDA · SIMD · Multimedia programming

1 Introduction

Multimedia applications have been driving the need for increased computing capability on devices such as personal computers, mobile phones, and consumer electronics. Multimedia applications are characterized by large amounts of data and require large amount of processing, storage, and communication resources. The continuing decrease in the cost of multimedia data acquisition has made the data easily available and the decreasing bandwidth costs have made it easier for users to share videos and other data on the Internet. This increase in the amount of data has put heavier burden on the computing infrastructure necessary to process multimedia data. For example, users are allowed to upload HD resolution video to YouTube and YouTube processes the video to allow other users to access it at various bitrates and resolutions. Some of the computing requirements can be addressed using dedicated hardware; e.g, for video encoding and decoding. However, the use of dedicated hardware for video processing on general purpose devices is

H. Kalva (✉) · A. Colic · A. Garcia · B. Furht
Department of Computer & Electrical engineering and Computer Science, Florida Atlantic University, Boca Raton, FL 33431, USA
e-mail: hari@cse.fau.edu

not a cost effective solution. Innovations in algorithms, software, and tools have made possible the use of general purpose processors for high performance multimedia applications. Of all the methods to develop high performance multimedia applications, parallel programming stands out because of the performance gains it promises. As the name implies, parallel programming involves executing portions of a problem in parallel in order to reduce the overall execution time. Multimedia applications exhibit parallelism and can benefit tremendously from such an implementation.

While computing capabilities have increased greatly, especially with the recent availability of quad-core CPUs and many core GPUs that support parallel programming, effective use of these capabilities seems to have trailed. The use of parallel execution capabilities on these processors requires the utilization of proper programming and software development techniques. This leads to one of the challenges in bridging the gap between parallel processing capabilities of the hardware and the end user applications: the lack of software that exploits the hardware as most programmers are trained in and the use of sequential programming [1]. With many-core processors with hundreds of cores on the horizon [2], there is an urgent need for jump starting the use of parallel programming in high performance applications. Another trend that has emerged over the last 4 years is the use of graphics processors (GPU) for general purpose programming [20]. GPUs present another alternative for parallel implementation of multimedia applications and are especially suitable for highly data parallel problems. Multimedia applications, with a need for large amount of computing resources, and data that lends itself to parallel processing, are the first to benefit form these new developments in parallel programming.

Figure 1 shows the parallel programming hierarchy—parallel programming and processing possible at different levels in a computing environment. At the lowest level is the instruction level parallelism supported by Single Instruction Multiple Data (SIMD) instruction sets available on most processors today. These instructions can be used to speedup data parallel computations. Each core of multi-core processors supports SIMD instructions and optimizations at this level require thorough knowledge of instruction sets and assembly programming. SIMD instruction set programming is usually used to accelerate code segments or sub components of larger programs.

Multi-core processors and shared memory multiprocessor systems can speed up applications when using multiple threads and/or multiple processes. At this level, parallel programs can be written using multi threaded programming using explicit threading supported by the operating system or using programming frameworks such as OpenMP. Multithreaded programming at this level is independent of SIMD programming and both can be used to speedup program execution. At the highest level of hierarchy is cloud or cluster computing where multiple systems connected over networks are used to solve large scale problems or process large amounts of data. In distributed programming, data is spread among the available systems in a cluster/cloud and processes executed on each system computes the data assigned to it. The processes executing on each node can be optimized using multithreaded and SIMD instruction set programming. Effective implementation can thus exploit parallel processing capabilities at all levels of execution there by maximizing the speedup.

In this paper each of the four approaches described above are discussed emphasizing their usage in the multimedia area. Specifically, Section 2 discusses the multimedia capabilities on General Purpose Processors and current multimedia instruction sets. Section 3 overviews parallelism in multimedia applications highlighting the parallelism in the area of video encoding in particular; finally Section 4 discusses programming languages and tools for parallel programming, specifically OpenMP and CUDA and briefly overviews

