Openlab summer student report

Performance and threading studies of High Energy Physics software

Marc D'Arcy

CERN openlab
21st June - 14th August 2009

Supervisor: Andrzej Nowak

Distribution: Public

29th September 2009
Version 1.1

Openlab summer student report

Marc D'Arcy

21st June - 14th August 2009

Abstract

This report details some of the recent work carried out during a summer placement with CERN openlab. The report covers two main projects; firstly experience with Intel IPP and its integration into the High Energy Physics application, ROOT. This initial investigation suggests that Intel IPP can bring about some good increases in performance particularly on the Nehalem architecture, with minimal inconvenience incurred.

The second part of the report details the multi-threaded Geant4 prototype scalability testing carried out on the Nehalem architecture including perfmon analysis. The results suggest good potential for thread scalability on Nehalem, but some scaling problems are highlighted with the multi-threaded Geant4 prototype beyond 8 threads.

Contents

List of Figures

List of Tables

Glossary of technical terms

Perfmon2 Hardware based performance monitoring interface for the Linux kernel. Perfmon2 makes use of the Performance monitoring unit (PMU), which is hardware within the CPU collecting micro-architectural events on the pipeline, system bus, and caches. As well as providing access to these low-level counters, it also provides a high-level analysis facility.

Pfmon Performance monitoring client which works together with perfmon2 to monitor counters.

Pfmon deluxe A python script which facilitates easier pfmon usage.

Intel IPP Intel Integrated Performance Primitives (Intel IPP) is an extensive library of multicore-ready, highly optimised software functions for digital media and data-processing applications.

ROOT An object oriented framework for large scale data analysis, developed at CERN.

Geant4 A toolkit for the simulation of the passage of particles through matter, developed at CERN.

Multi-threaded Geant4 prototype A multi-threaded Geant4 prototype developed by Xin Dong and Gene Cooperman both of Northeastern University. The prototype includes the full CMS benchmark.

icc Intel C++ Compiler available for Linux, Microsoft Windows and Mac OS X. The current version at the time of writing and the one used throughout this report is 11.1.

Chapter 1

Introduction

1.1 Motivation

Optimisation is not a process that is performed only once, but rather a continual process that evolves with the latest software developments and hardware availability. In order to achieve the highest level of performance possible from the High Energy Physics applications used at CERN, openlab investigates a range of optimisation avenues.

With the recent developments in Intel processor architecture there are numerous openings to achieve additional levels of performance. This report will detail some of the work I have carried out in the area in recent weeks.

1.2 Aims and objectives

Having defined the motivation behind this study, the objectives can be summarised as follows:

- To investigate and evaluate Intel IPP.
- To integrate IPP functions to enhance the speed and performance of the zlib compression algorithm available within the ROOT framework.
- To evaluate the scaling performance of the multi-threaded Geant4 framework on Nehalem.

1.3 Report organisation

The report has been divided into two main chapters. The first of these chapters details the work undertaken with Intel IPP. The latter part of this chapter goes on to discuss the inclusion of Intel IPP into the High Energy Physics application; ROOT.

Chapter 3 covers some recent testing carried out with the multi-threaded prototype of the Geant4 toolkit.

Chapter 2

Intel IPP

2.1 Intel IPP

2.1.1 Overview of Intel IPP

Intel Integrated Performance Primitives (Intel IPP) is a library of software functions, produced and provided by Intel. The aim of IPP is to provide a highly optimised set of core functions which outperform what optimised compilers can deliver alone.

The library provides functions that cover a number of compute areas including general signal and image processing, computer vision, speech recognition, data compression, cryptography, string manipulation, audio processing, video coding, realistic rendering and 3D data processing [2]. By providing these low-level functions, it allows applications to be developed across a wide array of domains.

2.1.2 How IPP optimises for performance

IPP employs two main techniques in order to realise an increase in compute performance;

- Not using complicated data structures, and therefore reducing overall execution overhead.
- By utilising all the benefits of the Intel architecture on which the software is being run. For example through data-level parallelism via Intel SSE, as well as cache use optimisation. For current Nehalem CPUs the latest version of SSE is SSE4.2. On the next generation of Intel CPUs (Sandy Bridge) SSE will be extended with the introduction of Advanced Vector Extensions (AVX). This is a 256 bit instruction set which will largely benefit floating point intensive computations. The current release of IPP at the time of writing is 6.1 and introduces support for AVX. However, this report focuses on a comparison between the current Nehalem generation supporting SSE 4.2 and the legacy Penryn based Harpertown Xeon processors supporting SSE 4.1.

2.1.3 The future with Intel AVX

Following on from the point made about the introduction of the AVX instruction set in upcoming Intel processor architecture, we should consider how today's applications will be effected by the move. Article [4] details a number of key enhancements that AVX brings as well as some detail on how existing SSE instructions will be ported. From [4] the following points are worth noting here;

- A large number (200+) of legacy Intel SSEx instructions are upgraded by the enhanced instruction encoding to take advantage of features like a distinct source operand and flexible memory alignment.
- A moderate number (< 100) of legacy 128-bit Intel SSEx instruction have been promoted to process 256-bit vector data.
- A number of new data processing and arithmetic operations (< 100), not present in legacy Intel SSEx, are added to Intel processors to be launched in 2010 and beyond.

Also in [4], the following interesting points are highlighted with regards to developing with Intel AVX in mind;

- Most apps written with intrinsics need only recompile.
- There is a straight forward porting of existing Intel SSE to Intel AVX 256 with Intel libraries and Intel IPP.
- All Intel SSE/2 instructions are extended via simple prefix ("VEX").

2.1.4 Advantages and limitations of IPP

When evaluating a new technology for inclusion in existing software frameworks, as far as is possible the advantages and limitations should be made known from the start. Having read around numerous resources regarding IPP, this can be summarised as;

2.1.4.1 Advantages

- Allows for additional performance increase over standard compiled code.
- Cross-architecture support through a common API for all processors, with underlying function implementations accounting for variations in the processor architecture being used.
- All IPP functions are thread-safe in both dynamic and static libraries, and can therefore be used in multi-threaded applications.
- Common API across Windows, Linux and MacOS
- Support for both 32 and 64 bit processors.
- Using dynamically linked libraries, the latest additions in Intel processor architecture can be taken advantage of without recompiling.

Table 2.1: Intel IPP linkage method summary comparison [2]

Feature	Dynamic Linkage	Static Linkage with Dispatching	Static Linkage without Dispatching	Using Custom SO
Processor Updates	Automatic	Recompile & redistribute	Release new processor-specific application	Recompile & redistribute
Optimization	All processors	All processors	One processor	All processors
Build	Link to stub static libraries	Link to static libraries and static dispatchers	Link to merged libraries or threaded merged libraries	Build separate SO
Calling	Regular names	Regular names	Processor-specific names	Regular names
Total Binary Size	Large	Small	Smallest	Small
Executable Size	Smallest	Small	Small	Smallest
Kernel Mode	No	Yes	Yes	No

2.1.4.2 Limitations

- Unless using dynamic linking of libraries, recompiling is required to fully benefit from the latest hardware additions in future processors. This is summarised in table 2.1.

2.1.5 IPP availability at CERN

IPP is installed and maintained by CERN openlab, available on AFS. The current release at the time of writing, 11.1 038, can be found at;

`/afs/cern.ch/sw/IntelSoftware/linux/x86_64/Compiler/11.1/038/ipp/em64t/`

Further information regarding the setup of IPP as well as other Intel tools made available by CERN openlab can be found at;

`https://twiki.cern.ch/twiki/bin/view/Openlab/IntelTools`

In order to setup the IPP environment variables a script is provided, located at;

`/afs/cern.ch/sw/IntelSoftware/linux/x86_64/Compiler/11.1/038/ipp/em64t/tools/env/ippvarsem64t.sh`

Alternatively the environment variables can be setup manually, for which Chapter 3 of [2] should be consulted.

