Technical Report 2013

ECE 759 - High-Performance Computing for Applications in Engineering

Comparison of Parallel Programming Frameworks: Performance analysis and Tool-chain Support

Omkar Deshmukh, Shiva Prashant Chada

December 17, 2013

Table of Contents

Abstract

The motivation for this study was to explore different parallel programming frameworks made available on multitude of hardware systems. Depending upon the hardware support, the idea of executing a piece of code in parallel can span right from single CPU with multiple cores to heterogeneous computing using modern GPUs as co-processors. Therefore, in this study we included in our coverage following sets of approaches:

1. Multi-core Single Processor: SSE/AVX, OpenMP, OpenCL
2. Processor with Single GPU accelerator: CUDA, OpenCL

The core of this study was to understand the opportunities these approaches give us to parallelize or vectorize our program. Using these opportunities, our aim was be to get optimal performance out of our program and compare the speedup over the base version against other approaches. To this purpose, we implemented an algorithm that calculated 2D convolution of two square matrices.

The secondary aim of this study was also to look at, for each of these approaches, the effort involved in achieving the program speedup and the tool-chain support for these models. We believe that the presence of a strong ecosystem of development tools goes long way to augment the functionality of capable hardware.

INTRODUCTION

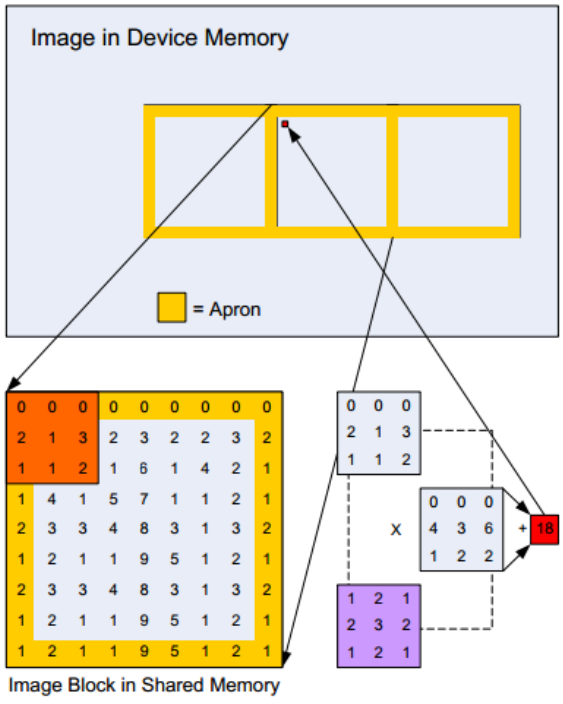
Problem Description

In Image processing, 2D convolution involves calculating an output image from the input image by means of a convolution kernel. The convolution kernel is a square 2D matrix with given number of rows and columns (typically 3x3, 5x5, 15x15, etc...). When the input image is processed, an output pixel is calculated for every input pixel by mixing the neighborhood of the input pixel according to the filter. In our case, the final value of the pixel is sum of the products with kernel elements. This type of convolution comprises of large number of independent floating point multiplications and summations. As such, the process conforms to the single instruction, multiple data (SIMD) paradaigm. Therefore, image convolution can be sped up leveraging the todays modern GPUs, which are designed to execute a huge number of floating point operations in parallel.[i]

This report describes the findings of parallel execution of convolution program on different set of hardware, each with its unique way of extracting data level parallelism. We start with single-thread performance and use it as a base performance. Then we parallelize the code with OpenMP, OpenCL and CUDA to understand how each of these programming models approach the target of parallelization and speedups they can provide.

Methodology

Here we have an 8×8 input image we want to convolve it with a 3×3 mask or kernel.



Each pixel of the resultant output image is generated by

- Placing the kernel over the input image, centered at the corresponding pixel location
- Weighting (multiplying) all the input image pixels covered by the kernel with the corresponding kernel values, and
- Accumulating (adding) all the results of step (b)

.

These set of operations are over increasing sizes of kernel and input images. Scaling the size of input images increases overall no. of computations involved and the associated data transfer delay. Thus with larger inputs, the problem is bound by memory. By scaling the kernel size, we increase the arithmetic intensity of computations. This way can check the computational performance of the hardware.

More specifically we scaled input image dimensions from 4K to 16K elements, keeping the kernel dimension to 8x8. On the other hand, while scaling the kernel, the input image was fixed to 8K x 8K in size with kernel dimension varying from 4 to 20 elements in each direction.

To run these tests, we had following set of systems at our disposal:

- OpenCL and AVX versions of the code were run on Intel Core i3 3227U CPU.
- OpenMP version of the code was run on AMD Opteron Processor 6274
- CUDA version of the code was run on Tesla K20Xm GPU.

PERFORMANCE

SSE / AVX

Starting with Sandy bridge, Intel processors have 256bit wide vector units which means that each CORE can perform certain operations on up to eight 32-bit floats or four 64-bit doubles per clock cycle. So, on a quad core you have 4 vector units (one per core) and could operate on up to 16 doubles or 32 floats per clock cycle.[ii] There are many ways to implement this form of vectorization such as auto-vectorization with compiler optimizations [iii], use of vectorized libraries etc. For this study we used intrinsic operations included in standard C library supported by GCC. The header "<immintrin.h>" provides interface to "_mm256_mul_ps" and "_mm256_hadd_ps" functions which are the core of the computation among other operations. These intrinsic operations map directly to underlying assembly instruction that can perform packed arithmetic calculations. For e.g, above instruction of multiplying 4 sets of floating point numbers translate to following set of operations:

```
prod = _mm256_mul_ps(data, kernel);
    a3f:      c5 fc 28 84 24 88 00      vmovaps ymm0,YMMWORD PTR [rsp+0x88]
    a46:      00 00
    a48:      c5 fc 28 4c 24 68         vmovaps ymm1,YMMWORD PTR [rsp+0x68]
    a4e:      c5 f4 59 c0               vmulps ymm0,ymm1,ymm0
```

Although the use of intrinsics reduced the overall execution time, the speedup was not quite what was expected. This in turn can be attributed to the fact that all types of compiler optimizations were disabled in order to accurately reflect the effect of changes we made. With these changes we observed that both SSE and AVX versions scaled in similar manner as base. For image scaling, SSE speedup was around 1.6 where as it was around 2.1 for AVX. Similar speedups were observed for kernel scaling. In each of these three cases, the execution benefits from similar set of instruction level parallelization techniques utilized by the CPU. Note that difference between SSE and AVX performance is not significant.