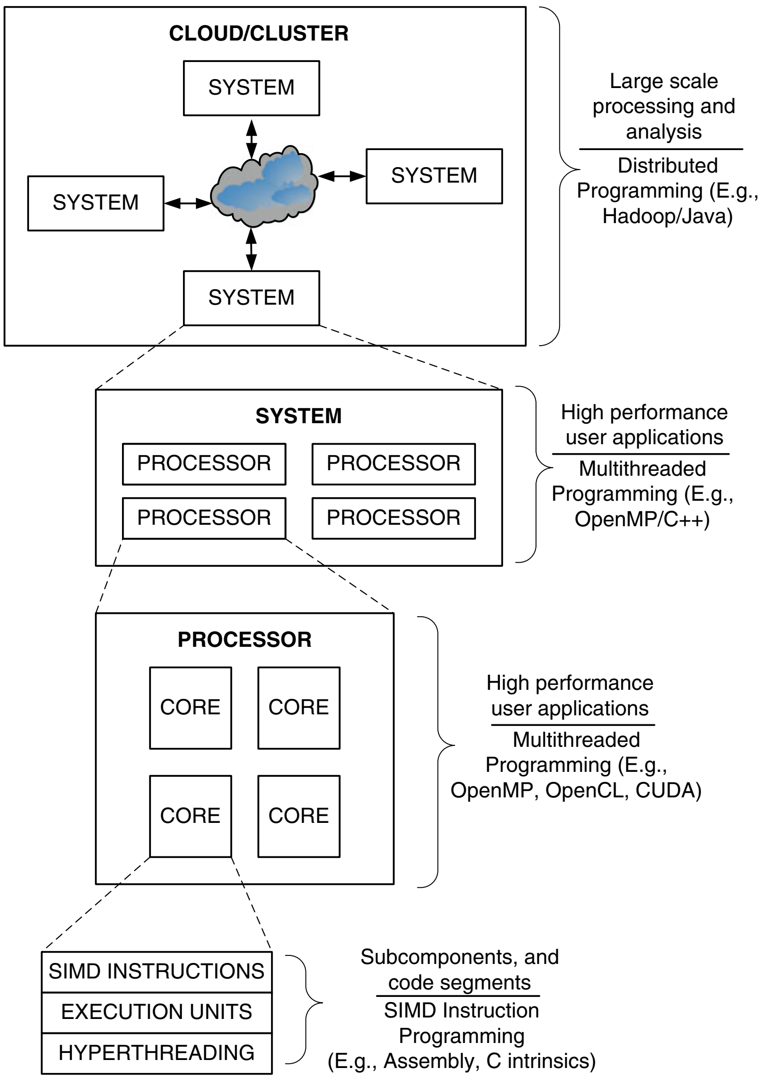

CLOUD/CLUSTER
SYSTEM
SYSTEM
SYSTEM
SYSTEM
Large scale processing and analysis
Distributed Programming (E.g., Hadoop/Java)
SYSTEM
PROCESSOR
PROCESSOR
PROCESSOR
PROCESSOR
High performance user applications
Multithreaded Programming (E.g., OpenMP/C++)
PROCESSOR
CORE
CORE
CORE
CORE
High performance user applications
Multithreaded Programming (E.g., OpenMP, OpenCL, CUDA)
SIMD INSTRUCTIONS
EXECUTION UNITS
HYPERTHREADING
Subcomponents, and code segments
SIMD Instruction Programming (E.g., Assembly, C intrinsics)

Fig. 1 Hierarchy in Parallel Programming

the subject of cloud computing for multimedia applications with a particular emphasis on the Hadoop platform, whose free and open nature make it worth exploring.

2 Multimedia capabilities on general purpose processors

As computing needs driven by multimedia applications grew, processors with increasingly more capabilities were developed. In the 90s, the emphasis was on packing more transistors, increasing the operating frequency, and adding dedicated instructions to increase the processing capabilities. With the realization that the operating frequencies cannot continue to increase, the industry focus has shifted to the use of multiple cores on the same processors. This trend is expected to grow and processors with hundreds of cores are on the horizon. However, software parallelization is a complex task and converting all the existing software to exploit parallel hardware is unlikely to happen. Multimedia applications are the kind of applications that can benefit from parallel implementations and given their significance and wide use, are expected to drive the demand for multicore processors and parallel implementations.

2.1 Multimedia instruction sets

The multimedia extensions to general purpose processors were primarily driven by the advent of digital video, 3D graphics, and gaming applications. The need for increased performance on PCs and the data parallel nature of the target applications resulted in processors with Single Instruction Multiple Data (SIMD) instruction sets. The MAX extensions to the HP PA-RISC architecture were one the first SIMD extensions to general-purpose processors [17]. The Visual Instruction Set supported on the Sun UltraSparc processors provides the SIMD extensions to support multimedia and data parallel applications [25]. The MMX instruction set for the Intel Pentium processors [23], the 3DNow! extensions for the AMD K6 processors, and the AltiVec extensions for the Motorola/IBM PowerPC processor [11] all provided SIMD instructions to speedup data parallel applications. The SIMD instructions have become common and are used even in embedded processors as a way of supporting parallelism at instruction level.

The SIMD instruction set typically consists of parallel arithmetic, logical, comparison, and data movement instructions. Some instruction sets provide type conversion and application specific instructions. A comparison of the SIMD support on general purpose processors can be found in [12]. Development of high-performance application for such processors has traditionally been done using the assembly language programming or a programming library provided by the processor vendor. Both of these approaches render the developed code incompatible for porting to other processors. The lack of software support is the primary reason for problems in developing applications that harness the full potential of SIMD instruction sets. The difficulties in developing applications that exploit SIMD instruction is well discussed in [9].

There have been few efforts in developing compilers that exploit SIMD parallelism [16]. Compilers that generate optimized code that takes advantage of the SIMD capabilities are becoming available but are limited by their performance. For example, Intel C++ compiler performs some optimization by parallelizing identifiable loops but cannot generate efficient code in all cases. This also is limited to generating code for Intel processors. Developing a compiler that can understand the parallelism inherent in an application is a very difficult task. In addition to this, a generic compiler cannot produce optimal code in all situations. An ANSI C extension to support multimedia extensions on general purpose processors is presented in [3]. Leupers' work on code generation for multimedia processors addresses some of the issues in optimized code generation for SIMD capable processors [19]. While these techniques may be applied to develop compilers that parallelize code such as the Intel C++ compiler, they have the disadvantage of being a generic solution for all applications. While the past research addresses the issue of exploiting SIMD parallelism to certain extent, it fails to provide ways to shield application developers from having to master multiple processor instruction sets in order to maximize performance.

In fact, application developers have traditionally relied on hand-coded assembly level programming in order to achieve high performance. This is the case due to the difficulty of creating a generic compiler that can optimize the performance for all applications and architectures. This creates a scenario where applications are developed for a specific target processor, however, such applications cannot be ported to different processor platforms in a way that maximizes the performance on that platform as well. To achieve comparable performance, the applications have to be re-written using the assembly language of the target processors. One alternative to this lower level code development is the use of libraries that implement optimized primitives for particular problem domains (e.g,. optimized libraries for image processing, cryptography).

Development of optimized code using a SIMD instructions set is time consuming and higher level programming alternatives may be appropriate. Since SIMD instruction sets are orthogonal to multi-core capabilities, one may achieve the desired performance goals by using parallel programming on multi-core processors and leave instruction level parallelism to the compiler.

3 Parallelism in multimedia applications

Two processes can be executed in parallel when results of one process do not depend on and do not affect the second process. Parallelism can be broadly classified into data parallelism and task parallelism. Multimedia applications exhibit both types of parallelism. In data parallel problems, data can be processed in parallel by multiple processors. For example, encoding individual frames of video sequence as Intra frames or JPEG frames can be performed in parallel as there are no dependencies between processes encoding two separate frames. Figure 2 shows two ways of partitioning data among the available processors. The data partitioning decisions depend on the application needs. If the application needs each frame to be processed as fast as possible, the partitioning shown in 2.b would be appropriate. On the other hand, if the application needs to maximize the total number of frames processed in a given time, the partitioning shown in 2.a is more appropriate.