2.1.6 Compression algorithms within Intel IPP

As it has been briefly described already, IPP provides a broad array of low-level functions, applicable to a range of domains. With the size of the software frameworks in use at CERN there are numerous areas which IPP could be applied. Due to the limited amount of time available for this initial research into IPP, efforts have been focused on data compression. The reason for this being that the developers of ROOT, a widely used data analysis framework at CERN, identified that the compression part of the framework was consuming some 20-30% of the run-time.

This bottleneck presented an ideal platform on which to begin a study into the integration of IPP functions into an existing high energy physics application.

Shipped with Intel IPP 6.1 are a number of data compression implementations;

- Ipp_gzip - Implements effective loss-less data compression by utilising the Intel IPP data compression domain API. Dictionary-based IPP functions are used, implementing Lempel-Ziv (LZ77) algorithm and the original GZIP data formats. This results in compressed data formats being fully compatible with the original GZIP formats. [1] [2]

- Ipp_bzip2 - The use of IPP functions implemented over the top of bzip2/libbzip2; a program and library for loss-less, block-sorting data compression.

- Ipp_compress – Universal primitives for loss-less data compression using the IPP data compression domain (without support for existing formats). Includes but is not limited to, Huffman encoding, Run length encoding, move-to-front transform and burrows-wheeler.

- Ipp_zlib - Implements the same API as the latest version of the standard zlib library. Functions include well-known deflateInit, Deflate, deflateEnd, deflateReset inflateInit, Inflate, inflateEnd Adler32 and crc32 checksum calculating functions. Figures 2.1 and 2.2 describe respectively where IPP functions are used to perform the zlib compression algorithm.

In terms of additional performance increase alone the default choice would have been to implement a data compression function using the universal IPP compression primitives. This does however bring with it a major disadvantage in that there is no support for existing compressed formats. Since this work is primarily being carried out to enhance the compression part of the ROOT framework, the existing compression implementation should be taken into account.

2.1.7 Moving from Penryn to Nehalem and beyond without recompiling

By the nature of the work carried out at CERN, the workloads require the highest level of compute performance available. This therefore leads to the searching of performance enhancements from both software and hardware systems. Upgrade hardware arrives at the CERN computer centre on six month intervals, with the lifetime of the hardware being approximately three years. With frequent upgrades to hardware, the recompiling of software for each new architecture would be inconvenient. For this reason, the dynamic linking method has been used to include the IPP library files, the benefits of which are summarised in table 2.2.

[1] Although IPP GZIP is claimed to be fully compatible with GZIP, a bug has been discovered with the decompression of IPP gzipped files, as later detailed in this report.

[2] Also implemented in the sample code provided is a way of parallelizing an application using Intel Virtual Machine.

Figure 2.1: Encoding Data Block with Intel IPP ZLIB Functions [1]

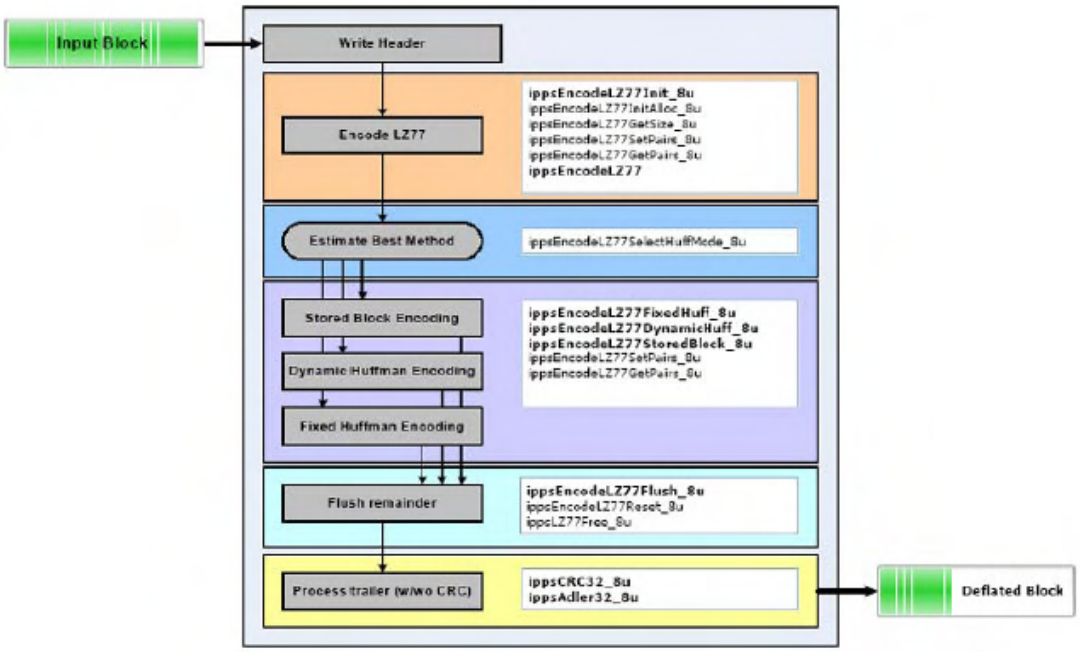

Figure 2.2: Decoding Data Block with Intel IPP ZLIB Functions [1]

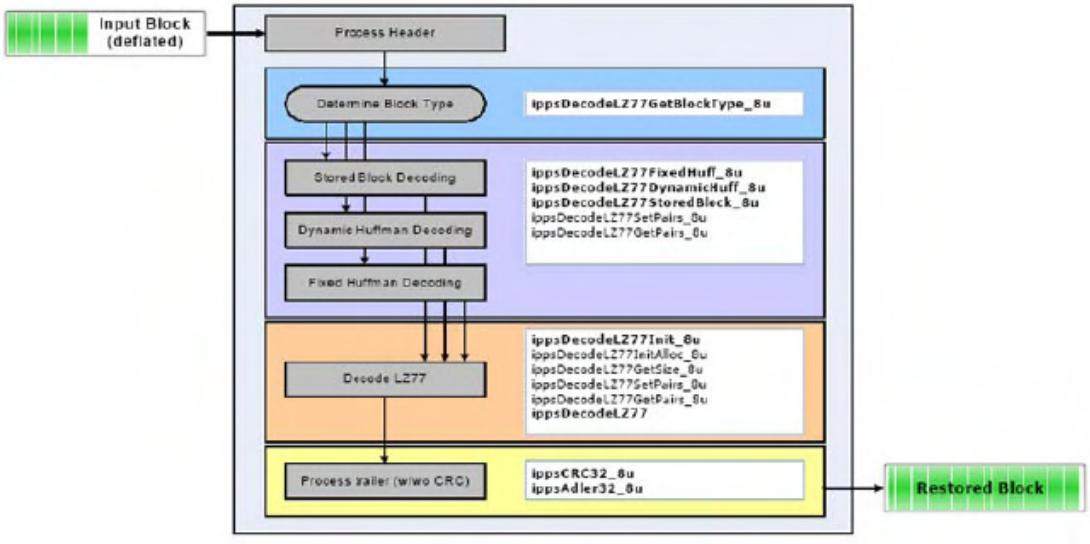

Table 2.2: Summary of dynamic linking features [2]

Benefits	Considerations
• Automatic run-time dispatch of processor-specific optimizations • Enabling updates with new processor optimizations without recompile/relink • Reduction of disk space requirements for applications with multiple Intel IPP-based executables • Enabling more efficient shared use of memory at run-time for multiple Intel IPP-based applications	• Application executable requires access to Intel IPP run-time dynamic link libraries (DLLs) to run • Not appropriate for kernel-mode/device-driver/ring-0 code • Not appropriate for web applets/plug-ins that require very small download • There is a one-time performance penalty when the Intel IPP DLLs are first loaded

2.2 Intel IPP within a real High Energy Physics application

2.2.1 Overview of ROOT

ROOT is an object oriented framework for large scale data analysis developed at CERN. It was originally for, and continues to be used for particle physics data analysis, although it's usage has now extended beyond this into other applications such as astronomy and data mining.