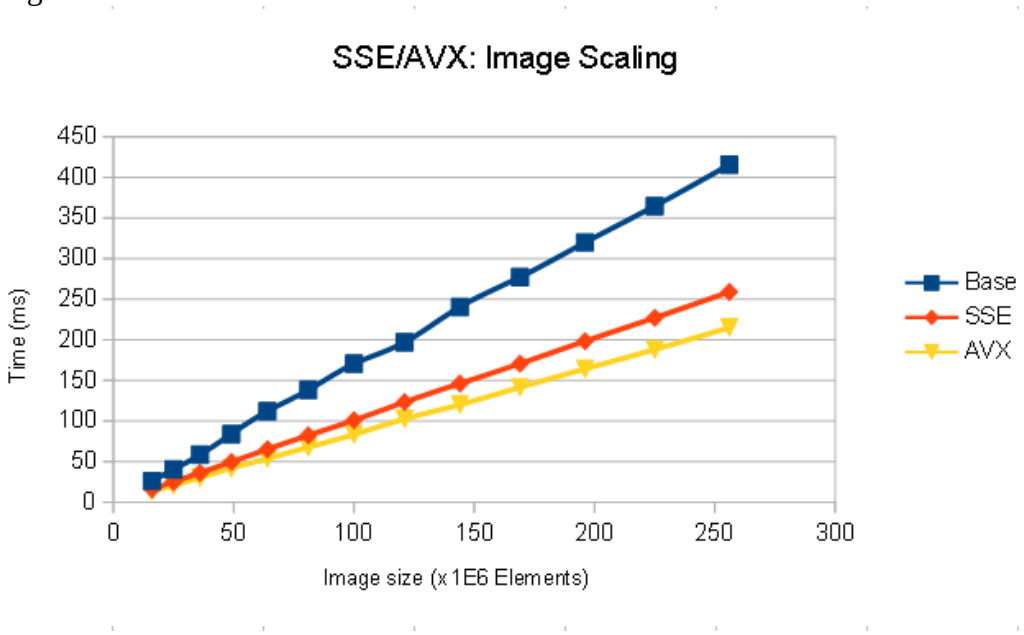


Figure 1: SSE / AVX Image Scaling - Timing Analysis

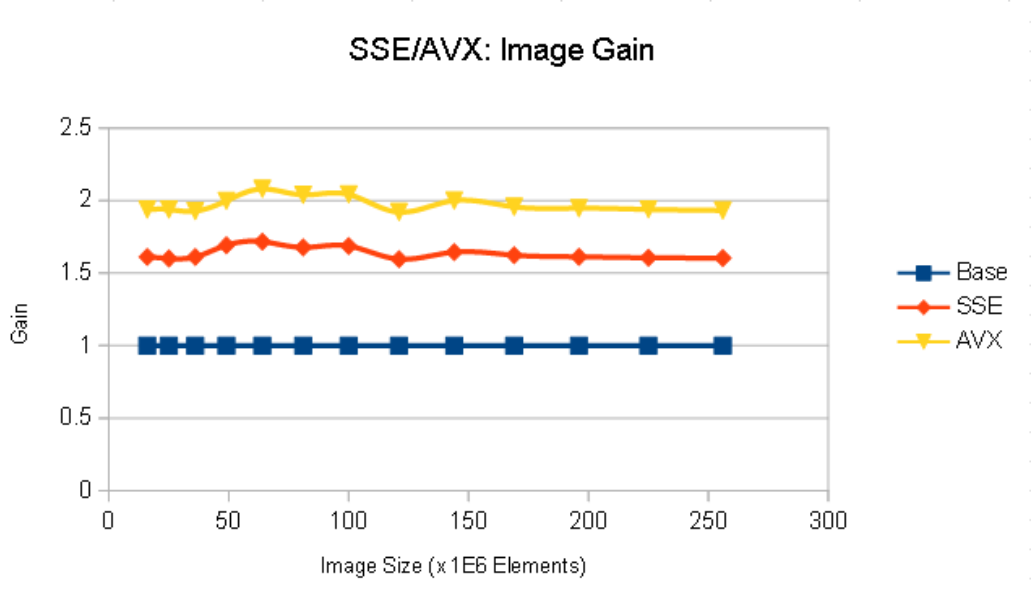


Figure 2: SSE / AVX Image Scaling - Gain

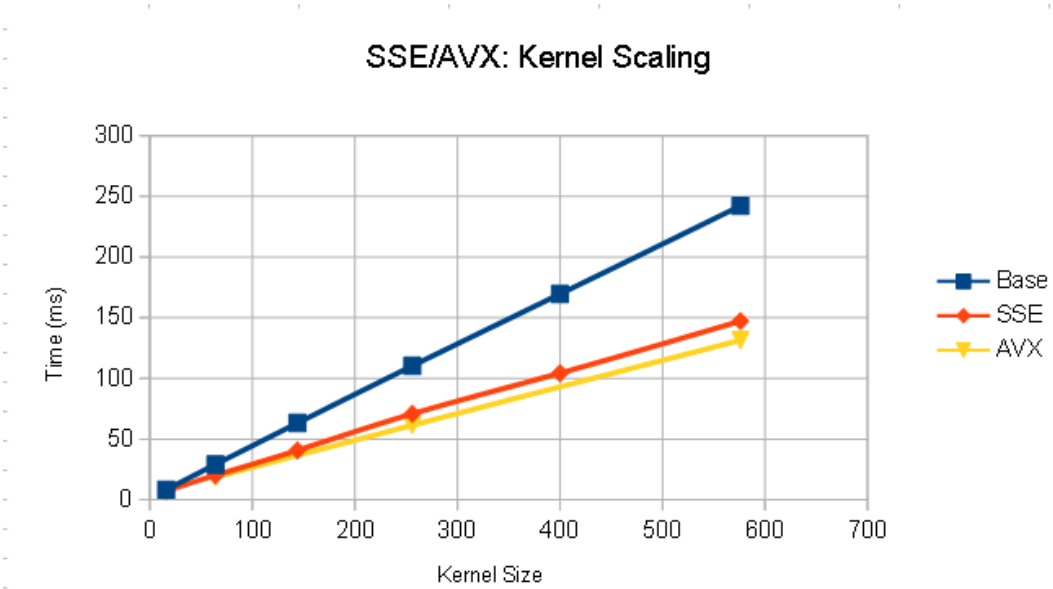


Figure 3: SSE / AVX Kernel Scaling - Timing Analysis

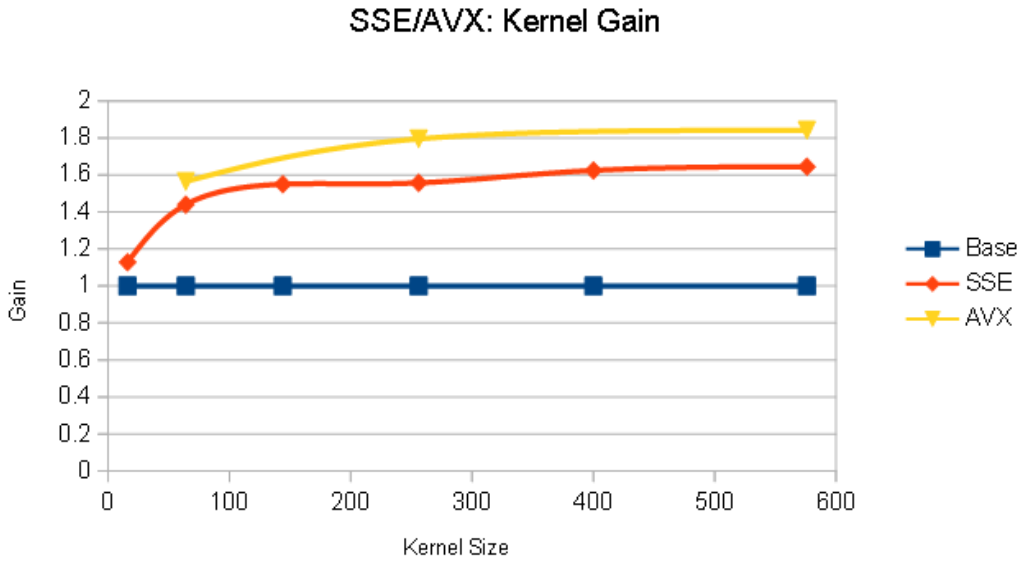


Figure 4: SSE / AVX Kernel Scaling - Gain

OpenMP

With OpenMP, the code remains same except for some compiler directives. Yet, with these simple changes and 64-core AMD Opteron Processor, considerable amounts of speedups were achieved.

Since the execution happens on single CPU, rather co-processor or networked node, the scaling profiles match those of base. The value of average gains for kernel scaling was less than that image scaling, although not by much. Finding out the cause behind this behavior would require more detailed analysis, though from initial observations, this

could be attributed to cache re-use behavior on single-threaded Vs. multi-threaded execution.

