Universidad Nacional de La Plata
Facultad de Informática

Tesis presentada para obtener el grado de
Magíster en Cómputo de Altas Prestaciones

Infraestructura para el Análisis de Rendimiento

Alumno: Andrés More - amore@hal.famaf.unc.edu.ar
Director: Dr Fernando G. Tinetti - fernando@lidi.info.unlp.edu.ar

Junio de 2016

Resumen

En el área del cómputo de altas prestaciones las aplicaciones son construidas por especialistas del dominio del problema, no siempre expertos en optimización de rendimiento. Se identifica entonces la necesidad de un método sistemático y automático para dar soporte al análisis de rendimiento.

Esta tesis revisa la construcción de una infraestructura que simplifica el análisis de rendimiento; incluyendo su aplicación mediante casos de estudio como etapa de validación. El objetivo es facilitar la tarea evitando la tediosa recolección de datos relevantes de modo manual, permitiendo más tiempo de experimentación para la optimización propiamente dicha.

La infraestructura brinda información de contexto sobre un programa y el sistema donde se ejecuta. Esto incluye su comportamiento al incrementar las unidades de cómputo o el tamaño del problema, el perfil de ejecución, sus cuellos de botella y la utilización de los recursos disponibles.

En particular, esta tesis contribuye entonces con un generador automático de reportes de rendimiento para aplicaciones de cómputo de altas prestaciones utilizando tecnología *OpenMP* sobre sistemas *GNU/Linux*.

Índice general

1. Introducción **6**
- 1.1. Motivación . . . 6
- 1.2. Objetivos . . . 7
- 1.3. Metodología . . . 7
- 1.4. Contribuciones . . . 7
- 1.5. Estructura . . . 8

2. Trabajo Relacionado **9**
- 2.1. Análisis de Rendimiento . . . 9
  - 2.1.1. Rendimiento . . . 9
  - 2.1.2. Paralelismo . . . 9
  - 2.1.3. Métricas . . . 12
  - 2.1.4. Técnicas de Análisis . . . 12
- 2.2. Herramientas . . . 14
  - 2.2.1. Pruebas de Rendimiento . . . 14
  - 2.2.2. Utilización de las Herramientas . . . 21
  - 2.2.3. Tiempo de Ejecución . . . 21
  - 2.2.4. Perfil de Ejecución Funcional . . . 23
  - 2.2.5. Perfil de Ejecución Asistido por *Hardware* . . . 24
  - 2.2.6. Reporte de Vectorización . . . 26
- 2.3. Trabajo Previo . . . 26

3. Descripción del Problema **27**
- 3.1. Análisis de Rendimiento . . . 27
  - 3.1.1. Problemas Usuales . . . 27
  - 3.1.2. Métodos de Optimización . . . 28
- 3.2. Infraestructura de Soporte . . . 29

4. Propuesta de Solución **31**
- 4.1. Procedimiento . . . 31
- 4.2. Infraestructura . . . 34
- 4.3. Teoría de Operación . . . 35
  - 4.3.1. Arquitectura . . . 35
  - 4.3.2. Funcionamiento Interno . . . 35
- 4.4. Diseño de Alto Nivel . . . 36
- 4.5. Diseño de Bajo Nivel . . . 37
  - 4.5.1. Implementación . . . 38
  - 4.5.2. Archivos . . . 38

4.5.3. Configuración 39
4.6. Reporte Generado 40
4.6.1. Consideraciones Generales 40
4.6.2. Consideraciones Particulares 40

5. Casos de Aplicación **43**
5.1. Sistema de Prueba 43
5.2. Código de Prueba 44
5.2.1. Multiplicación de Matrices 44
5.2.2. Transmisión de Calor en 2 Dimensiones 45
5.2.3. Conjunto de *Mandelbrot* 48

6. Conclusiones y Trabajo Futuro **51**
6.1. Conclusiones 51
6.2. Trabajo Futuro 52

A. Salida de Ejemplo **57**

B. Reportes de Ejemplo **60**
B.1. Multiplicación de Matrices 60
B.2. Propagación de Calor en 2 Dimensiones 71
B.3. Conjunto de Mandelbrot 86