Since one of the main uses of ROOT is for data analysis and data acquisition it should store this data efficiency on disk; both in terms of disk space usage and ease and speed of access for the physicists.

2.2.2 The benefit of IPP integration within ROOT

IPP makes available a set of low-level fundamental functions, which live in many existing algorithms and applications. HEP applications are typically very large and complex with many years of development involved; one such example in the ROOT framework.

For this initial study into IPP, the data compression algorithm within ROOT has been targeted although in such an application there are many areas that could benefit from low-level optimisation, from data I/O to visualisation. Where possible Intel has used the same API for their IPP functions which replace existing functions to perform the same task. This can make the transition from standard code to code optimised for the latest Intel processors relatively quick while maintaining dependencies and functional compatibly.

2.2.3 Extending the current compression methods within ROOT

ROOT currently uses the latest version of the zlib 1.2.3 library [5–7], which includes an abstraction of the DEFLATE algorithm. A ROOT specific compression function has then been implemented in bits.h. The function shown in 2.1, R__zip, is an in-memory compression function and a variant of the standard in-memory R__memcompress function.

Algorithm 2.1 R__zip deflate function implemented within ROOT for compression

```
void R__zip(int cxlevel, int *srcsize, char *src, int *
    tgtsize, char *tgt, int *irep)

Input cxlevel = compression level
srcsize = size of input buffer
src = input buffer
tgtsize = size of target buffer
Output tgt = target buffer (compressed)
irep = size of compressed data
```

The R__zip function performs the following tasks in order;

1. Creates a zlib stream
2. Error checking on the target and source buffers
3. Uses the zlib lib to set the next input byte, number of bytes available at next_in, next output byte, remaining free space at next_out...etc
4. deflateInit() initialises the internal stream state for compression, passing the z_stream and compression level to be used
5. deflate() compresses as much data as possible, and stops when the input buffer becomes empty or the output buffer becomes full

There are currently two zlib implementations included in ROOT - the old zlib algorithm and the current standard zlib algorithm. The new IPP zlib based algorithm would therefore be implemented as a third, initially experimental algorithm. This approach ensures backward compatibility with existing files compressed with ROOT, while allowing the additions of IPP to be taken advantage of.

Since ROOT already implements the zlib compression function, the easiest and most compatible IPP implementation should be ipp_zlib. From [8] it is said that ipp_zlib implements the same API as standard zlib. The process of replacing it should therefore be a relatively simple exchange of the source and header files.

Another function worth highlighting is *void R__SetZipMode(int mode)*. Since ROOT already implements two compression algorithms, this function is used to switch between each algorithm. This function can be utilised to further extend the compression capabilities of ROOT to include the new IPP based compression algorithm(s). Continuing on from the current trend, this could be configured as;

(R__ZipMode = 1) The current latest zlib implementation

(R__ZipMode = 0) A legacy zlib implementation

(R__ZipMode = 2) New IPP based compression implementation

In terms of the enhancement of the compression capabilities in ROOT, there are two main objectives in the following order;

1. Achieve a significant decrease in the time taken for compression at run-time, by use of IPP data compression functions. This should then be presented as a proof of concept, to then consider the stability of the algorithm for inclusion in a future ROOT release.

2. Maintain the portability of the source code across Linux, Windows and Solaris operating systems[3].

2.2.4 Testbed configuration

Event is an example ROOT application, which shall be used as a benchmark, allowing the new IPP compression algorithm to be critically compared with those existing implementations. Event consists of a ROOT file with a tree and two histograms. The Event benchmark is located and can be run from `/root/test/Event`, together with the following steps;

1. Event is created by compiling MainEvent.cxx and Event.cxx, which is performed by running the included 'make' file.

2. During this testing the Event benchmark has been used as follows to perform compression and decompression respectively (where 1 denotes write and 20 denotes read);

```
./Event 5000 0 99 1

./Event 5000 0 99 20
```

Event can also take some optional arguments, as described in table 2.3.

2.3 Performance results and analysis

2.3.1 IPP gzip & zlib - Intel Harpertown vs Nehalem

In order to evaluate the compression algorithm implemented with IPP functions it has been compared against the standard gzip compression tool. A sample .tar file of 1.4 GB has been used which comprises of regular data. In fact, it is the source files for some application, and so the data represents realistic data that may be used. The results are included in table 2.4.

In each case listed in table 2.4, the highest level of compression was used; level 9. The following commands invoke each test in turn to perform compression followed by the decompression of the test file l_cproc_p_11.1.038_backup.tar.

[3]IPP currently supports Windows, Linux (including Moblin and QNX) and Mac OS. At the time of writing there is no support for Solaris within Intel IPP. It may however be possible to investigate the use of the static IPP libraries on Solaris if it uses the ELF object file format compatible with Linux.

Table 2.3: Arguments for the sample ROOT application; Event [3]

	Argument	Default
1	Number of Events (1 ... n)	400
2	Compression level: 0: no compression at all. 1: If the split level is set to zero, everything is compressed according to the `gzip` level 1. If split level is set to 1, leaves that are not floating point numbers are compressed using the `gzip` level 1. 2: If the split level is set to zero, everything is compressed according to the `gzip` level 2. If split level is set to 1, all non floating point leaves are compressed according to the `gzip` level 2 and the floating point leaves are compressed according to the `gzip` level 1 (`gzip` level –1). Floating point numbers are compressed differently because the gain when compressing them is about 20 - 30%. For other data types it is generally better and around 100%.	1
3	Split or not Split 0: only one single branch is created and the complete event is serialized in one single buffer 1: a branch per variable is created.	1 (Split)
4	Fill 0: read the file 1: write the file, but don't fill the histograms 2: don't write, don't fill the histograms 10: fill the histograms, don't write the file 11: fill the histograms, write the file 20: read the file sequentially 25: read the file at random	1 (Write, no fill)

```
time gzip -9 l_cproc_p_11.1.038_backup.tar

time gzip -d l_cproc_p_11.1.038_backup.tar.gz

time ./ipp_gzip -9 l_cproc_p_11.1.038_backup.tar

time ./ipp_gzip -d l_cproc_p_11.1.038_backup.tar.gz

time ./ipp_minigzip -9 l_cproc_p_11.1.038_backup.tar

time ./ipp_minigzip -d l_cproc_p_11.1.038_backup.tar.gz
```

From the results presented in table 2.4 a number of observations can be made. Looking at the Harpertown architecture alone, ipp_gzip achieves only a small performance increase over standard gzip. Interestingly, on the same architecture, zlib achieves a further performance gain of almost two fold for compression and 1.5x for decompression. Moving to Nehalem sees this increase further to 2.3x for compression and 3.2x on decompression. It can also be seen that on Nehalem, the ipp_gzip implementations out perform ipp_zlib, whereas on Harpertown the inverse is true.

Compiling the IPP code with icc on both Harpertown and Nehalem systems brings about additional performance optimisations resulting in faster compression at run-time. Finally, aside from the differences observed with regard to the compiler used, all implementations see some performance gain when porting to the Nehalem architecture.