When the frames of a sequence are coded using predictive coding, they cannot be encoded in parallel because the current frame depends on the previous frame. In such cases, an encoding task can be broken down into sub-tasks that are performed in a sequence and this sequence of tasks can use pipelining to execute in parallel. Alternatively, data parallel sub-tasks can be identified and executed in parallel. When performing motion estimation non-overlapping blocks in a video frame can be processed in parallel. In task parallel problems, different tasks can be executed in parallel. For example, in video encoding, while one processor is encoding frame n, a second processor can perform pre-processing on frame n+1.

Consider an example where each frame of video has to go through a set of tasks A, B, C, and D respectively. The problem is task parallel if the tasks A, B, C, and D can be executed in parallel. If there are dependencies between the tasks, they cannot be executed in parallel and alternative strategies such as pipelining can be used (Fig. 3). Tasks can be pipelined

Fig. 2 Example of a data parallel problem

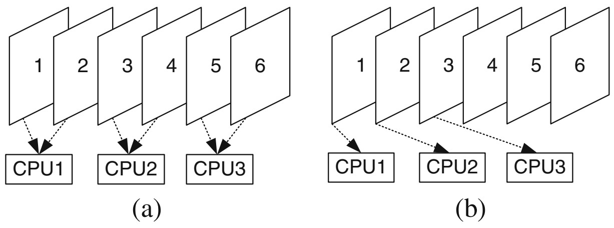
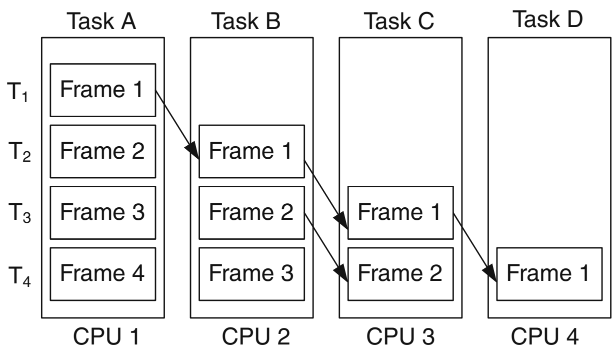


Fig. 3 Use of pipelining to parallelize dependent tasks



such that CPU1 performs task A on a given frame and the output of task A is used by CPU2 to perform task B. Even though there are dependencies that prevent parallel execution (task B cannot be executed before task A), pipelining can be used to execute A and B in parallel but on different data sets. With pipelining, CPU1 runs Task-A on Frame 1 and CPU2 waits until CPU1 is finished with Task-A on Frame 1. At this time CPU 1is free to run Task A on Frame 2 and CPU 2 can run Task B on Frame 1 in parallel. This continues until all the tasks and/or CPUs are executing in parallel.

3.1 Parallelism in video encoding

Video encoding is a computationally intensive process and exhibits data parallelism. The data parallelism varies with the type of the encoders used. All practical video encoders today are based on hybrid video coding techniques—compressing video with a hybrid of motion compensation and transform coding. Video encoders reduce the redundancies among the frames of a video sequence using predictive coding techniques. Since predictive coding techniques use previously coded data to form predictions, this introduces dependencies that reduce data parallelism.; frames of a video cannot be encoded in parallel because of inter-frame dependencies. However, independent operations such as preprocessing on video frames could be performed in parallel. A video frame is typically encoded as macro blocks (16×16 pixel blocks) and coding of pixels in a block could depend on coding of pixels of previously coded blocks. Figure 4 shows a typical encoder and amount of dependencies and parallelism in the process. Some video encoders offer opportunities to parallelize the encoding process by allowing the encoding process to encode a set of blocks, referred to as a slice, independent of other slices.

The performance of parallel processing is measured using a metric known as speedup; speedup is defined as the ratio of time taken to execute a parallel program and

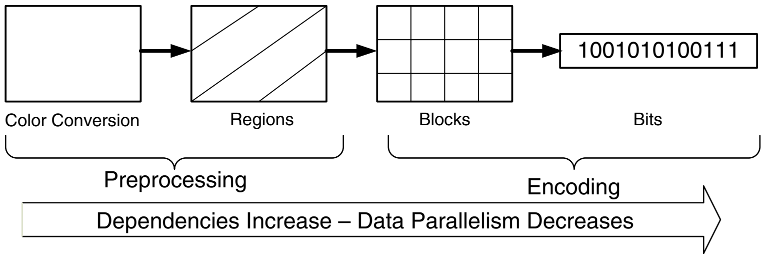


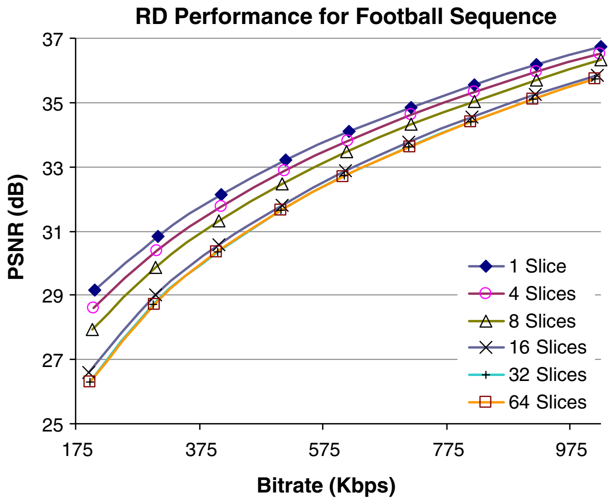
Fig. 4 Parallelism in video encoding

the time taken by a best sequential implementation to accomplish the same task. Any sequential processing required in an application reduces the overall speedup. The amount of parallelism and hence the speedup that can be achieved depends on the particular problem.