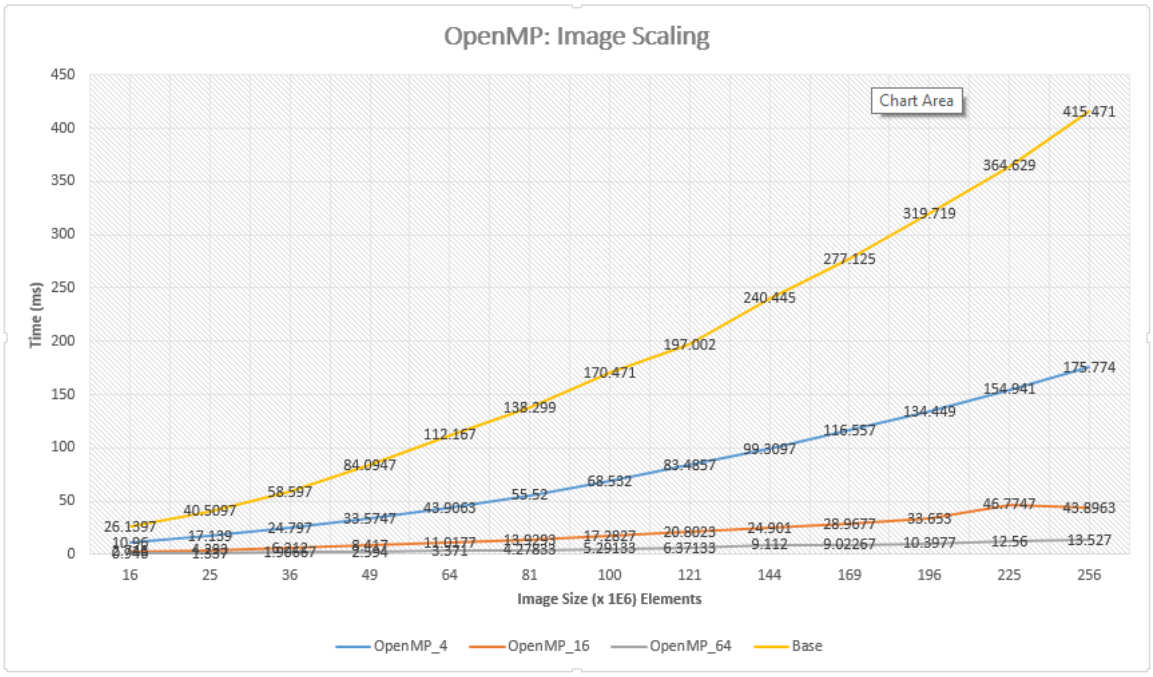


Figure 5: OpenMP Image Scaling - Timing Analysis

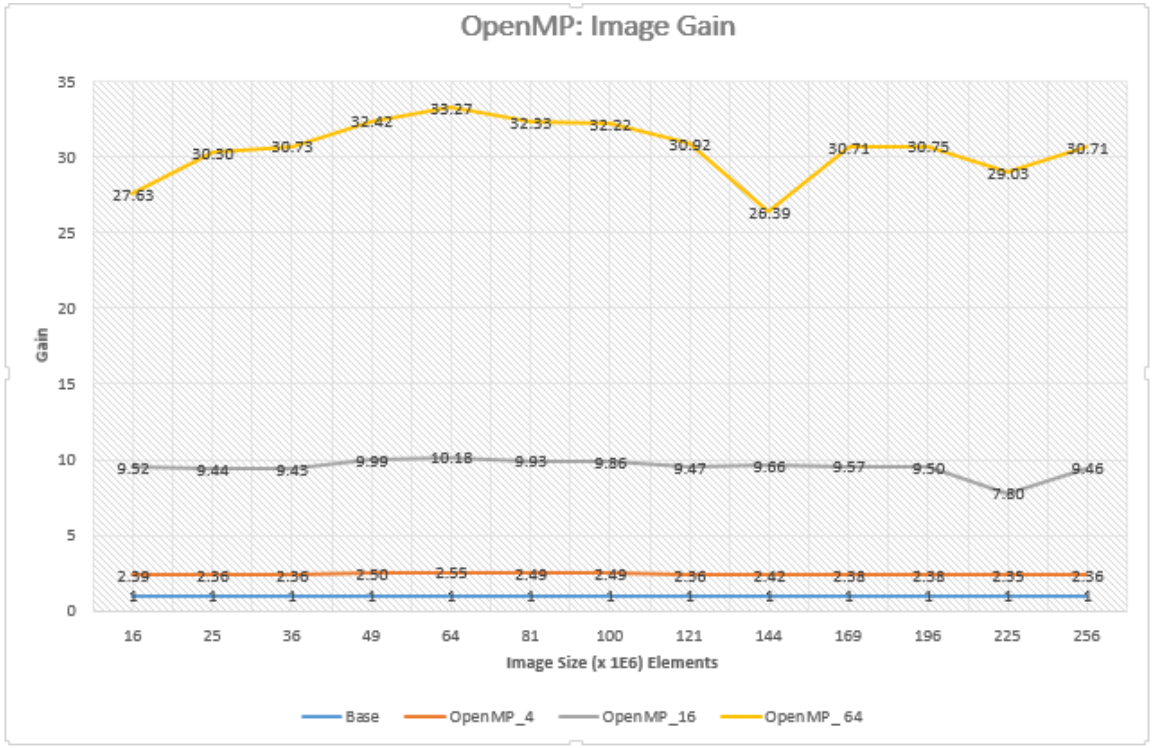


Figure 6: OpenMP Image Scaling - Gain

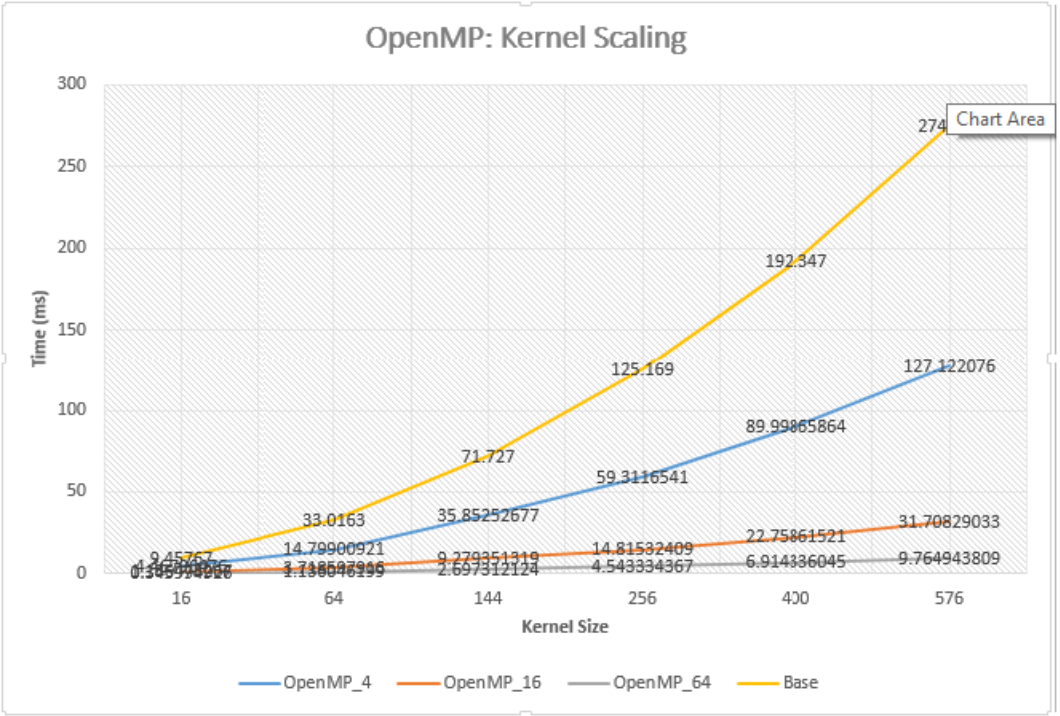


Figure 7: OpenMP Kernel Scaling - Timing Analysis

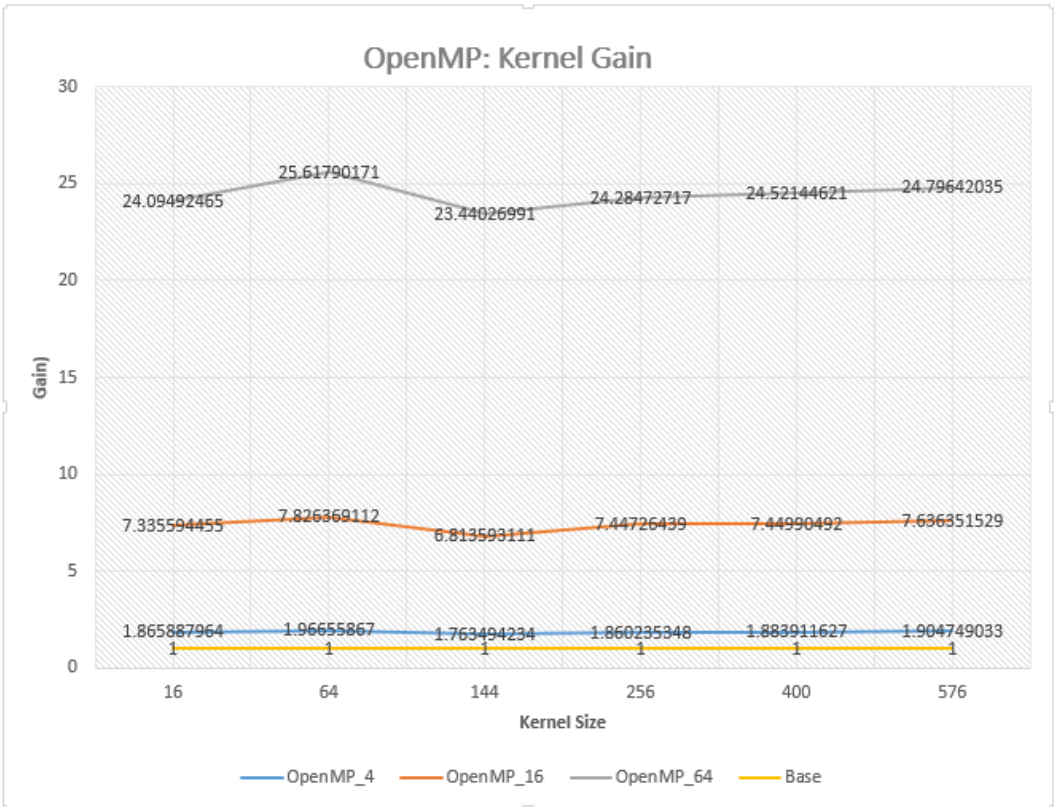


Figure 8: OpenMP Kernel Scaling - Gain

CUDA

For CUDA version, the implementation made use of the shared memory for input image and constant memory for kernel. The data grabbed from global memory was put in shared memory before the computation. Once all threads within a block are done copying data, the computation can begin. The computation will be identical and it happens on the data in the local memory, which is much faster than global memory. Also, since the kernel does not change during execution, instead of putting it in global memory, we can store it in constant memory. The constant memory is faster than global memory as it is pre-cached before kernel launch