2.3.2 IPP gzip decompression problems

During the testing of the compression algorithms utilising IPP a bug was discovered with the decompression part of IPP gzip. After conducting an in-depth investigation into the issue, a bug report was filed with Intel; the status of which is still pending.

At the time of writing only the compression part of IPP GZIP is functional as was intended. It was found that when attempting to decompress with IPP GZIP the results are of mixed success. Generally the smaller files would decompress without problems, while larger files would not. More dangerously, some files appeared to decompress correctly but the byte count did not match that of the original file before compression took place. This problem is isolated to ipp_gzip only, with ipp_zlib decompressing as expected.

For further details, the bug report which was filed with Intel has been included in the appendix.

2.3.3 ROOT vs ROOT with new IPP zlib compression algorithm

Presented in table 2.5 are the performance results for the de-/compression part of ROOT compared with the new IPP implementation. The results are based on modified ROOT source files compiled with icc running the Event benchmark on Harpertown architecture.

Table 2.4: gzip vs ipp_gzip on Intel Harpertown and Nehalem based systems
sample .tar file = 1.4 GB l_cproc_p_11.1.038.tar
icc –version = 11.1 20090511 gcc –version = 4.1.2 20080704

	Time (secs)			
	Real	User	System	Speedup[1]
Intel Xeon E5450 @ 3.00 GHz (Harpertown), icc 11.1				
gzip compress	79.790	76.485	3.184	0.98
gzip decompress	14.879	11.293	2.464	1.00
ipp_gzip compress	64.856	54.051	5.636	1.21 x
ipp_gzip decompress[2]	-	-	-	-
Intel Xeon E5450 @ 3.00 GHz (Harpertown), gcc 4.1.2				
gzip compress	78.381	75.489	2.804	1 (base)
gzip decompress	14.884	11.197	2.472	1 (base)
ipp_gzip compress	73.239	73.193	5.776	1.07 x
ipp_gzip decompress	-	-	-	-
ipp_minigzip (zlib) compress	39.349	35.758	2.404	1.99 x
ipp_minigzip (zlib) decompress	10.038	3.464	2.716	1.48 x
Intel Xeon X5570 @ 2.93 GHz (Nehalem), icc 11.1				
gzip compress	66.867	65.960	0.897	1.17 x
gzip decompress	10.096	9.270	0.825	1.47 x
ipp_gzip compress	27.312	82.925	3.653	2.87 x
ipp_gzip decompress	-	-	-	-
ipp_minigzip (zlib) compress	33.943	32.319	1.501	2.31 x
ipp_minigzip (zlib) decompress	4.653	2.814	1.690	3.20 x
Intel Xeon X5570 @ 2.93 GHz (Nehalem), gcc 4.1.2				
gzip compress	67.852	66.857	0.985	1.16 x
gzip decompress	10.241	9.505	0.735	1.45 x
ipp_gzip compress	30.732	112.161	3.144	2.55 x
ipp_gzip decompress	-	-	-	-
ipp_minigzip (zlib) compress	35.357	33.138	1.857	2.22 x
ipp_minigzip (zlib) decompress	4.656	2.867	1.650	3.20 x

[1] Based on real time with respect to gzip de-/compression on Harpertown compiled with gcc.

[2] Due to the bug discovered and reported regarding the decompression part of ipp_gzip, it has not been possible to include these figures in this report.

Using a compression level of 1, compression sees a 1.52x speedup in performance over the standard implementation. Increasing the compression level to the maximum of 9, the speedup is further increased to 2.25x. This is comparable to the two fold increase observed with ipp_zlib compression on Harpertown presented in 2.4. It is the level 1 compression that is of particular interest since it is the one most commonly used. The reason for this being that little disk space is saved by using the higher compression level, with a relatively high penalty paid in the compression time.

With the decompression times being relatively low with the standard implementation, there is little to no performance gain with ipp_zlib.

2.4 Conclusions

The results indicate that there is little performance gain when comparing ipp_gzip to standard gzip on Harpertown, whereas zlib does incur a good performance increase.

Moving to Nehalem a large performance increase is observed with ipp_gzip. It should be noted that this increase is not due to parallelization occurring since the functions used are non-threaded, and observing the CPU utilisation at run-time shows only a single core being used. For a complete list of the IPP functions being used in the gzip and zlib algorithms, [8, 9] should be consulted respectively with detailed information on each function found in Chapter 13 of Data Compression Functions of Intel® Integrated Performance Primitives Reference Manual: Volume 1 [1]. Although an increasing number of the IPP functions are becoming threaded with each release of IPP, none of the data compression functions used in ipp_gzip or ipp_zlib are threaded (using IPP 6.1) [10].

This suggests that there is either some problem with the Harpertown architecture and ipp_gzip or that ipp_gzip is taking advantage of additions in the instruction set on Nehalem architecture; namely SSE 4.2.

When evaluating the ipp_zlib compression algorithm within ROOT on Harpertown a good increase in performance has been achieved, which mirrors the performance gain observed for the standalone implementations outside of ROOT. This study should be further extended to include ROOT running on Nehalem with both icc and gcc compiled code.

2.5 Further work

- Extending the study to include testing of the ROOT framework with IPP based compression on Nehalem architecture.
- Evaluating other IPP based compression functions, particularly the universal IPP functions within ROOT.
- Identifying and applying IPP functions to other parts of ROOT beyond data compression. Through further investigation IPP could also be applied to

Table 2.5: Performance results for standard ROOT vs ROOT with new IPP based zlib compression algorithm on Harpertown

	File Size (bytes)	Tree compression factor	RealTime	CpuTime	Mbytes / Realtime seconds	Mbytes / Cputime seconds	Speedup
Write (compress)							
ipp event_5000_0_99_1	356406277	1.03	22.61	5.43	15.60	64.93	1.40 x
ipp event_5000_1_99_1	151518257	2.43	13.12	9.59	26.87	36.77	1.52 x
ipp event_5000_9_99_1	140939892	2.61	38.75	36.85	9.10	9.57	2.25 x
event_5000_0_99_1	356451045	1.03	31.56	6.76	11.17	52.16	-
event_5000_1_99_1	149054839	2.47	19.93	15.43	17.69	22.85	-
event_5000_9_99_1	140933731	2.61	87.18	86.16	4.04	4.09	-
Read (de-compress)							
ipp event_5000_0_99_20	-	-	3.80	3.8	92.67	92.78	0.61 x
ipp event_5000_1_99_20	-	-	3.81	3.81	92.60	92.54	1.05 x
ipp event_5000_9_99_20	-	-	3.82	3.82	92.35	92.30	0.99 x
event_5000_0_99_20	-	-	2.31	2.31	152.62	152.63	-
event_5000_1_99_20	-	-	4.00	4.01	88.10	87.93	-
event_5000_9_99_20	-	-	3.78	3.79	93.15	93.03	-

many other software applications and frameworks currently in development at CERN.

- Building minimal custom dynamic shared libraries specific to the IPP functions required by ROOT. Refer to 'Building a Custom SO' in [2] for further information.

Chapter 3

Multi-threaded Geant4 scalability testing

3.1 Overview of Geant4

Geant4 is described by it's developers as

> "... a toolkit for the simulation of the passage of particles through matter. Its areas of application include high energy, nuclear and accelerator physics, as well as studies in medical and space science."

For further information on the toolkit itself, the reader should consult the two main reference papers published in [11] and [12].

In addition to the standard Geant4 toolkit there exists a multi-threaded Geant4 prototype. The prototype has been developed by Xin Dong and Gene Cooperman both of Northeastern University. The prototype includes the full CMS benchmark, which shall be used to perform this scaling performance testing.