C. Contribuciones **97**
C.1. Reportes Técnicos 97
C.1.1. Estudio de Multiplicación de Matrices 97
C.1.2. Comparación de Implementaciones de una Operación BLAS 105
C.2. Artículos en Conferencias 114
C.2.1. *Optimizing Latency in Beowulf Clusters* 114
C.2.2. *Lessons Learned from Contrasting BLAS Kernel Implementations* 129
C.2.3. *Hotspot: a Framework to Support Performance Optimization on Multiprocessors* 138
C.3. Libros 145
C.3.1. Contribuciones en el Libro *Programming Intel Xeon Phi* . 145
C.3.2. Reseña del Libro *Intel Xeon Phi Coprocessor High Performance Programming* 145

Índice de figuras

2.1. Optimización Iterativa 13
2.2. Desviación en una distribución normal 13
2.3. Rendimiento Agregado del Top500 [Top500]) 16
2.4. Localidad temporal versus espacial en resultados de HPCC . . . 19
2.5. Diagrama Kiviat [Top500] 20

4.1. Procedimiento de Análisis 31
4.2. Diagrama de Secuencia 35
4.3. Diseño de Alto Nivel 36
4.4. Diseño de Bajo Nivel 37

Índice de cuadros

2.1. Mejora Máxima según *Amdahl* 10
2.2. Mejora Máxima según *Gustafson* 11
2.3. Pruebas de Rendimiento 15
2.4. Operaciones del Benchmark STREAM 15
2.5. Aplicación Gradual de Herramientas 21

Índice de Listados

1. Ejecución del Programa 22
2. Tiempo de Ejecución 22
3. Compilación con Información de Depuración 23
4. Perfil de Rendimiento 23
5. Gráficos de Llamadas 24
6. Estadísticas de Contadores 25
7. Perfil de Rendimiento 25
8. Código Anotado 25
9. Información de Vectorización 26
10. Vectorización de Código Recursivo 26
11. Instalación de HPCC 32
12. Compilación del Binario 32
13. Estabilidad de Resultados 33
14. Escalamiento de Problema 33
15. Escalamiento de Cómputo 33
16. Generación de Perfil de Rendimiento 34
17. Generación de Perfil de Sistema 34
18. Información de Vectorización 34
19. Ayuda de `hotspot` 39
20. Configuración de `hotspot` 39
21. Código de Multiplicación de Matrices 44
22. Caso de Prueba de Multiplicación de Matrices 44
23. Código de Transmisión de Calor en 2 Dimensiones 47
24. Caso de Prueba de Distribución de Calor en 2 Dimensiones . . . 48
25. Código de Conjunto de Mandelbrot 49
26. Caso de Prueba de Conjunto de Mandelbrot 49
27. Salida de Ejemplo 57

Capítulo 1

Introducción

Este capítulo introduce el tema bajo estudio, definiendo los objetivos principales, las contribuciones logradas durante la investigación, y concluye detallando la estructura del resto del documento.

1.1. Motivación

En el área de cómputo de altas prestaciones (o *HPC*, por las siglas en inglés de *High Performance Computing*) los desarrolladores son los mismos especialistas del dominio del problema a resolver. Las rutinas más demandantes de cálculo son en su mayoría científicas y su gran complejidad y grado de especialización hace que los mismos investigadores del área de aplicación sea quienes las implementan. Estas cuestiones resultan en una inversión no trivial en el análisis de rendimiento que impacta directamente en la productividad de los grupos de investigación y desarrollo.

Con mayor impacto que en otras áreas de la computación, el código optimizado correctamente puede ejecutarse órdenes de magnitud mejor que una implementación directa [1]. Esto implica que invertir en optimización de un programa puede tener una ganancia interesante a largo plazo si las simulaciones a realizar son demasiadas.

Los investigadores suelen utilizar programación en paralelo para obtener una mejor utilización de la capacidad de cómputo disponible. [2]. Esto aumenta por lo tanto la complejidad de implementación, depuración y optimización con la que el experto en el dominio necesita trabajar en el día a día [3] [4].

Frecuentemente el proceso de optimización termina siendo hecho de modo *ad-hoc*, sin conocimiento pleno de las herramientas disponibles y sus capacidades, y sin la utilización de información cuantitativa para dirigir los esfuerzos de optimización. Es incluso frecuente la implementación directa de algoritmos en lugar de la utilización de librerías ya disponibles, optimizadas profundamente y con correctitud garantizada [5].