We ran 2 versions of code. First one (called "normal") had CUDA kernel that was written by us and had the limitation of convolution kernel not being parameterized. For the second version, we ran tests with separable convolution code from CUDA SDK.[iv] This resulted in less number of operations per output pixel and greater speed-up for GPU executions. On the other hand, separable kernel for CPU results slightly worse execution times as compared to normal, possibly because normal version utilizes cache better. The separable version also allows for scaling kernel size. Such analysis shows that even though convolution throughput remains largely unchanged, larger kernel dimension yield bigger gains on GPU. This mean GPU scales much better when the problem is compute bound.

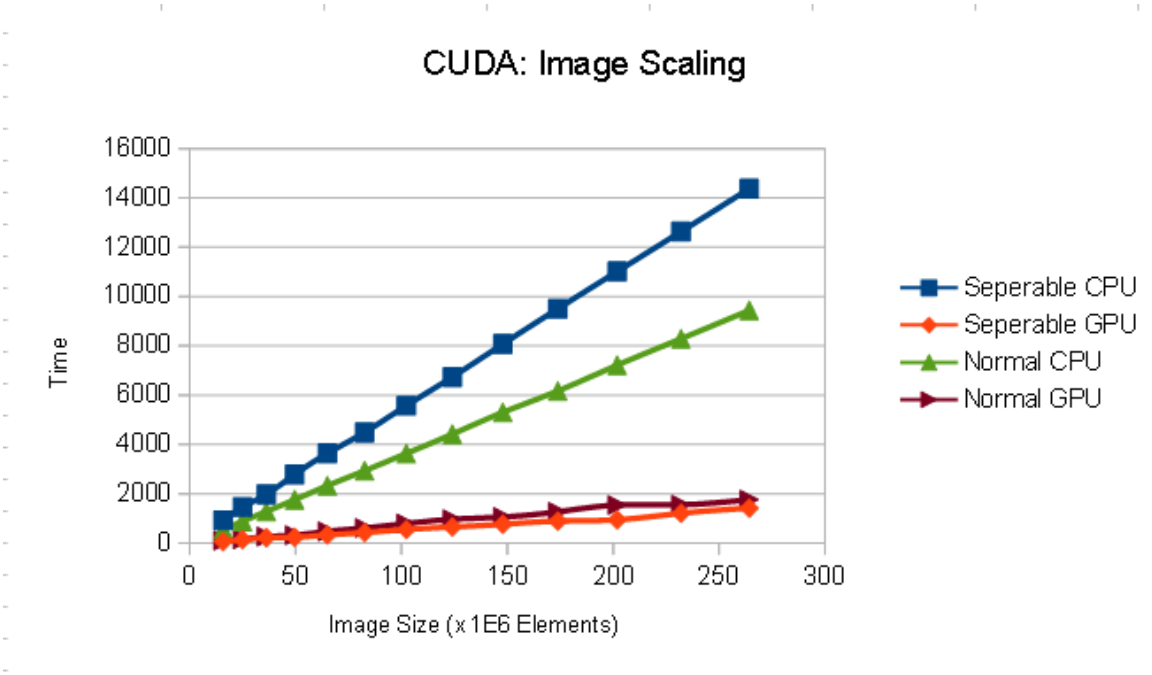


Figure 9: CUDA Image Scaling - Timing Analysis

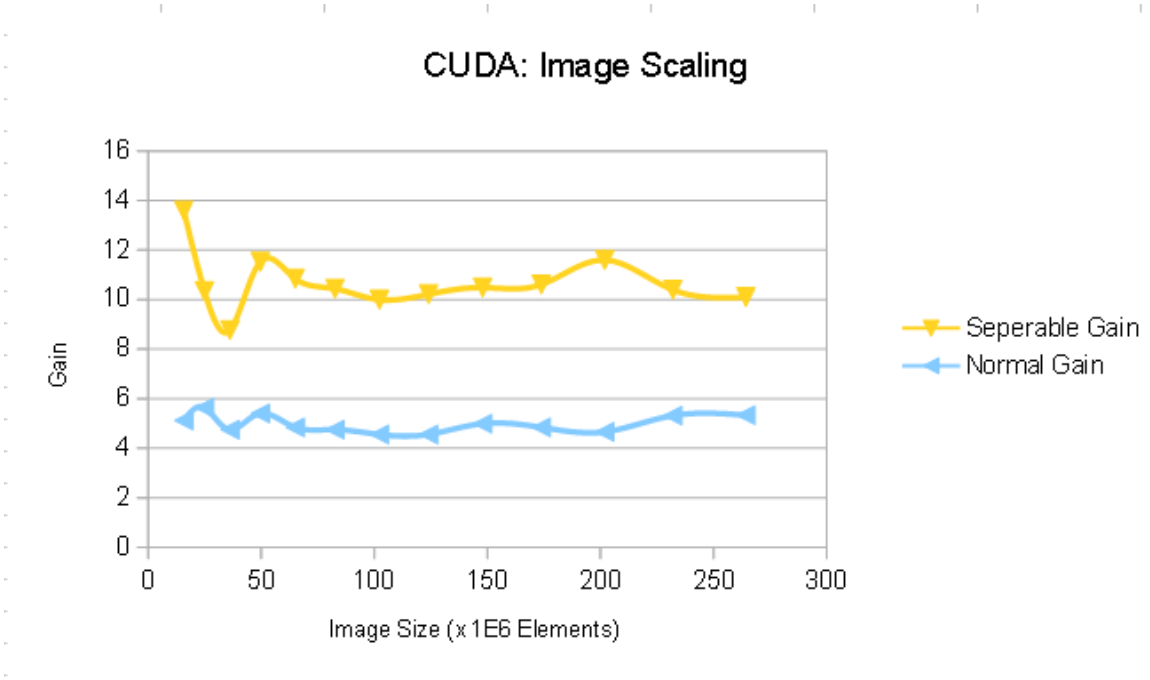


Figure 10: CUDA Image Scaling - Gain

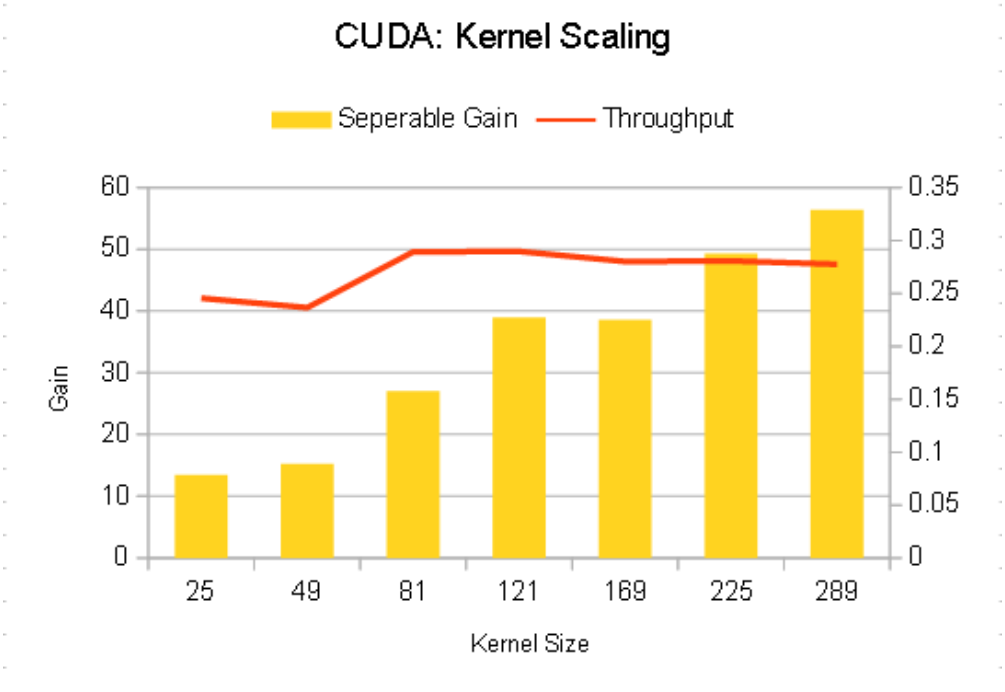


Figure 11: CUDA Kernel Scaling - Timing Analysis

OpenCL

OpenCL as a standard is now supported by wide variety of vendors and thus with little modifications it is possible to make the code portable. It supports multiple device classes such as CPUs, GPUs both embedded and dedicated, DSPs etc. For this study we initially planned to test the convolution runs on Intel HD Graphics 4000, an integrated GPU solution that ships with most desktop grade CPUs. However, due to some setup issues we were unable to do so. Instead we used OpenCL to run convolution on CPU itself. In this manner, CPU is presented to OpenCL as a multi-threaded execution unit, much like OpenMP's view of the processor. However, in contrast with OpenMP, OpenCL has support for utilizing SIMD units available on each core.[v] This way it is possible to use one packed-multiply and one packed-add to get the same result. This saves three multiply and three add instructions. As the saved instructions are in the innermost loop, speedups are noticeable.[vi] Moreover, unlike intrinsics which are more or less wrappers for in-lined assembly calls, this approach allows for better compiler optimizations.