3.2 Objective of scalability testing

One of the key activities within openlab is to plan for the inclusion of future released hardware into the computer centre. This is achieved by evaluating the latest hardware platforms and simulating future software and hardware configurations to the best ability at the current point in time.

It has become apparent in recent years that it is no longer possible to reap the performance increases on processor frequency scaling alone. Instead, we now see each new processor architecture including an increasing amount of cores. Available today are multi-core processors for which the high-end models from Intel are typically clocked at around 3.0 GHz. The CERN computer centre currently runs mostly quad-core Intel Xeon E5450 @ 3.00GHz (Harpertown) in a dual-socket system totalling 8 cores for a single system. Within CERN openlab the hardware is more recent, currently experimenting with Intel Xeon X5570 @ 2.93GHz (Nehalem), again in a dual-socket system providing 8 cores and 16 hardware threads for a single system.

It is clear that the current trend of multi-core systems will continue on to become many-core systems. Within the next couple of years it is quite possible that we will

be dealing with systems with tens to hundreds of relatively low (not increasing much beyond today's clock speeds) clocked cores. For example, Intel has recently published [13] which details a GPU-type architecture built on x86 micro-architecture. At the time of writing details on Larrabee are rather limited, but it is evident that this will be a many-core x86 architecture with a coherent cache, as well as GPU-like qualities such as wide SIMD vector units and texture sampling hardware. Where Larrabee will differ greatly from today's GPUs is that it will contain little specialised graphics hardware but rather perform these graphics tasks in software. Although Larrabee seems to be initially targeted at the current GPU/gaming market, it's flexible many-core architecture will also prove very useful to the high performance computing (HPC) market.

Aside from multi-core technology, the latest generation of Intel CPUs also reintroduce Simultaneous Multi-threading (SMT) under the same name it was previously seen; Intel Hyper-Threading Technology. This was last seen on the Intel Pentium 4 processor, where its effective performance was questionable. More recently SMT has been reintroduced in today's Intel Atom and Nehalem/Core i7 based processors, and all indications point to hyper-threading being present in future Intel processor architectures.

This is hardware level support to efficiently execute multiple threads. The aim is to increase the utilisation of a single core by leveraging thread-level and instruction-level parallelism, resulting in increased performance on multi-threaded software.

Although multi-threading often results in increased performance, it should not be preempted since it brings with it a number of additional considerations. It can be advantageous since unused resources can be utilised by other threads while a particular thread may be receiving a lot of cache misses for example, or just simply unable to fully utilise all available compute resources with the current workload. As well as this, if multiple threads are using the same dataset, the cache can be shared between all the threads resulting in more efficient use of the cache.

On the other hand, running multiple threads can interfere with each other when sharing hardware resources such as the cache and translation lookaside buffers (TLBs). Multi-threading can also have a detrimental effect on the execution times of individual threads due to reduced clock frequencies and additional pipeline stages.

With the aid of perfmon2, the Performance monitoring unit (PMU) in the CPU can be used to monitor micro-architectural events; providing detailed information on the effective use of the available multi-core, multi-threaded hardware.

3.3 Results

3.3.1 Single run benchmarks on Nehalem

Table 3.1 details the results of individual runs of the MTGeant4 prototype on a Nehalem based system, using 1 to 16 threads. These results were recorded without running perfmon2.

Looking at the results there are a couple of interesting points to note. The total run-time for the benchmark decreases as the number of threads is increased from one to eight. Eight threads results in the optimal run-time on this eight core

Table 3.1: Single run scaling performance testing of MTGeant4, running on a Nehalem based Xeon X5570 8 core system

Dual-socket Intel Xeon X5570, SLC5, 8 cores total, 16 HW threads. gcc 4.3.3, 64-bit, 24 GB memory (opladev27). tcmalloc 1.3, 500 pi- events	1	2	4	8	12	16
Total run-time	3887	2028	1265	842	899	878
Initialisation	271	238	271	238	237	239
Worker initialisation (max)	140	142	146	151	206	224
Worker simulation time (max)	3476	1649	852	456	493	418
Total worker time (max)	3616	1790	993	604	662	639

Table 3.2: Time [s] output by MTGeant4 running on a Nehalem based Xeon X5570 8 core system

Dual-socket Intel Xeon X5570, SLC5, 8 cores total. gcc 4.3.3, 64-bit, 24 GB memory (opladev27). tcmalloc 1.3, 500 pi- 300GeV evts total	1	2	4	8
Time [s] as output by MTGeant4	3296	1861	1048	613

system, after which 12 and 16 threads see an increase in the run-time. It can also be seen from the results that the initialisation time remains constant at around 238 seconds, apart from when running one thread and four threads. Other runs have seen this reduced slightly to 254 seconds but further investigation is required to fully understand why this is occurring.

As expected, the time taken to initialise the worker threads increases as the number of threads increases.

3.3.2 Batch script with perfmon on Nehalem

Table 3.2 shows the time in seconds taken to execute the MTGeant4 benchmark. The time displayed is that output by the benchmark itself as the 'time for the simulation'. These results were recorded while also taking perfmon2 counter readings. The full table of results including the perfmon counters can be found in the appendix.

Looking at the perfmon output it can be seen that both the UNHALTED_CORE _CYCLES and INSTRUCTIONS_RETIRED counts remain relatively constant as the number of threads is increased from one to eight. This suggests that good scalability exists at a micro-architectural level.

The output also indicates that the LAST_LEVEL_CACHE_MISSES count increases considerably as the number of threads is increased from one to two. This is normal and as expected, due to the overhead incured by the introduction of

multiple threads. An interesting point to note however is that when going from two to eight threads, this count increases by 50%. Despite this, it should be noted that the last level cache miss percentage is still relatively low even with eight threads.

The RESOURCE_STALLS:ANY count does not increase significantly as the number of threads is scaled from one to eight. However, it can be seen from the perfmon output that the RESOURCE_STALLS:LOAD, RESOURCE_STALLS:STORE and RESOURCE_STALLS:ROB_FULL count does see an increase when scaling from one to eight threads; which suggests a memory pressure.

3.4 Conclusions

This chapter has seen some scaling performance results including perfmon counts for the multi-threaded Geant4 application.

The perfmon output indicates that the Nehalem micro-architecture has good potential for thread scalability. By comparison to similar studies that have been carried out on the Harpertown architecture in the past, it would appear from these initial studies that Nehalem performs better in terms of thread scalability than the legacy Harpertown architecture.

For the single runs of the benchmark on Nehalem some hardware threading testing has been included for 12 and 16 threads. Observing the total worker time, a 9.6% penalty can be seen when moving from 8 to 12 threads. From 8 to 16 threads there is also a penalty incurred of 5.8%. This suggests a software problem since the use of SMT should typically result in around a 25% performance increase.

Further investigation should be performed to gain a full picture of the thread potential of Nehalem, particularly for the multi-threaded Geant4 prototype. Recommendations for this continued work have been outlined in the next section.

3.5 Further work

1. These studies should be continued to include hardware threading together with perfmon to gain a better insight into how the application scales beyond the physical number of cores. This has not been included due to some technical difficulties resulting in a lack of time to collect and analyse the results.

2. The studies conducted here can be further extended to execute a fixed number of events per thread.

Bibliography

[1] Intel Corporation. *Intel® Integrated Performance Primitives for Intel® Architecture Reference Manual, Volume 1: Signal Processing*, a24968-025us edition, March 2009.

[2] Intel Corporation. *Intel® Integrated Performance Primitives for Linux* OS on IA-32 Architecture User's Guide*, 320271-003us edition, March 2009.

[3] ROOT team, CERN, ftp://root.cern.ch/root/doc/20TutorialsandTests.pdf. *The Tutorials and Tests*, August 2009.