To maximize the speedup for a parallel implementation, the sequential portions of the code must be reduced. The extent to which problems can be parallelized depend on the nature of the problem. The MPEG AVC/H.264 video encoder is an interesting problem to study the trade offs in parallel encoding. The MPEG AVC/H.264 video encoders compress a video frame as one or more slices [18]. A video frame is comprised of a fixed number of macro blocks (16×16 non-overlapping blocks) and a slice is a set of macro blocks. A slice can be encoded independent of the other slices in the frame. Macro blocks use data from other coded macro blocks in the slice to improve compression efficiency (i.e,. reduces bitrate for the same quality). As the number of slices increases, parallelism increases because the slices can be encoded independent of each other. However, the compression efficiency suffers because of increased slice overhead and also because the macro blocks of a slice have fewer neighboring MBs to depend upon. This presents an interesting problem of trading off compression performance for parallelism and encoding speed that is illustrated here via experimentation.

Using the x264 video encoder Experiments were conducted to study such tradeoff. Video sequences at 352×288 resolution and 30 frames per second were encoded with 1 to 64 slices and at bitrates from 200 to 1000 Kbps. The quality of video was measured using peak signal to noise ratio (PSNR). Figure 5 shows a plot of PSNR, a measure of video quality, and the bitrate of the compressed video with the video encoded using different number of slices per frame. As the number of slices increase, dependencies decrease, and compression performance (quality for a given bitrate) decreases. On the other hand, parallelism increases because slices of a frame can be encoded in parallel. On a four core machine used for these experiments the increased parallelism resulted in a speed up of 2.65 for video coded with four slices. Servers with eight processing cores are fairly common and encoding the video with eight slices will further improve the speedup. However, with eight slices, the quality of the video drops by about 1 dB compared with single slice encoding. The video quality achieved with 8 slices encoded at 500 Kbps can be achieved by a single

Fig. 5 RD performance curve



slice video encoder at about 400 Kbps. The cost of the increasing the parallelism is increase in bitrate of 100 Kbps or a decrease in quality of 1 dB.

4 Languages and tools for parallel programming

The languages and tools used to write parallel programs depend on the underlying the hardware and the programming model supported by the hardware. The available tools are still limited and continue to evolve into simpler tools and languages. The performance gains depend on the problem, the amount of effort in tuning the software, and the underlying hardware. Programmers have to consider their application requirements and performance targets when selecting a parallel programming framework. A sophisticated vectorizing compiler may be able to generate code with parallel instructions and provide the desired performance without explicitly parallelizing the coder. As with any software optimization efforts, developers should check for correctness and performance after every parallelization step.

The following sections present most commonly used tools for parallel programming, focused on programming applicable to common class of multimedia applications—applications used by end users and run on desktops or individual workstations. We also discuss the use of Hadoop for large scale problems using distributed programming.

4.1 OpenMP

OpenMP is an API to write parallel programs in C/C++ and Fortran for shared memory processors [5]. OpenMP began as an industry driven effort to provide a standardized parallel programming framework for shared memory multi-processor systems. Shared memory processors have a common memory that is shared by all the computing cores of the processor. All the multi-core processors that are being used today fall in this category and are suitable for developing parallel programs using OpenMP. In addition, most of the current C/C++ compilers support OpenMP

OpenMP programmers use extensions to C/C++ and Fortran—compiler directives, a runtime library, and environment variables in developing parallel applications. The compact nature of the OpenMP framework and the ability to parallelize the code incrementally makes parallel programming with OpenMP easier. Programmers describe parallelism at a higher level and the OpenMP compiler handles the creation and management of threads necessary for parallel execution on the underlying hardware. OpenMP offers incremental parallelism; i.e, only portions of a sequential program can be parallelized allowing for quick performance gains on multi-core processors.

OpenMP is based on a fork-join model of parallelization where a master thread forks multiple worker threads to execute in parallel. The worker threads join and synchronize at the end of the parallel section and the sequential execution continues.

Consider the problem of motion estimation in video compression. A prediction is computed for every macro block (MB)—a 16×16 block of pixels—in a frame and the difference between the current and the prediction is then compressed. The prediction for a given block is determined by searching a reference frame—a previously encoded frame, and finding a block that is closest to the current block in a given search range. Motion estimation represents the most complex stage of video compression and can account for as much as 80% of the encoding time. Figure 6 shows a full search motion estimation algorithm implemented in C for sequential execution. The motion estimation is performed

```
char **pCur; // current frame buffer
char **pRef; // reference frame buffer
int nWidth; // width of the frame
int nHeight; // height of the frame
int nRange; // motion search range

// iterate over all MBs in a frame
for(int y = 0; y<nHeight; y+=16){
   for(int x = 0; x<nWidth; x+=16){
      BlockMotionEstimation(pCur, pRef, x, y, nWidth, nHeight,
nRange);
   }
}

// Full Search Motion Estimation
BlockMotionEstimation (unsigned char *pCur, unsigned char *pRef,
int x, int y, int nWidth, int nHeight, int nRange){
      unsigned int nMinSad = 0xFFFFFFFF, nSad;
      int nMinMVX, nMinMVY;

      int nBlockNumber = (nWidth/16)*y/16 + x/16;
      for(int rx = x – nRange; rx < x + nRange; rx++){
         for(int ry = y – nRange; ry < y + nRange; ry++){
            for(int i = 0; i<16; y++){
               for(int j = 0; j<16; x++){
                  nSad += pCur[x+i][y+j] – pRef[rx+i][ry+j];
               }
            }
            if(nSad < nMinSad){
               nMinSad = nSad;
               nMinMVX = i;
               nMinMVY = j;
            }
         }
      }
      // motion vector with the best match (smallest cost)
      g_nMVx[nBlockNumber] = nMinMVX - x;
      g_nMVy[nBlockNumber] = nMinMVY - y;
}
```

Fig. 6 Full Search motion estimation computed sequentially

for all MBs in a frame. For each MB, a BlockMotionEstimation() function is called to perform motion estimation for the block with in a given search range.