In order to make use of this feature, we worked with AMD's implementation of OpenCL on x86 architecture.[vii] This implementation uses 'packed-arithmetic' instructions whenever it encounters a vector datatype in the kernel. For the convolution example, it uses float4 multiply and a float4 add in the inner loop.

The act of launching OpenCL kernel is much more complicated than its CUDA counterpart. The overall setup involves following set of steps:[viii]

- Bind to platform to determine availability of usable platforms.
- Query devices on the platform and select the required type (CPU, GPU etc.)
- Setup context for device which is used to manage queues, memories etc.
- Create command queue to stream commands from the host to device.
- Compile the kernel from source with just-in-time compilation.
- Create buffers between host and device.
- Launch Kernel
- Copy results back to host
- Cleanup.

Our tests were run on CPU itself, device type being CL_DEVICE_TYPE_CPU. This mean both our host and kernel shared the same memory space eliminating the need of moving data between buffers. For such setup buffers were created with CL_MEM_USE_HOST_PTR flag. Being run on quad-core CPU, scaling profiles were similar to that of OpenMP. Particulary, for image scaling, both vectorized and non-vectorized version posted consistent speed-ups and they proved to scale much better than basic version. As mentioned before, kernel scaling runs tested the performance in compute bound scenarios. In this case vectorized version provided near-theoretical 4 times the performance than non-vectorized version only at larger kernel sizes.