1.2. Objetivos

La propuesta principal consiste en el desarrollo de una infraestructura de soporte para el análisis de aplicaciones de cómputo de altas prestaciones.

El problema a resolver es simplificar la compleja tarea de realizar un análisis de rendimiento sobre programas de simulación y cálculo científico. El análisis implica la ejecución repetitiva de varios casos de prueba bajo diferentes condiciones, además de la aplicación de múltiples herramientas para analizar el comportamiento y la utilización de los recursos disponibles.

La infraestructura desarrollada implementa un procedimiento de análisis de rendimiento ejecutando pruebas de referencia, herramientas de perfil de rendimiento y graficación de resultados. La infraestructura genera como etapa final un informe detallado que da soporte a la tarea de optimización con información cuantitativa. El reporte final incluye datos estadísticos de la aplicación y el sistema donde se ejecuta, además de gráficos de desviación de resultados, escalamiento de problema y cómputo, e identificación de cuellos de botella.

1.3. Metodología

En base al problema y a los objetivos establecidos previamente, la metodología adoptada es la siguiente:

1. Analizar el estado del arte del análisis de rendimiento y las herramientas utilizadas para ello.
2. Formular la solución específica necesaria para simplificar la tarea.
3. Identificar que tipos de gráficos y tablas pueden resumir la información obtenida de modo de facilitar su utilización.
4. Dada una propuesta de procedimiento sistemático de análisis de rendimiento, automatizarlo y analizar potenciales mejoras de utilidad.
5. Aplicar la infraestructura sobre núcleos de cómputo conocidos para poder focalizar los esfuerzos en la mejora del reporte.
6. Documentación de la experiencia.

1.4. Contribuciones

La siguiente lista enumera las diferentes publicaciones realizadas durante el cursado del magíster y el desarrollo de la tesis.

1. Estudio de Multiplicación de Matrices. Reporte Técnico. Realizado como parte del curso *Programación en Clusters* dictado por el Dr *Fernando Tinetti* [6].
2. Artículo *Optimizing Latency in Beowulf Clusters*. HPC Latam 2012 [7].
3. Comparación de Implementaciones de una Operación BLAS. Reporte técnico realizado como parte del curso *Programación GPU de Propósito General* dictado por la Dra *Margarita Amor*.
4. Sección *Intel Cluster Ready* e *Intel Cluster Checker* en el libro *Programming Intel Xeon Phi*. Intel Press. 2013 [8].

5. Reseña del Libro *Intel Xeon Phi Coprocessor High Performance Programming* - JCS&T Vol. 13 No. 2.
6. Artículo *Lessons Learned from Contrasting BLAS Kernel Implementations* - XIII Workshop Procesamiento Distribuido y Paralelo (WPDP), 2013 [9].
7. Artículo *Hotspot: a Framework to Support Performance Optimization on Multiprocessors* - PDPTA'15, 2015 International Conference on Parallel and Distributed Processing Techniques and Applications [10].

Las publicaciones realizadas pueden encontrarse en el apéndice C.

1.5. Estructura

El resto del documento se estructura de la siguiente manera:

- Capítulo 2: revisa el estado del arte de los temas incluidos dentro del análisis de rendimiento: paralelismo, leyes fundamentales, métricas, técnicas de análisis, herramientas de soporte y pruebas de rendimiento.
- Capítulo 3: describe la problemática a resolver como parte de esta tesis. Los problemas usuales del propio análisis, métodos de optimización a dar soporte, y una discusión de cómo debería ser una infraestructura de soporte.
- Capítulo 4: muestra la propuesta de solución. El procedimiento general propuesto, las herramientas a utilizar, y la arquitectura y diseño de la solución implementada junto con una descripción de las diferentes secciones generadas por la herramienta en el reporte final.
- Capítulo 5: aplica la solución a casos de estudio. Se incluyen los resultados de tres ejemplos ejercitando núcleos de cómputo conocidos.
- Capítulo 6: concluye reflejando aportes y proponiendo trabajo futuro.
- Apéndice A: incluye los resultados de la infraestructura.
- Apéndice B: muestra los reportes completos de los casos de aplicación.
- Apéndice C: contiene las publicaciones realizadas durante el transcurso del trabajo de tesis.

Capítulo 2

Trabajo Relacionado

Este capítulo revisa el estado del arte de los temas relevantes a esta tesis. Nos basaremos en una definición más teoría básica de análisis de rendimiento para dar contexto a los capítulos siguientes donde se aplican estos conceptos utilizando un procedimiento soportado por herramientas.