Parallelizing this motion estimation loop using OpenMP is very simple and requires only one line of code—an OpenMP directive declaring a parallel region (Fig. 7). The "omp parallel for" directive instructs the compiler to distribute the work done in the for-loop immediately following the directive among all the processors (cores) on the system. In this example, if the height of the video is 320 pixels, the outermost for-loop will have 20 iterations. On a quad-core processor, each core will execute 5 successive iterations of y in parallel. The OpenMP compiler generates the code necessary to create the threads and distribute the workload among the threads. To guarantee correctness, programmers must ensure that the successive iterations are independent and can be executed in parallel.

```
// split the motion estimation task among all
// the available cores/processors
#pragma omp parallel for
for(int y = 0; y<nHeight; y+=16){
   for(int x = 0; x<nWidth; x+=16){
      BlockMotionEstimation(pCur, pRef, x, y, nWidth, nHeight,
nRange);
   }
}
```

Fig. 7 Full Search motion estimation computed in parallel using OpenMP

OpenMP provides multiple ways to set the number of processors used for parallel execution: fixed, dynamic, conditional, and run-time decision. By default, OpenMP uses all the available processors thus making the software independent of the hardware capabilities. The performance of the software will scale when new hardware is released without having to rewrite the software. As with any parallel programming framework, programmers have to take caution to prevent race conditions. All code before and after this code segment is executed sequentially.

4.2 SIMD instructionset programming

Most modern processors support SIMD instructions that execute the same operations on an array of data. These SIMD instructions use large registers to load multiple data elements to operate on. SIMD instructions support basic operations such as arithmetic, logical, and data movement operations. For example, a parallel *add* operation can be used to perform 16 additions in parallel using a pair 128 bit registers loaded with 8-bit integers. Since the parallelism is supported at instruction level, programmers need a complete understanding of instruction sets and have to write code using assembly programming techniques.

The popularity and usefulness of SIMD instruction sets in multimedia programming have resulted in extensions with instructions specifically targeting multimedia applications. For example, video compression algorithms compute the sum of absolute difference of pixels often and a dedicated instruction, PSADBW, was introduced in Intel processors. New compilers support SIMD parallelism by generating parallel instructions for clearly defined loops and parallel structures. However, hand coding is necessary to optimize reasonably complex code segments. Compilers make SIMD programming simpler by providing a C-like API for the SIMD instruction set. These C-functions, referred to as intrinsics, are defined for each SIMD instruction and guaranty that that particular instruction will be used in the assembly. Programming with intrinsics relieves the programmer of register management but the programmers still need instruction level knowledge and expertise.

SIMD instructions can be used to speedup code segments that exhibit data parallelism and are independent of multi-threaded and multi-core programming. Figure 8 shows the use of SIMD intrinsics to accelerate the computation of motion estimation shown in Fig. 6. The SIMD implementation loads 16 pixels from the source and destination and computes the sum of absolute differences of 16 pixels in parallel.

4.3 GPU programming with CUDA

In the quest for maximum speed, GPUs (Graphics Processing Units) have evolved far beyond single processors. Modern GPUs are not single processors but rather are parallel

```
#ifdef USE_SIMD_OPT
      _asm{
            //setting starting points for current and reference block
            mov ebx,pCurMBData;
            mov ecx,pRefMBData;
            mov eax,blockArea;
      }
            _mm_xor_si128(xmm2,xmm2);
L1:         _mm_store_si128 ([ebx],xmm0);//loading current 16 values
from current block
            _mm_store_si128 ([ecx],xmm1);//loading current 16 values
from reference block
            _mm_sad_epu8(xmm0,xmm1);// doing sum of absolute
differenies for current 16 values of current and reference block
            _mm_add_epi64 (xmm2,xmm0);//adding results to the
accumulator
            _asm{
            add ebx,0x10;//increment adresses to pick next 16 values
            add ecx,0x10;
            sub eax,0x10;//decrementing counter
            jnz L1;            //checking if all values from the block
have been processed
            }
            _mm_unpackhi_epi64 (xmm3,xmm2);//finalizing of calculation
            _mm_unpackhi_epi64 (xmm2,xmm4);
            _mm_srli_si128 (xmm3,8);
            _mm_add_epi64 (xmm2,xmm3);//sadValue for current block is
stored in xmm2
            _mm_xor_si128(xmm3,xmm3);//cleanup
            _mm_xor_si128(xmm1,xmm1);//cleanup

            sadValue=_mm_cvtsi128_si32(xmm2);

      }
#else
                  for(int i = 0; i<16; y++){
                     for(int j = 0; j<16; x++){
                        nSad += pCur[x+i][y+j] - pRef[rx+i][ry+j];
                     }
                  }
#endif
```

Fig. 8 Optimizing motion estimation using SIMD programming

supercomputers on a chip that consist of many fast processors. The large amount of compute resources on GPUs make them a good candidate for high performance computing. NVIDIA has put a lot of emphasis on making general purpose programming on GPUs simpler by abstracting the GPU with threaded programming model. NVIDA's toolkit for GPU programming, called CUDA—Computer Unified Device Architecture—allows applications developers to write code that can be run on NVIDIA's massively parallel GPUs [15]. This allows applications developers to plug in a 500 gigaflop, 256-processor, NVIDIA-based card and upload applications to run within the NVIDIA GPU at far greater speed than possible on even the fastest general purpose CPU on the motherboard.