[4] Intel® avx: New frontiers in performance improvements and energy efficiency. Technical report, Intel Corporation, http://software.intel.com/en-us/articles/intel-avx-new-frontiers-in-performance-improvements-and-energy-efficiency/, March 2008.

[5] P. Deutsch and J.-L. Gailly. Zlib compressed data format specification version 3.3, 1996.

[6] P. Deutsch. Deflate compressed data format specification version 1.3, 1996.

[7] P. Deutsch. Gzip file format specification version 4.3, 1996.

[8] Intel Corporation. *Intel® Integrated Performance Primitives Ipp zlib Sample for Linux*, ipp 6.1 edition, August 2009.

[9] Intel Corporation. *Intel® Integrated Performance Primitives Data Compression-based IPP GZIP Sample for Linux*, ipp 6.1 edition, August 2009.

[10] Intel Corporation. *Threaded Functions List*, intel ipp 6.1 edition, August 2009.

[11] "S. Agostinelli, J. Allison, K. Amako, J. Apostolakis, H. Araujo, P. Arce, M. Asai, D. Axen, S. Banerjee, G. Barrand, F. Behner, L. Bellagamba, J. Boudreau, L. Broglia, A. Brunengo, H. Burkhardt, S. Chauvie, J. Chuma, R. Chytracek, G. Cooperman, G. Cosmo, P. Degtyarenko, A. Dell'Acqua, G. Depaola, D. Dietrich, R. Enami, A. Feliciello, C. Ferguson, H. Fesefeldt, G. Folger, F. Foppiano, A. Forti, S. Garelli, S. Giani, R. Giannitrapani, D. Gibin, J. J. Gómez Cadenas, I. González, G. Gracia Abril, G. Greeniaus, W. Greiner, V. Grichine, A. Grossheim, S. Guatelli, P. Gumplinger, R. Hamatsu, K. Hashimoto, H. Hasui, A. Heikkinen, A. Howard, V. Ivanchenko, A. Johnson, F. W. Jones, J. Kallenbach, N. Kanaya, M. Kawabata, Y. Kawabata, M. Kawaguti, S. Kelner, P. Kent, A. Kimura, T. Kodama, R. Kokoulin, M. Kossov, H. Kurashige,

E. Lamanna, T. Lampén, V. Lara, V. Lefebure, F. Lei, M. Liendl, W. Lockman, F. Longo, S. Magni, M. Maire, E. Medernach, K. Minamimoto, P. Mora de Freitas, Y. Morita, K. Murakami, M. Nagamatu, R. Nartallo, P. Nieminen, T. Nishimura, K. Ohtsubo, M. Okamura, S. O'Neale, Y. Oohata, K. Paech, J. Perl, A. Pfeiffer, M. G. Pia, F. Ranjard, A. Rybin, S. Sadilov, E. Di Salvo, G. Santin, T. Sasaki, N. Savvas, Y. Sawada, S. Scherer, S. Sei, V. Sirotenko, D. Smith, N. Starkov, H. Stoecker, J. Sulkimo, M. Takahata, S. Tanaka, E. Tcherniaev, E. Safai Tehrani, M. Tropeano, P. Truscott, H. Uno, L. Urban, P. Urban, M. Verderi, A. Walkden, W. Wander, H. Weber, J. P. Wellisch, T. Wenaus, D. C. Williams, D. Wright, T. Yamada, H. Yoshida, and D. Zschiesche". "g4–a simulation toolkit". *"Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment"*, "506"("3"):"250 – 303", "2003". "".

[12] J. Allison, K. Amako, J. Apostolakis, H. Araujo, P.A. Dubois, M. Asai, G. Barrand, R. Capra, S. Chauvie, R. Chytracek, G.A.P. Cirrone, G. Cooperman, G. Cosmo, G. Cuttone, G.G. Daquino, M. Donszelmann, M. Dressel, G. Folger, F. Foppiano, J. Generowicz, V. Grichine, S. Guatelli, P. Gumplinger, A. Heikkinen, I. Hrivnacova, A. Howard, S. Incerti, V. Ivanchenko, T. Johnson, F. Jones, T. Koi, R. Kokoulin, M. Kossov, H. Kurashige, V. Lara, S. Larsson, F. Lei, O. Link, F. Longo, M. Maire, A. Mantero, B. Mascialino, I. McLaren, P.M. Lorenzo, K. Minamimoto, K. Murakami, P. Nieminen, L. Pandola, S. Parlati, L. Peralta, J. Perl, A. Pfeiffer, M.G. Pia, A. Ribon, P. Rodrigues, G. Russo, S. Sadilov, G. Santin, T. Sasaki, D. Smith, N. Starkov, S. Tanaka, E. Tcherniaev, B. Tome, A. Trindade, P. Truscott, L. Urban, M. Verderi, A. Walkden, J.P. Wellisch, D.C. Williams, D. Wright, and H. Yoshida. Geant4 developments and applications. *Nuclear Science, IEEE Transactions on*, 53(1):270–278, Feb. 2006.

[13] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, August 2008.

Appendix A

IPP & ROOT

A.1 Process to integrate IPP zlib compression algorithm into ROOT

In order to draw comparison between a standard ROOT implementation and the one which includes IPP functions, two ROOT environments have been configured. The standard implementation uses the zlib 1.2.3 compression algorithm, whereas the latter uses ipp_zlib. The general build steps are as follows;

1. Setup Intel C++ 11.1 compiler environment
   (a) source `/afs/cern.ch/sw/IntelSoftware/linux/setup.sh`
   (b) source `/afs/cern.ch/sw/IntelSoftware/linux/x86_64/Compiler/11.1/038/bin/iccvars.shintel64`
2. Setup the IPP build environment
   (a) source `/afs/cern.ch/sw/IntelSoftware/linux/x86_64/Compiler/11.1/038/ipp/em64t/tools/env/ippvarsem64t.sh`
   (b) Alternatively the environment variables `IPPROOT` and `LD_LIBRARY_PATH` can manually be created.
      i. `IPPROOT=/afs/cern.ch/sw/IntelSoftware/linux/x86_64/Compiler/11.1/038/ipp/em64t/`
      ii. `LD_LIBRARY_PATH=$IPPROOT/sharedlib/`

If building the sample IPP source codes you would then build with './buildem64t.sh icc111' for icc 11.1, or './buildem64t.sh gcc4' for gcc 4.3.3. When building the ipp_zlib sample sources it was discovered that an additional library needs to be linked, which has not been included in the ippvarsem64t.sh script. After step 2a the following environment variable should also be created;

`LD_LIBRARY_PATH=/afs/cern.ch/sw/IntelSoftware/linux/x86_64/Compiler/11.1/038/lib/intel64:$LD_LIBRARY_PATH`

A.1.0.1 Replacement of standard zlib files with ipp_zlib files

The IPP based implementation of zlib used a combination of some of the original zlib source and header files, and some updated files. Included below is a list of the updated IPP files that need to replace the original zlib files.

Updated IPP header files

- deflate.h
- inflate.h

Updated IPP source files

- adler32.c
- crc32.c
- deflate.c
- inflate.c
- ipp_static.cpp

These files can be obtained from the zlib sample code shipped with IPP under the following location;
`/ipp-samples/data-compression/ipp_zlib/src/`
Should additional information be required regarding the changes to these files, the readme file can be consulted at;
`/ipp-samples/data-compression/ipp_zlib/readme.htm`
Within ROOT the source files for the compression part are located at `/root/core/zip/src/` and the header files at `/root/core/zip/inc`. The updated IPP files should override the ones existing in these two directories already.

A.1.0.2 Building ROOT with IPP zlib compression

After completing this file replacement, the ROOT framework can now be built.