2.1. Análisis de Rendimiento

Esta sección introduce el concepto de rendimiento y teoría básica sobre su análisis; además revisa las herramientas disponibles para ello.

2.1.1. Rendimiento

El rendimiento se caracteriza por la cantidad de trabajo de cómputo que se logra en comparación con la cantidad de tiempo y los recursos ocupados. El rendimiento debe ser evaluado entonces de forma cuantificable, utilizando alguna métrica en particular de modo de poder comparar relativamente dos sistemas o el comportamiento de un mismo sistema bajo una configuración distinta.

2.1.2. Paralelismo

Una vez obtenida una implementación eficiente, la única alternativa para mejorar el rendimiento es explotar el paralelismo que ofrecen los sistemas de cómputo. Este paralelismo se puede explotar a diferentes niveles, desde instrucciones especiales que ejecutan sobre varios datos a la vez (vectorización), hasta la utilización de múltiples sistemas para distribuir el trabajo.

El cálculo de las mejoras posibles de rendimiento, cómo priorizarlas y la estimación de su límite máximo es una tarea compleja. Para ello existen algunas leyes fundamentales utilizadas durante el análisis de rendimiento.

Ley de *Amdahl*

La ley de *Amdahl* [11] dimensiona la mejora que puede obtenerse en un sistema de acuerdo a las mejoras logradas en sus componentes. Nos ayuda a establecer

un límite máximo de mejora y a estimar cuales pueden ser los resultados de una optimización.

La mejora de un programa utilizando cómputo paralelo está limitada por el tiempo necesario para completar su fracción serial o secuencial. En la mayoría de los casos, el paralelismo sólo impacta notoriamente cuando es utilizado en un pequeño número de procesadores, o cuando se aplica a problemas altamente escalables (denominados *Embarrassingly Parallel Problems* en inglés). Una vez paralelizado un programa, los esfuerzos suelen ser enfocados en cómo minimizar la parte secuencial, algunas veces haciendo más trabajo redundante pero en forma paralela.

Suponiendo que una aplicación requiere de un trabajo serial más un trabajo paralelizable, la ley de *Amdahl* calcula la ganancia (S) mediante la Ecuación 2.1. Donde P es el porcentaje de trabajo hecho en paralelo, $(1-P)$ es entonces el trabajo en serie o secuencial, y N la cantidad de unidades de cómputo a utilizar.

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{2.1}$$

Esta ley establece que incluso teniendo infinitas unidades de cómputo la ganancia está limitada. La Tabla 2.1 muestra que no importa la cantidad de unidades de procesamiento que sean utilizadas, siempre existe un límite en la práctica.

Tabla 2.1: Mejora Máxima según *Amdahl*

		Porcentaje de Paralelismo					
		0.1	0.3	0.5	0.8	0.9	0.95
Número de Procesadores	1	1.00	1.00	1.00	1.00	1.00	1.00
	2	1.05	1.14	1.33	1.60	1.82	1.90
	4	1.08	1.23	1.60	2.29	3.08	3.48
	8	1.10	1.28	1.78	2.91	4.71	5.93
	16	1.10	1.31	1.88	3.37	6.40	9.14
	32	1.11	1.32	1.94	3.66	7.80	12.55
	64	1.11	1.33	1.97	3.82	8.77	15.42
	128	1.11	1.33	1.98	3.91	9.34	17.41
	256	1.11	1.33	1.99	3.95	9.66	18.62
	512	1.11	1.33	2.00	3.98	9.83	19.28
	1024	1.11	1.33	2.00	3.99	9.91	19.64
	2048	1.11	1.33	2.00	3.99	9.96	19.82
	4096	1.11	1.33	2.00	4.00	9.98	19.91
	8192	1.11	1.33	2.00	4.00	9.99	19.95
	16384	1.11	1.33	2.00	4.00	9.99	19.98
	32768	1.11	1.33	2.00	4.00	10.00	19.99
	65536	1.11	1.33	2.00	4.00	10.00	19.99

Por ejemplo, en el caso de tener sólo un 10 % de paralelismo en una aplicación, la mejora nunca va a superar 1,1 veces la original. En el caso de tener un 95 % de paralelismo, la mejora no puede ser mayor a 20 veces la original.