GPU is specialized for compute-intensive, highly parallel computation because of the sole purpose of the GPU in the first place—graphics rendering. Because of that GPU and CPU differs in a way that more GPU transistors are devoted to data processing rather than data cashing and flow control. This makes them excellent candidates to be used to solve highly arithmetically intensive problems. Taking into consideration that CUDA enabled GPU's are multiple core systems, the conclusion arises that the CUDA architecture is best suited for solving highly arithmetic problems that can be performed on many data elements in parallel—data parallelism. The processing capabilities on a NVIDIA GPU are organized into multiprocessors. Each GPU has several multiprocessors and each multiprocessor has 8 cores (32 cores on the NVIDIA Fermi architecture GPUs). GPUs have large global memory that is common to all the cores, shared memory that is common to all the cores of a multiprocessor, and registers that are exclusive to a core (16,000 registers in a multiprocessor). The effective use of shared memory is essential to maximize the performance.

CUDA allows developers to use C as a high-level programming language. Using already well established high level language and subtly adding additional, CUDA specific features (implemented in a way that they follow C programmer's way of thinking as close as possible) provides an extremely user friendly environment. When a CUDA programmer starts developing CUDA software he needs to understand what is different from regular C code and what is similar. The biggest difference is that CUDA code is divided into two parts, host code and device code. The host code which runs on the CPU, controls the whole procedure, including when to run the device code—the code that runs on the GPU.

A CUDA function referred to as a kernel represents the portion of code that will be executed on the GPU device. The portion of the program running on the CPU initializes the GPUs and controls the execution of kernels on the GPU. The key idea with CUDA programming is to have the kernels execute in parallel on the many cores available on the GPU with each kernel execution processing an independent block of data. For example, if there is a need to process K data elements, K kernel executions have to happen to process all of them. If GPU has N cores that means that N out of K kernels can be executed in parallel. So kernels become independent threads assigned to some core to be executed. The threads are organized as a set of thread blocks with each thread block containing a fixed number of threads. Thread blocks are assigned to multiprocessors for execution. The thread block index and the thread index identify a unique thread and these indices are typically used to determine the data to be processed by this thread. The cost of creating and executing threads on GPUs is minimal and applications that use millions of threads are common. For example, CUDA threads can operate on individual pixels in parallel thereby speeding up the processing time.

The motion estimation problem discussed in the previous section can be parallelized effectively using CUDA. The first step in executing a CUDA program is moving data from the main memory to the GPU. Memory movement between a CPU and GPU is expensive and should be minimized. For the motion estimation problem, the current and reference frames have to be moved to the global memory on the GPU. Figure 9 shows the example of a kernel setup to compute motion estimation in parallel. A thread block with 32 threads each is defined and an array of thread blocks is defined to process all macro blocks in a frame. A simple kernel to execute the motion estimation in parallel is shown in Fig. 10. The kernel is designed such that each thread processes one macro block. A thread block has 32 threads and thus computes motion vectors for 32 macro blocks. Each thread uses the unique thread and block index to determine the macro block being processed. Thread blocks are scheduled for execution on multiprocessors on the GPU. The example shows that parallelizing code in CUDA is relatively simple. The implementation shown is a simple example and efficient implementations use shared memory and other optimizations to maximize performance. At the end of kernel execution, results are moved back to the CPU for further processing.

```
      // Full Search Motion Estimation HOST code


      // allocating frame memory on GPU
      cudaMalloc( (void **)&cFrame_d, (width*height);
      cudaMalloc( (void **)&rFrame_d, (width+2*range)*(height+2*range));

      // copy frame from CPU to GPU memory
      cudaMemcpy(cFrame_d,cFrame,(width*height, cudaMemcpyHostToDevice);
      cudaMemcpy(rFrame_d,rFrame,(width+2*range)*(height+2*range),
cudaMemcpyHostToDevice);

      //setting up kernel variables
      //define a thread block with 32 threads
      dim3 dimBlock(32, 1);

      //set grid size to have enough thread blocks to process all macroblocks
      int SizeGrid = (width/16 * height/16)/32 + ((width/16 * height/16)%32)
!= 0 ? 1:0);
      dim3 dimGrid(SizeGrid ,1);

      //calling CUDA kernel
      //finding SAD values on GPU
      BlockMotionEstimation_CUDA<<<dimGrid,dimBlock>>>(cFrame_d,rFrame_d,widt
h,height,searchRange,SADValues_d);

      //synchronizing CPU and GPU
      cudaThreadSynchronize();

      //copying back the results
      cudaMemcpy(minSAD,minSAD_d ,3*sizeof(int), cudaMemcpyDeviceToHost);
```

Fig. 9 Host code to set up execution on GPU

Recent papers report the use of CUDA for motion estimation [7]. The results show that by efficiently using shared memory and other CUDA tools, a speedup of over 120 can be achieved.

4.4 Cloud computing for multimedia applications

Applications such as video analytics, semantics extraction, format conversion, and content adaptation require large computing resources and are typically performed on servers where abundant compute resources are available. Processing requirements vary depending on applications. If the requirement is to analyze a day's worth of surveillance video for suspicious events, the compute requirements are different depending on how soon the results are needed. If the turnaround time is not critical, the video can be processed on a single machine and processing time depends on the number of cores on the machine. On the other hand, if the results are needed as soon as possible, alternatives strategies need to be considered. The turnaround time can be reduced by using a machine with large number of cores. Machines with 8 processing cores are common today and as the number of processors is increased, we need specialized architectures that are expensive, difficult to program, and typically used for high performance computing applications. A cost effective alternative is the use of Cloud Computing.