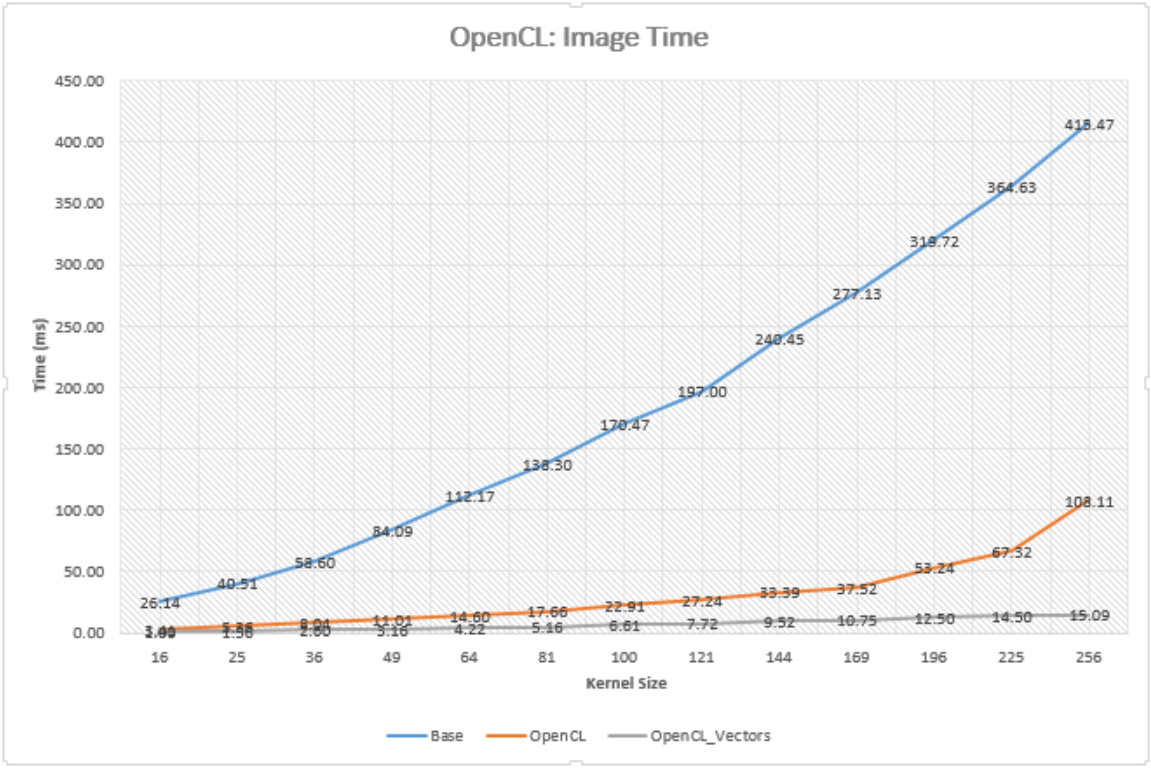


Figure 12: OpenCL Image Scaling - Timing Analysis

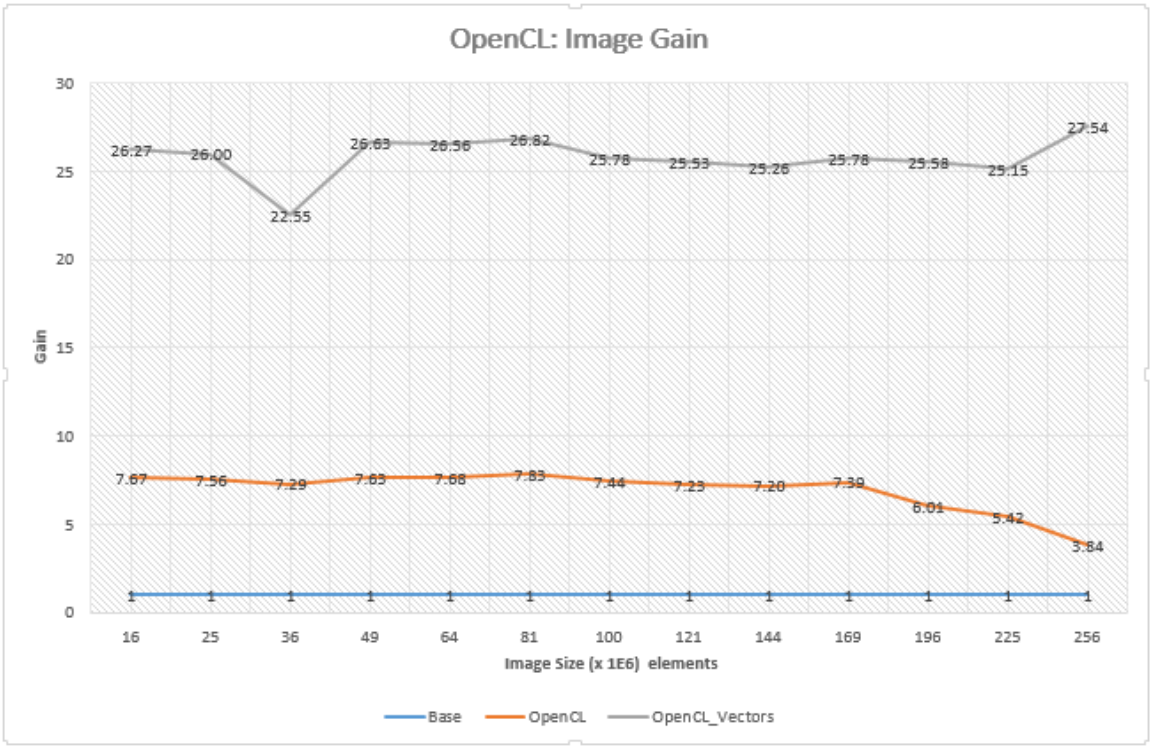


Figure 13: OpenCL Image Scaling - Gain

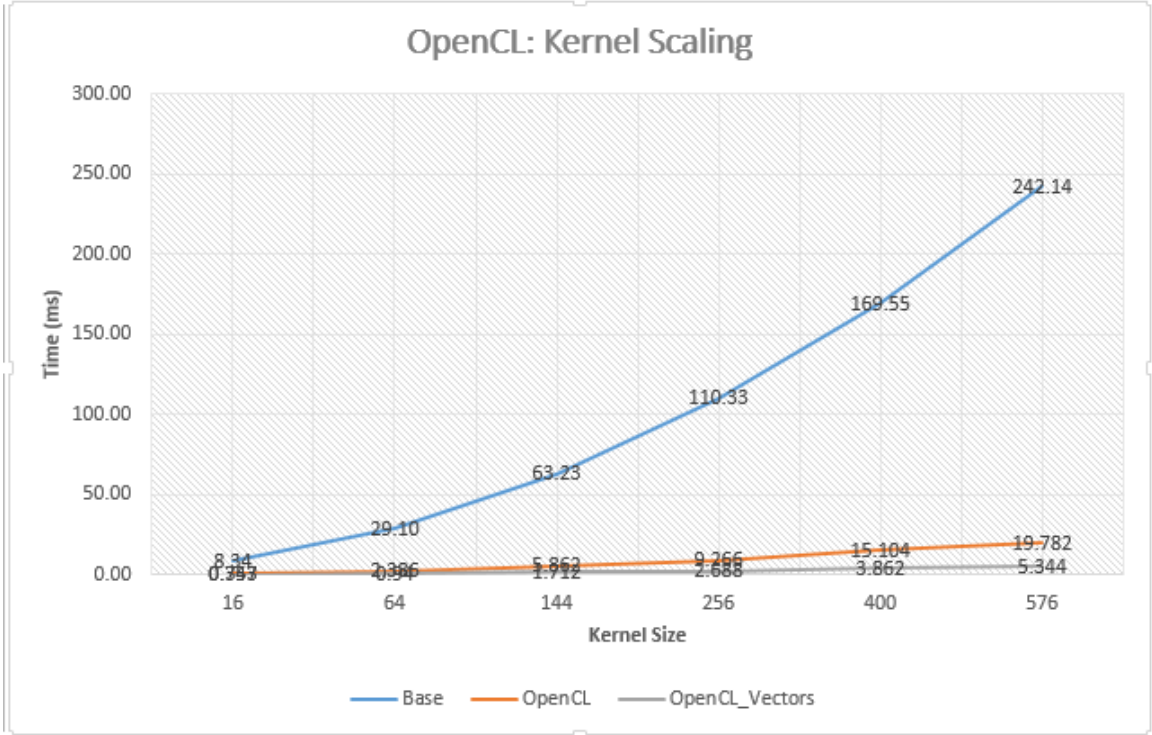


Figure 14: OpenCL Kernel Scaling - Timing Analysis

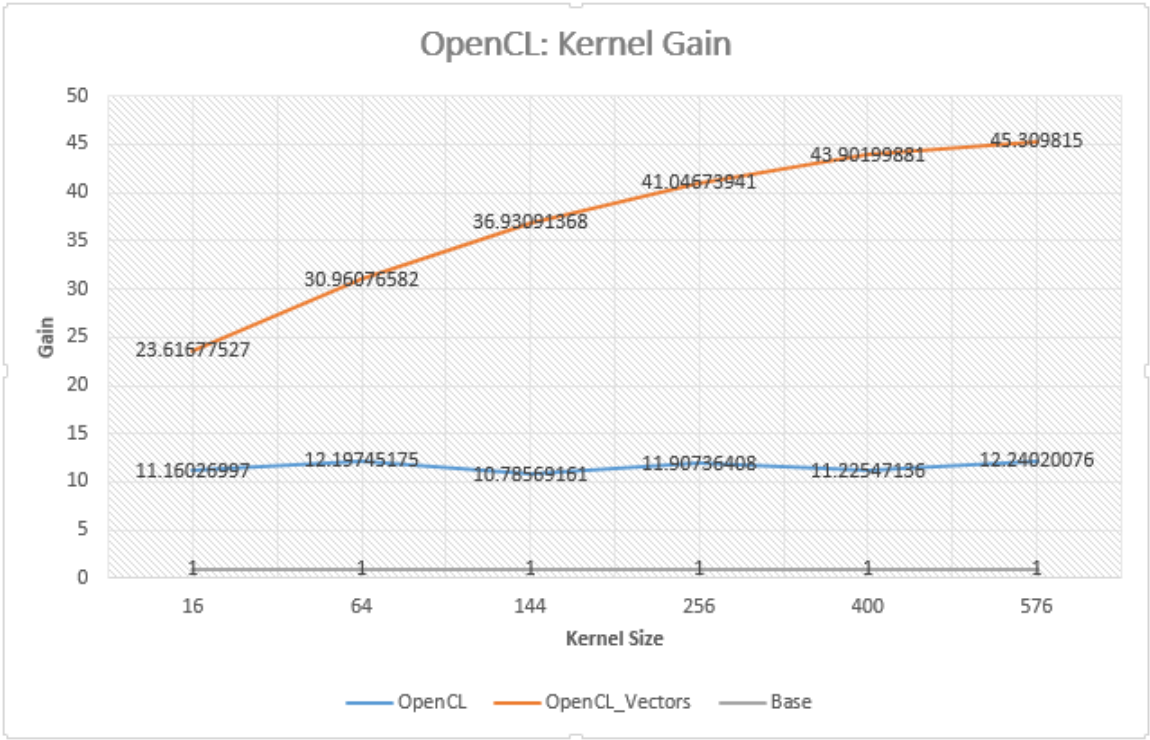


Figure 15: OpenCL Kernel Scaling - Gain

Development Environments

The secondary goal of this project was to understand development process with each of these approaches and study the tool-chain ecosystems and libraries made available to augment the development process.

As opposed to our approach, use of intrinsics is not the go-to way of making use of AVX for most developers. This is mainly because it results into more 'close to the metal' level of the code which leaves little room for compiler optimizations. To remedy this, many modern compilers can automatically vectorize C, C++ or Fortran code including gcc, PGI and icc. By adding appropriate compiler flags one can vectorize the code in question. In practice, vectorization isn't always automatic and the programmer needs to give the compiler some assistance but it is a lot easier than hand-coding intrinsics. On the other hand, the presence of vectorized libraries such as Intel AVX optimization in Intel Math Kernel Library (MKL), Intel Integrated Performance Primitives (IPP) Functions Optimized for AVX allows developers to work on higher levels of abstraction.[ix]

As an emerging platform, support for OpenCL is growing in both terms of compatible devices and availability of libraries for scientific computation. AMD and Intel have SDKs for OpenCL development on x86 platforms. NVidia supports programming their GPUs in OpenCL in addition to CUDA. ARM has Mali line of embedded GPUs that is conformant to OpenCL specifications. Altera became the first FPGA vendor to support OpenCL on their products. Thus it can be safely stated that interest in OpenCL is growing with ever growing developer community. On the HPC front, numerical libraries are made available by both industrial vendors like AMD in form of Accelerated Parallel Processing Math Libraries and institutions like Argonne National Laboratories (ViennaCL) and OpenCL Data Parallel Primitives Library (clpp).