En el caso de conocer los tiempos de ejecución para distinto número de procesadores, la porción serial/paralelo puede ser aproximada. En procesadores modernos, la mejora es afectada por la sincronización interna de las unidades de cómputo [12].

Ley de *Gustafson*

Desde un punto de vista más general, la ley de *Gustafson* [13] (Ecuación 2.2) refleja que en la práctica el tamaño del problema es escalado junto con el número de procesadores. En comparación, la ley anterior no ajusta el tamaño o resolución del problema cuando se incrementa la potencia de cálculo, es decir asume un tamaño de problema fijo.

$$Speedup(P) = P - \alpha \times (P - 1) \tag{2.2}$$

donde P es el número de unidades de cómputo y α el porcentaje de trabajo paralelizable.

Al aplicar esta ley obtenemos que un problema con datos grandes o repetitivos en cantidades grandes puede ser computado en paralelo muy eficientemente. Nos es útil para determinar el tamaño de problema a utilizar cuando los recursos de cómputo son incrementados. En el mismo tiempo de ejecución, el programa resuelve entonces problemas más grandes.

Tabla 2.2: Mejora Máxima según *Gustafson*

		Porcentaje de Paralelismo					
		0.1	0.25	0.5	0.75	0.9	0.95
Número de Procesadores	1	1	1	1	1	1	1
	2	1	1	2	2	2	2
	4	1	2	3	3	4	4
	8	2	3	5	6	7	8
	16	3	5	9	12	15	15
	32	4	9	17	24	29	30
	64	7	17	33	48	58	61
	128	14	33	65	96	115	122
	256	27	65	129	192	231	243
	512	52	129	257	384	461	486
	1024	103	257	513	768	922	973
	2048	206	513	1025	1536	1843	1946
	4096	411	1025	2049	3072	3687	3891
	8192	820	2049	4097	6144	7373	7782
	16384	1639	4097	8193	12288	14746	15565
	32768	3278	8193	16385	24576	29491	31130
	65536	6555	16385	32769	49152	58983	62259

Similarmente al cuadro anterior, podemos deducir de la Tabla 2.2 que en el caso de un programa con sólo 10 % de paralelismo, al incrementar los recursos 64 veces sólo podemos incrementar el tamaño del problema 7 veces. En el otro extremo, nos estima un incremento de 61 veces en el caso de tener 95 % de paralelismo.

Métrica de *Karp-Flatt*

Esta métrica es utilizada para medir el grado de paralelismo de una aplicación [14]. Su valor nos permite rápidamente dimensionar la mejora posible al aplicar un alto nivel de paralelismo.

Dado un cómputo paralelo con una mejora de rendimiento ψ en P procesadores, donde $P > 1$. La fracción serial *Karp-Flatt* representada con e y calculada según la Ecuación 2.3 es determinada experimentalmente, mientras menor sea e mayor se supone el nivel de paralelismo posible.

$$e = \frac{\frac{1}{\psi} - \frac{1}{P}}{1 - \frac{1}{P}} \tag{2.3}$$

Para un problema de tamaño fijo, la eficiencia típicamente disminuye cuando el número de procesadores aumenta. Se puede entonces proceder a determinar si esta disminución es debida a un paralelismo limitado, a un algoritmo no optimizado o un problema de arquitectura del sistema.

2.1.3. Métricas

Algunos ejemplos de métricas de rendimiento son:

1. El ancho de banda y la latencia mínima de un canal de comunicación, una jerarquía de memorias o de una unidad de almacenamiento.
2. La cantidad de instrucciones, operaciones, datos o trabajo procesado por cierta unidad de tiempo.
3. El rendimiento asociado al costo del equipamiento, incluyendo mantenimiento periódico, personal dedicado y gastos propios del uso cotidiano.
4. El rendimiento por unidad de energía consumida (electricidad).

Un método de medición de rendimiento indirecto consiste en medir el uso de los recursos del sistema mientras se ejercita el mismo con un trabajo dado. Por ejemplo: el nivel de carga de trabajo en el sistema, la cantidad de operaciones realizadas por el sistema operativo o la unidad de procesamiento, la utilización de memoria o archivos temporales e incluso el ancho de banda de red utilizado durante la comunicación.