4.4.1 Distributed computing with Hadoop

The recent emergence of Cloud Computing has allowed the access to large computing resources without having to acquire the necessary hardware. This availability can be

```
// Full Search Motion Estimation KERNEL code executed on the GPU

__global__ void BlockMotionEstimation_CUDA (unsigned char *pCur, unsigned
char *pRef, int nWidth, int nHeight, int nRange,unsigned int g_nMVx){

      unsigned int nMinSad = 0xFFFFFFFF, nSad;
      int nMinMVX, nMinMVY;

      // compute x position of MB from the
      // thread/block index and width/height
      int x = ((blockIdx.x * 32) * 16 ) % nWidth + (threadIdx.x * 16) %
nWidth;
      // compute y position of MB from the
      // thread/block index and width/height
      int y = ((blockIdx.x * 32) * 16 ) / nWidth + (threadIdx.x * 16) /
nWidth;

      // macroblock processed by the current thread
      int nBlockNumber = (nWidth/16)*y + x;

      for(int rx = x - nRange; rx < x + nRange; rx++){
         for(int ry = y - nRange; ry < y + nRange; ry++){
            for(int i = 0; i<16; y++){
               for(int j = 0; j<16; x++){
                  nSad += pCur[x+i][y+j] - pRef[rx+i][ry+j];
               }
            }
            if(nSad < nMinSad){
               nMinSad = nSad;
               nMinMVX = i;
               nMinMVY = j;
            }
          }
      }

      // motion vector with the best match (smallest cost)
      g_nMVx[nBlockNumber] = nMinMVX - x;
      g_nMVy[nBlockNumber] = nMinMVY - y;
}
```

Fig. 10 Device code computing motion estimation in parallel

achieved on a service-based type of interface where the users can make use of the available resources in an on-demand manner, reducing the implementation and maintenance cost to a single, usage based bill. Fueled by this “utility-computing” concept where computing resources are consumed the same way as electricity would, huge commercial endeavors such as Amazon Web Services were born. In addition, cloud computing has become an integral part of the operations at Google and Yahoo and Facebook, among others, processing vast amount of data (such as search logs for example) in a reduced time frame.

“Hadoop Map/Reduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner” [21]. It implements the MapReduce model suggested by Google in [10] but in free distributions such as Apache Hadoop, Yahoo Hadoop and Cloudera that are compatible and can leverage the power of Amazon Web Services, for example. Hadoop has been typically used for large scale tasks such as log file analysis and multimedia applications have been limited.

4.5 Multimedia computing using Hadoop

The published implementation of media processing on cloud environments has been focused specially on image processing, not video: Specifically, [24] and [4] discuss an example and an implementation respectively, of image processing tasks on cloud environments. In [6], this processing is actually achieved through a MapReduce implementation which estimates geographic information (where is the scene?) given a specific image, by leveraging a data set of around 6 million GPS-tagged images and using scene matching to find the most similar image through a reduce-less program. In the same fashion, [8] implements on MapReduce a set of three experiments in image processing, namely a Simple Pixel Detection test, a Blob Detection test and a Sobel Edge Detection test. One of the major examples of image processing/transformation on cloud environments is the New York Times archive conversion, where the TIFF images of the public domain scanned articles from 1851 to 1922 were converted to PDF format [22]: Around 11 million articles were converted and glued together in under 24 h processing using around 100 nodes of Amazon Web Services' EC2 instances using the MapReduce model. As far as video processing goes, fewer examples are available, one of them being HP Labs' VideoToon implementation [14], where a service is implemented to "cartoonize" videos with the help of Hadoop streaming.

Most of the multimedia processing using Hadoop has been for non-realtime applications. Recent work studied the use of Hadoop for video streaming applications such as HTTP Live Streaming [13] showing that Hadoop can potentially and effectively be used for real-time applications such as video adaptation.

5 Conclusions

The performance of multimedia applications can be improved by exploiting the parallel processing capabilities of the hardware infrastructure. The computing hardware today offers opportunities to parallelize software at different levels—instruction level, system level, and Cloud/cluster level. The selection of a parallel programming framework depends on the application needs and desired performance levels. OpenMP provides a quick way to get started with parallel programming by allowing incremental parallelism. Programming models such as CUDA offer higher performance gains using GPUs and works especially well for highly data parallel applications but requires significant efforts in porting code to run on them. CUDA programming. But while CUDA is specific to NVIDIA GPUs, a platform independent called OpenCl is evolving and is expected to provide a standard parallel programming model for CPUs and GPUs.On the other hand, large scale applications can see significant performance improvements using distributed frameworks such as Hadoop on the Cloud Computing infrastructure. These parallel programming techniques can be used together to maximize the performance, additionally, instruction level parallelism can be used to speedup code segments in tasks; a process can run tasks on multiple threads processing independent data blocksor it can be run on multiple nodes of a Cloud and operate on distributed data.

Parallel programming requires intimate knowledge of the problem in order to maximize the performance. Without domain knowledge, a programmer cannot understand dependencies and identify the parallelism inherent to the problem. Programmers have to learn to think parallel ad develop algorithms that can exploit the parallel hardware. Since parallel programmers cannot be expected to parallelize any application, it is imperative that domain expert training starts and multimedia programmers begin to venture in parallel programming. We expect parallel programming to become integral part of learning programming languages.