1. First source the IPP variables;
   source `/afs/cern.ch/sw/IntelSoftware/linux/x86_64/Compiler/11.1/038/ipp/em64t/tools/env/ippvarsem64t.sh`
2. Source the compiler to be used;
   (a) For gcc the gcc-setup.sh script can be used;
   ```
   . ./gcc.sh 4.3.3
   ```
   (b) For icc, the Intel environment and compiler should first be configured following the instructions at
   `https://twiki.cern.ch/twiki/bin/view/Openlab/IntelTools`

3. Execute the configuration script with flags for the architecture and operating system configuration being used[1]. For example, the following configurations were used during this testing for gcc and icc running on x86-64 hardware respectively;

```
./configure linuxx8664gcc --enable-xrootd

./configure linuxx8664icc --enable-xrootd
```

4. Finally 'make' can be run to build ROOT[2].

5. After the build process, a 'bin' directory will have been created containing a 'thisroot.sh' script which must be sourced before running ROOT.

A.2 BKM for setup and compiling of ROOT when used with IPP

Having spent some time investigating the compression part of ROOT together with IPP for its inclusion within the framework, a number of problems have been encountered. This BKM highlights some of these issues and how they can be overcome.

Use of ICC with IPP?

When compiling source code which includes IPP functions, both gcc and icc compilers can be used. Throughout this investigation both compilers have been used; icc 11.1 and gcc 4.3.3. When using the sample codes, the Intel compiler icc 11.1 can be selected by specifying 'icc111' at the command line during the build process. Similarly, gcc 4.3.3 can be selected by issuing 'gcc4' at the command line.

No shared object library was found in the Waterfall procedure

If the error message "No shared object library was found in the Waterfall procedure" is received, it means that Linux is unable to determine the location of the Intel IPP shared object libraries. To solve this issue:

- Ensure that the Intel IPP directory is in the path.
- Before using the Intel IPP shared object libraries, add the path to the shared object libraries to the system variable LD_LIBRARY_PATH as described in "Using Intel IPP Shared Object Libraries (SO)" in Chapter 3 of [2].

[1] Should a different configuration be required, './configure –help' lists all supported architectures and operating systems.

[2] With the standard ROOT source it is possible to build the source in parallel with 'make -j8', although this has not been verified for the modified ROOT source including IPP zlib compression.

Which IPP header files to include?

The IPP functions are defined in a number of header files within the include folder. For example, ippdc.h has been used within the software discussed in this report. In order to ensure forward compatibility, individual header files for each domain should not be included, but instead ipp.h. This header file includes other IPP header files as required.

Linking IPP libraries

The shared libraries libipp*.so.6.1 (where * denotes the appropriate function domain) are dispatcher dynamic libraries[3], as shown in table A.1. At run time, they detect the processor and load the correct processor-specific shared libraries as detailed in table A.2. This allows code to be written that calls the Intel IPP functions without worrying about which processor the code will execute on - the appropriate version is automatically used.

These processor-specific libraries are named libipp*px.so.6.1, libipp*w7.so.6.1, libipp*t7.so.6.1, libipp*v8.so.6.1, and libipp*p8.so.6.1. For example, in the ia32/sharedlib directory, libippiv8.so.6.1 reflects the imaging processing libraries optimised for the Intel Core 2 Duo processors.

Include in the project soft links to the shared libraries instead of the shared libraries themselves. These soft links are named as the corresponding shared libraries without version indicator: libipp*-6.1.so, libipp*px-6.1.so, libipp*w7-6.1.so, libipp*t7-6.1.so, libipp*v8-6.1.so, and libipp*p8-6.1.so.

A.3 Intel ipp_gzip decompression bug report

-Overview-

Synopsis: ipp_gzip –decompress (IPP 6.1) code sample results in CRC errors and unsuccessful uncompressing of .tar.gz

Description:

Compiled the supplied ipp_gzip sample with both gcc 4.1.2 and icc 11.1 on Harpertown and Nehalem systems. The compression of .tar's of all sizes functions as expected (mirroring that of gzip). When decompressing the .tar.gz files created in the previously described step with ipp_gzip -d, a number of CRC errors are encountered, usually resulting in unsuccessful decompression.
-Routing information-

Version: Linux Intel IPP 6.1.0.047 samples

Regression version: Unknown, first tested with this latest release

Severity: High

[3]Dispatching refers to detection of the CPU and selecting the Intel IPP binary that corresponds to the hardware being used.

Table A.1: Types of Libraries of Intel IPP [2]

Library types	Description	Folder location	Example
Dynamic	Shared object libraries include both processor dispatchers and function implementations	ia32/sharedlib	libipps.so.6.1, libippst7.so.6.1
	Soft links to the shared object libraries	ia32/sharedlib	libipps.so libippst7.so
Static merged	Contain function implementations for all supported processor types:		
	libraries with position independent code (PIC)	ia32/lib	libippsmerged.a
	non-PIC)* libraries	ia32/lib/nonpic	libippsmerged.a
Threaded static merged	Contain threaded function implementations	ia32/lib	libippsmerged_t.a
Static emerged	Contain dispatchers for the merged libraries:		
	libraries with position independent code (PIC)	ia32/lib	libippsemerged.a
	non-PIC*) libraries	ia32/lib/nonpic	libippsemerged.a

*) non-PIC libraries are suitable for kernel-mode and device-driver use.

Table A.2: Identification of Codes Associates with Processor-Specific Libraries [2]

Abbreviation	Meaning
	IA-32 Intel® architecture
px	C-optimized for all IA-32 processors
w7	Optimized for processors with Intel® Streaming SIMD Extensions 2 (Intel SSE2)
t7	Optimized for processors with Intel® Streaming SIMD Extensions 3 (Intel SSE3)
v8	Optimized for processors with Intel® Supplemental Streaming SIMD Extensions 3 (Intel SSSE3)
p8	Optimized for processors with Intel® Streaming SIMD Extensions 4.1 (SSE4.1)
s8	Optimized for the Intel® Atom™ processor.

-Test case-

OS:

Harpertown machine: Custom build RHEL5 Linux 2.6.28.3-perfmon (x86_64)

Nehalem machine: Custom build RHEL5 Linux 2.6.18-128.1.1.e15 (x86-64)

Hardware:

Harpertown machine: Dual socket Intel Xeon E5450 @ 3.00GHz, 16GB physical memory

Nehalem machine: Dual socket Intel Xeon X5570 @ 2.93GHz, 24GB physical memory

Source code: `https://registrationcenter.intel.com/irc_nas/1454/l_ipp-samples_p_6.1.0.047.tgz`

Steps to reproduce:

1. Setup IPP build environment
2. Goto working dir: .../ipp-samples/data-compression/ipp_gzip/bin/linuxem64t_icc111
3. Compress a large tar (1 GB+): ./ipp_gzip fedCD.tar
4. Decompress the tar.gz: ./ipp_gzip -d fedCD.tar.gz

Expected result: ipp_gzip to decompress the tar.gz without errors and successfully, mirroring gzip functionality

Actual result: mixed array of errors, with mixed outcomes of successful uncompressing of .tar.gz (see 'ipp_gzip testing on Harpertown and Nehalem.txt' for testing results)

-Contact information- Marc D'Arcy mdarcy@cern.ch

Appendix B

MTGeant4

B.1 Configuration of the MTGeant4 prototype

To build the multi-threaded Geant4 prototype the following process should be followed;

1. Ensure the $HOME variable points to a known location and that it has sufficient space. If not, temporarily change the home directory e.g. 'export HOME=/....'

2. Edit the build script and set the build location and install paths as desired. The number of cores of the system should also be set here.