2.1.4. Técnicas de Análisis

El procedimiento de mejora general usualmente consiste en ciclos iterativos de medir, localizar, optimizar y comparar (Figura 2.1). Es muy importante mantener la disciplina en realizar un cambio a la vez ya que esto asegura resultados reproducibles y convergentes, sin efectos no deseados.

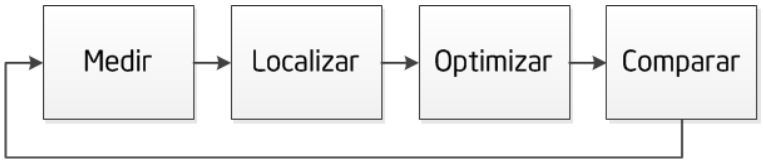

Figura 2.1: Optimización Iterativa

A la hora de tomar decisiones, éstas deben estar basadas en datos concretos, ya que en caso contrario se podría estar trabajando sin llegar a obtener un rédito adecuado.

En el caso de tener problemas de desviación en los resultados medidos, es aconsejable obtener un gran número de muestras y utilizar un valor promedio para asegurarse de evitar errores de medición tanto como sea posible. También es preferible aumentar el tamaño del problema a resolver, o la definición de los resultados, para ejercitar por más tiempo y tener así un resultado más estable [15].

Suponiendo una distribución normal de resultados, se suele controlar que haya menos de 3 desviaciones estándar de diferencia como prueba de normalidad [16]. Se busca que la mayoría de los resultados queden cerca de su promedio, como muestra la Figura 2.2.

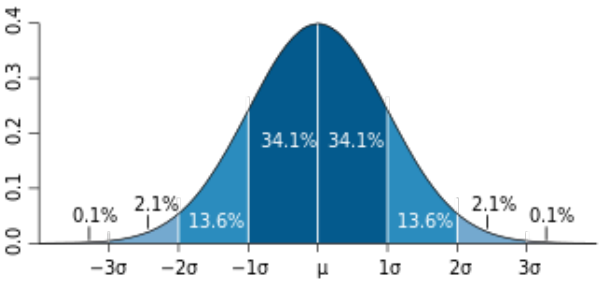

Figura 2.2: Desviación en una distribución normal

Los resultados deben también ser correctamente guardados para evitar problemas de datos, tanto numéricos como de utilización de recursos. Idealmente el caso de prueba debería validar la correctitud de la respuesta. Si la configuración del sistema bajo prueba es dinámica entonces la reproducción de los mismos es no trivial.

En el caso de no tener una configuración de sistema estable en el tiempo, es recomendable siempre ejecutar una versión optimizada contra una versión de referencia a continuación. Deben compararse primero los resultados númericos para asegurar la correctitud de los algoritmos y luego comparar la utilización de recursos para determinar si mejoró el rendimiento.

Para comparar es recomendable utilizar la media geométrica según la Ecuación 2.4 en lugar de la aritmética [17], ya que permite dimensionar la tendencia central de un valor típico en un conjunto de números. Esto permite reducir el impacto de ruido introducido por una ejecución problemática.

$$G = \sqrt[n]{x_1 \dots x_n} \tag{2.4}$$

La raíz n-ésima de un número (para un n posiblemente muy grande), es una operación ineficiente ya que se implementa con métodos numéricos de aproximación siguiendo el método de *Newton* [18]. En cambio se suele tomar el anti-logaritmo del promedio de los logaritmos de los valores siguiendo la ecuación 2.5.

$$G = 10^{(log_{10}(x_1)+\dots+log_{10}(x_n))/n} \tag{2.5}$$

2.2. Herramientas

Actualmente existen numerosas y diversas herramientas para el análisis de rendimiento [19]. Estas funcionan a diferentes niveles de abstracción: desde contadores de eventos a nivel de *hardware*, pasando por monitores de recursos dentro del núcleo del sistema operativo, instrumentación de código, y hasta la simple utilización del tiempo de ejecución de una aplicación o la comparación contra un trabajo similar de referencia.

2.2.1. Pruebas de Rendimiento

Para medir el rendimiento se utilizan pruebas de referencia (en inglés, *benchmarks*); éstas pueden ser aplicaciones sintéticas construidas específicamente, o bien el núcleo de cómputo de aplicaciones del mundo real computando un problema prefijado. Al tener valores de referencia se pueden caracterizar los sistemas de modo de predecir el rendimiento de una aplicación. Los valores a los que se llegan con un *benchmark* suelen ser más prácticos y comparables que los teóricos de acuerdo a condiciones ideales de uso de recursos. También es posible garantizar que el sistema sigue en un mismo estado con el correr del tiempo y después de cambios de configuraciones en *hardware* o *software*.