References

1. Asanovi'c K et al (2009) A view of the parallel computing landscape. Commun ACM 52(10):56–67
2. Borkar S (2007) Thousand core chips—a technology perspective. Proc. ACM/IEEE 44th Design Automation Conf (DAC). ACM Press, pp 746–749
3. Bulic P, Gustin V (2003) An extended ANSI C for processors with a multimedia extension. Int J Parallel Program 31(2):107–136
4. Chantry D (2009) Mapping applications to the Cloud. Microsoft Corporation—Platform Architecture Team. Online at: http://msdn.microsoft.com/en-us/library/dd430340.aspx
5. Chapman B, Jost G, van der Pas R (2007) Using OpenMP: portable shared memory parallel programming. MIT, October
6. Chen S, Schlosser SW (2008) Map-Reduce Meets Wider Varieties of Applications, IRP-TR-08-05, Technical Report, Intel Research Pittsburgh. Online at: http://www.pittsburgh.intel-research.net/~chensm/papers/IRP-TR-08-05.pdf
7. Colic A, Kalva H, Furht B (2010) Exploring NVIDA-CUDA for Video Coding. Proceedings of the first annual ACM SIGMM conference on Multimedia systems, pp 13–22
8. Conner J (2009) Customizing Input File Formats for Image Processing in Hadoop. Arizona State University. Online at: http://hpc.asu.edu/node/97
9. Conte G (2000) The long winding road to high-performance image processing with MMX/SSE. Proceedings of the Fifth IEEE International Workshop on Computer Architectures for Machine Perception, pp 249–258
10. Dean J, Ghemawat S (2004) MapReduce: Simplified Data Processing on Large Clusters. In Proceedings of OSDI '04: 6th Symposium on Operating System Design and Implemention, San Francisco, CA. Online at: http://labs.google.com/papers/mapreduce.html
11. Diefendorff K et al (2000) AltiVec extension to PowerPC accelerates media processing. IEEE Micro 20 (2):85–95
12. Ferretti M (2000) Multi-media extensions in super-pipelined micro-architectures. A new case for SIMD processing? Proceedings of the Fifth IEEE International Workshop on Computer Architectures for Machine Perception, pp 249–258
13. Garcia A Kalva H, Furht B (2010) Exploring A Study of Transcoding on Cloud Environments for Video Content Delivery. Proceedings of the Workshop on Mobile Cloud Media Computing, ACM Multimedia
14. HP Labs "VideoToon" Online at: http://www.hpl.hp.com/open_innovation/cloud_collaboration/cloud_demo_transcript.html
15. Kirk DB, Hwu WW (2010) Programming massively parallel processors—a hands on approach. Morgan Kaufmann
16. Krall A, Lelait S (2000) Compilation techniques for multimedia processors. Int J Parallel Program 28 (4):347–361
17. Lee RB (1996) Subword parallelism with MAX-2. IEEE Micro 16(4):51–59
18. Lee JB, Kalva H (2009) The VC-I and H.264 video compression standards: for broadband video service. Springer
19. Leupers, R (1999) Compiler Optimizations for Media Processors. In J-Y Roger, B Stanford-Smith, PT Kidd (eds) Business and Work in the Information Society: New Technologies and Applications, IOS Press, ISBN 90-5199-491-5
20. Luebke D, et al. (2004) Gpgpu: general purpose computation on graphics hardware. Proceedings of the conference on SIGGRAPH 2004 course notes (New York, NY, USA): ACM Press, p 33
21. "Map/Reduce Tutorial" Online at: http://hadoop.apache.org/common/docs/current/mapred_tutorial.html
22. New York Times (2007) Self-service, Prorated Super Computing Fun! Online at: http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/
23. Peleg A, Weiser U (1996) MMX technology extension to the Intel architecture. IEEE Micro 16(4):42–50
24. Trease H, Fraser D, Farber R, Elbert S Using transaction based parallel computing to solve image processing and computational physics problems. Online at: http://www.cca08.org/papers/Poster31-Harold-Trease.pdf
25. Tremblay M et al (1996) VIS speeds new media processing. IEEE Micro 16(4):10–20

Hari Kalva Dr. Kalva is an Associate Professor in the Department of Computer Science and Engineering at Florida Atlantic University. Dr. Kalva is an expert on digital audio-visual communications systems with over 15 years of experience in multimedia research, development, and standardization. He has made key contributions to the MPEG-4 Systems standard and also contributed to the DAVIC standards development. His current research activities include mobile video services, video transcoding, multi-view and 3D video coding, and multimedia programming. He is a recipient of the 2008 FAU Researcher of the Year Award and a 2009 ASEE Southeast New Faculty Research Award.

Dr. Kalva received a Ph.D. and an M.Phil. in Electrical Engineering from Columbia University in 2000 and 1999 respectively. He received an M.S. in Computer Engineering from Florida Atlantic University in 1994, and a B. Tech. in Electronics and Communications Engineering from N.B.K.R. Institute of Science and Technology, S.V. University, Tirupati, India in 1991.

Aleksandar Colic After receiving his Masters degree in December 2008 in Electrical and Computer Engineering at Faculty of Technical Sciences located in the city Novi Sad in his home country Serbia, he joined Florida Atlantic University as a PhD student in January 2009. He is closely working with his advisors Dr. Hari Kalva and Dr. Borivoje Furht at applying his knowledge of hardware and embedded system onto area of multimedia, devising efficient parallel systems and algorithms to boost performance and quality of video related processes such as video transcoding in today's top video system standards. His research interests include video compression, content adaptation and parallel systems.

Adriana Garcia Adriana Garcia is a recent Master's in Computer Engineering graduate from the Florida Atlantic University. She completed her undergraduate studies in Electronic Engineering at the Pontificia Universidad Javeriana in Bogota, Colombia and her research interests include cloud and parallel computing as well as embedded and mobile environments.

Borko Furht Borko Furht is a professor and chairman of the Department of Computer Science and Engineering at Florida Atlantic University in Boca Raton. He is also Director of the NSF-sponsored Industry/University Cooperative Research Center at FAU. His current research is in multimedia systems, video coding and compression, 3D video and image systems, video databases, wireless multimedia, and Internet and cloud computing. He has been Principal Investigator and Co-PI of several multiyear, multimillion dollar projects including the Center for Coastline Security Technologies, funded by the Department of Navy, One Pass to Production, funded by Motorola, the NSF PIRE project on Global Living Laboratory for Cyber Infrastructure Application Enablement, and the NSF funded High-Performance Computing Center. He is the author of numerous books and articles in the areas of multimedia, computer architecture, real-time computing, and operating systems. He is a founder and editor-in-chief of *the Journal of Multimedia Tools and Applications* (Springer, 1993). His latest books include "Handbook of Multimedia for Digital Entertainment and Arts" (Springer, 2009) and "Handbook of Media Broadcasting" (CRC Press, 2008). He has received several technical and publishing awards, and has consulted for many high-tech companies including IBM, Hewlett-Packard, Xerox, General Electric, JPL, NASA, Honeywell, and RCA, and has been an expert wetness for Cisco and Qualcomm. He has also served as a consultant to various colleges and universities. He has given many invited talks, keynote lectures, seminars, and tutorials. He serves on the Board of Directors of several high-tech companies.