3. Execute the build script.

4. MTGeant4 should be built with gcc 4.3.3, which can be sourced with the included '. gcc-setup.sh 4.3.3'.

5. Open the build script for editing. At the end of the script are a list of components that should be installed in order. Each component should be uncommented in the following order to install each individually;

   (a) install_clhep

   (b) install_xercesc

   (c) install_geant4

   (d) install_parfullcmsmt

6. After installing the full CMS benchmark, the directory 'ParFullCMSmt' should be created in the mtg4 directory.

7. Move to this directory and open the script onerun.sh for editing. Ensure that XERCESCROOT and LD_LIBRARY_PATH point correctly to the install path selected for mtg4 in step 1. In this script the required benchmark can also be selected.

8. Run the benchmark with 'time ./onerun.sh #threads'.

To run MTG4 in a new process after install

1. Ensure that the HOME environment variable is set identical to that in the initial installation.

2. Source gcc 4.3.3 with the gcc-setup.sh script.

B.2 MTG4 Nehalem Xeon X5570 scaling testing with perfmon output

MTG4 Nehalem Xeon X5570 scaling testing with perfmon output

Dual-socket Intel Xeon X5570, SLC5, 8 cores total **gcc 4.3.3, 64-bit, 24 GB memory (opladev27)** **tcmalloc 1.3, 500 pi- 300GeV evts total**	**1**	**2**	**4**	**8**
Time [s] as output by Geant4	3296	1861	1048	613
perfmon-output				
UNHALTED_CORE_CYCLES	10,315,270,232,308	10,799,625,738,304	11,220,482,050,468	10,923,260,596,904
INSTRUCTIONS_RETIRED	9,701,714,362,048	9,719,376,817,423	10,177,442,279,681	9,981,755,717,118
BR_INST_EXEC:ANY	1,643,994,712,887	1,641,866,144,222	1,700,535,843,855	1,668,765,316,944
BR_MISP_EXEC:ANY	53,496,225,086	53,370,344,809	54,606,992,447	53,668,163,833
INST_RETIRED:X87	1,178,882,152	1,200,737,345	1,268,405,796	1,317,447,103
INST_RETIRED:MMX	0	0	0	0
RESOURCE_STALLS:ANY	2,276,238,512,163	2,473,705,582,012	2,758,865,092,740	2,827,779,374,658
FP_COMP_OPS_EXE:MMX:SSE_DOUBLE_PRECISION:S	1,575,366,970,218	1,586,961,523,213	1,700,465,553,887	1,682,060,359,137
LAST_LEVEL_CACHE_REFERENCES	50,751,412,949	54,862,008,449	56,556,806,451	56,353,328,966
LAST_LEVEL_CACHE_MISSES	198,371,677	1,404,742,204	1,782,851,673	2,083,906,954
MEM_INST_RETIRED:LOADS	3,589,203,769,579	3,590,008,034,175	3,733,673,893,035	3,662,895,569,352
MEM_INST_RETIRED:STORES	1,650,380,514,908	1,650,892,319,610	1,709,370,483,966	1,681,048,197,467
perfmon-output-simd1				
UNHALTED_CORE_CYCLES	10,339,108,319,085	11,045,279,436,747	10,936,918,271,294	10,967,990,165,379
INSTRUCTIONS_RETIRED	9,699,493,208,065	9,753,595,884,049	9,791,668,928,776	9,978,120,009,627
FP_COMP_OPS_EXE:SSE_DOUBLE_PRECISION	1,246,009,490,069	1,264,191,517,236	1,285,527,289,369	1,343,438,935,500
FP_COMP_OPS_EXE:SSE_SINGLE_PRECISION	92,331,654,149	92,301,249,046	92,088,870,771	92,009,356,186
FP_COMP_OPS_EXE:SSE_FP_PACKED	30,263,034,411	30,244,845,754	30,246,517,814	30,190,289,839
FP_COMP_OPS_EXE:SSE_FP_SCALAR	1,308,078,109,807	1,326,247,920,527	1,347,369,642,326	1,405,258,001,847
FP_COMP_OPS_EXE:SSE_FP	1,336,672,709,112	1,356,576,077,099	1,379,311,792,752	1,433,535,811,702
FP_COMP_OPS_EXE:SSE2_INTEGER	36,371,729	56,343,985	65,013,119	68,308,008
FP_COMP_OPS_EXE:MMX	0	0	0	0
FP_COMP_OPS_EXE:X87	269,412,440,329	270,974,544,992	271,519,561,986	276,797,525,264

perfmon-output-simd2

UNHALTED_CORE_CYCLES	10,362,643,320,857	11,092,081,861,550	10,811,328,472,557	10,939,119,803,666
INSTRUCTIONS_RETIRED	9,695,592,279,168	9,742,085,390,002	9,714,985,348,299	9,942,722,010,126
SSEX_UOPS_RETIRED:PACKED_DOUBLE	39,201,571	40,754,190	42,291,803	46,559,098
SSEX_UOPS_RETIRED:PACKED_SINGLE	4,416,563	4,756,743	5,097,929	5,848,791
SSEX_UOPS_RETIRED:SCALAR_DOUBLE	2,357,232,891,942	2,384,624,057,313	2,397,727,062,952	2,485,604,645,665
SSEX_UOPS_RETIRED:SCALAR_SINGLE	248,834,934,654	248,565,998,154	246,145,152,314	247,347,012,645
SSEX_UOPS_RETIRED:VECTOR_INTEGER	14,176,659,139	14,671,608,433	15,045,412,620	16,210,914,212
SSEX_UOPS_RETIRED:PACKED_DOUBLE:PACKED_SINC	2,606,571,678,217	2,633,699,858,938	2,644,373,080,563	2,733,516,460,205
FP_COMP_OPS_EXE:SSE_FP	1,336,511,731,950	1,355,378,118,644	1,369,542,959,613	1,431,922,332,486
FP_COMP_OPS_EXE:SSE2_INTEGER	58,621,921	71,884,547	69,386,706	56,957,873
FP_COMP_OPS_EXE:MMX	0	0	0	0
FP_COMP_OPS_EXE:X87	269,694,230,947	270,612,273,162	270,174,789,805	276,066,509,918

perfmon-output-stalls

UNHALTED_CORE_CYCLES	10,453,676,685,546	10,629,690,245,761	10,917,389,327,810	10,921,950,376,361
INSTRUCTIONS_RETIRED	9,700,072,406,288	9,726,368,886,352	9,764,081,522,816	9,980,163,625,141
RESOURCE_STALLS:LOAD	232,398,398,286	299,949,197,511	299,257,375,151	339,177,545,291
RESOURCE_STALLS:STORE	101,939,051,147	146,134,420,183	191,971,366,243	246,655,760,082
RESOURCE_STALLS:RS_FULL	1,842,035,624,263	1,879,731,950,183	1,939,560,314,920	2,097,212,836,749
RESOURCE_STALLS:ROB_FULL	123,091,058,200	174,953,208,420	178,850,512,498	187,807,055,095
RESOURCE_STALLS:FPCW	0	0	0	0
RESOURCE_STALLS:MXCSR	0	0	0	0
RESOURCE_STALLS:OTHER	0	0	0	0
RESOURCE_STALLS:ANY	2,273,537,842,017	2,464,929,777,778	2,591,861,813,113	2,835,172,546,253
FP_COMP_OPS_EXE:SSE_FP	1,338,848,000,433	1,352,279,132,076	1,374,156,533,298	1,440,059,925,931
FP_COMP_OPS_EXE:SSE2_INTEGER	64,480,511	40,815,413	92,376,209	46,603,807
FP_COMP_OPS_EXE:MMX	0	0	0	0
FP_COMP_OPS_EXE:X87	269,404,306,095	270,503,209,951	271,233,018,152	278,028,343,691