Future Work

The use of CPU based vectorization is of particular interest to use. We plan to look into use of libraries such as Intel MKL and IPP for computational workloads such as simulation. Since these workloads are bigger in terms of complexity as compared to 2D image convolution, we plan to make use of profiling to tools to identify possibilities for vectorization. This approach can extended to study co-processors such as Intel Xeon Phi which support up to 244 threads per co-processor. This threads run on 61 cores residing on each card, each of which is modifies version of updated P54C core and handle 4 threads.[x]

References

[i] "A study of OpenCL image convolution optimization"
http://www.evl.uic.edu/kreda/gpu/image-convolution/

[ii] "Vectorising code to take advantage of modern CPUs (AVX and SSE)"
http://www.walkingrandomly.com/?p=3378

[iii] "Auto-vectorization in GCC"
http://gcc.gnu.org/projects/tree-ssa/vectorization.html

[iv] "CUDA Separable Convolution"
http://docs.nvidia.com/cuda/cuda-samples/index.html

[v] "The OpenCL power: offloading to the CPU (AVX+SSE)"
http://streamcomputing.eu/blog/2012-11-28/the-opencl-power-offloading-to-the-cpu-avx-sse/

[vi] "Image Convolution Using OpenCL"
http://developer.amd.com/resources/heterogeneous-computing/opencl-zone/programming-in-opencl/image-convolution-using-opencl/image-convolution-using-opencl-a-step-by-step-tutorial-5/

[vii] "AMD APP SDK Getting Started"
http://developer.amd.com/wordpress/media/2013/07/AMD_APP_SDK_Getting_Started_Guide.pdf

[viii] "OpenCL Vector Addition" https://www.olcf.ornl.gov/tutorials/opencl-vector-addition/

[ix] "Vectorising code to take advantage of modern CPUs (AVX and SSE)"
http://www.walkingrandomly.com/?p=3378

[x] "Intel's 50-core champion: In-depth on Xeon Phi"http://www.extremetech.com/extreme/133541-intels-64-core-champion-in-depth-on-xeon-phi