Las características deseables en un *benchmark* son portabilidad, simplicidad, estabilidad y reproducción de resultados. Esto permite que sean utilizadas para realizar mediciones cuantitativas y así realizar comparaciones de optimizaciones o entre sistemas de cómputo diferentes. También se pide que el tiempo de ejecución sea razonable y que el tamaño del problema sea ajustable para poder mantener su utilidad con el paso del tiempo y el avance de las tecnologías. La estabilidad de resultados sirve además para evitar errores de interferencia en los sistemas de cómputo durante la ejecución de pruebas.

A continuación se introducen algunos de los *benchmarks* más utilizados para cómputo de altas prestaciones (listados en la tabla 2.3), y posteriormente algunos detalles específicos e instancias de sus datos de salida para ser utilizados a manera de ejemplo.

Tabla 2.3: Pruebas de Rendimiento

Nombre	**Componente**	**Descripción**
STREAM	Memoria	Ancho de banda sostenido
Linpack	Procesador	Operaciones de punto flotante
IMB Ping Pong	Red	Latencia/Ancho de banda de red
HPCC	Sistema	Múltiples componentes

Los *benchmarks* pueden ser utilizados para diferentes propósitos. Primero, los valores reportados son usados como referencia para contrastar rendimiento. Segundo, su desviación demuestra que algo ha cambiado en el sistema (por lo tanto su no desviación indica que el sistema sigue estable). Por último, un *benchmark* sintético implementando el cómputo que uno quiere realizar muestra el rendimiento máximo posible a obtener en la práctica.

STREAM

STREAM [20] es un *benchmark* sintético que mide el ancho de banda de memoria sostenido en MB/s y el rendimiento de computación relativa de algunos vectores simples de cálculo. Se utiliza para dimensionar el ancho de banda de acceso de escritura o lectura a la jerarquía de memoria principal del sistema bajo análisis. Dentro de una misma ejecución de este *benchmark* se ejercitan diferentes operaciones en memoria, listadas en la tabla 2.4.

Tabla 2.4: Operaciones del Benchmark STREAM

Función	**Operación**	**Descripción**
`copy`	$\forall i\ b_i = a_i$	Copia simple
`scale`	$\forall i\ b_i = c \times a_i$	Multiplicación escalar
`add`	$\forall i\ c_i = b_i + a_i$	Suma directa
`triad`	$\forall i\ c_i = b_i + c \times a_i$	Suma y multiplicación escalar

La salida en pantalla muestra entonces los diferentes tiempos conseguidos y la cantidad de información transferida por unidad de tiempo. Como último paso, el programa valida también la solución computada.

```
STREAM version $Revision: 1.2 $
-------------------------------------------------------------
This system uses 8 bytes per DOUBLE PRECISION word.
-------------------------------------------------------------
Array size = 10000000, Offset = 0
Total memory required = 228.9 MB.
Each test is run 10 times, but only the *best* time is used.
-------------------------------------------------------------
Function     Rate (MB/s)   Avg time     Min time     Max time
Copy:        4764.1905       0.0337       0.0336       0.0340
Scale:       4760.2029       0.0338       0.0336       0.0340
Add:         4993.8631       0.0488       0.0481       0.0503
Triad:       5051.5778       0.0488       0.0475       0.0500
-------------------------------------------------------------
Solution Validates
```

La correcta utilización de la jerarquía de la memoria de un sistema de cómputo es una tarea no trivial [21].

Linpack

Linpack [22] es un conjunto de subrutinas *FORTRAN* que resuelven problemas de álgebra lineal como ecuaciones lineales y multiplicación de matrices. High Performance Linpack (HPL) [23] es una versión portable del *benchmark* que incluye el paquete Linpack pero modificado para sistemas de memoria distribuida.

Este *benchmark* es utilizado mundialmente para la comparación de la velocidad de las supercomputadoras en el ranking TOP500 [1]. Un gráfico del TOP500 de los últimos años (Figura 2.3) demuestra claramente la tendencia en crecimiento de rendimiento; también la relación entre el primero, el último y la suma de todos los sistemas en la lista.

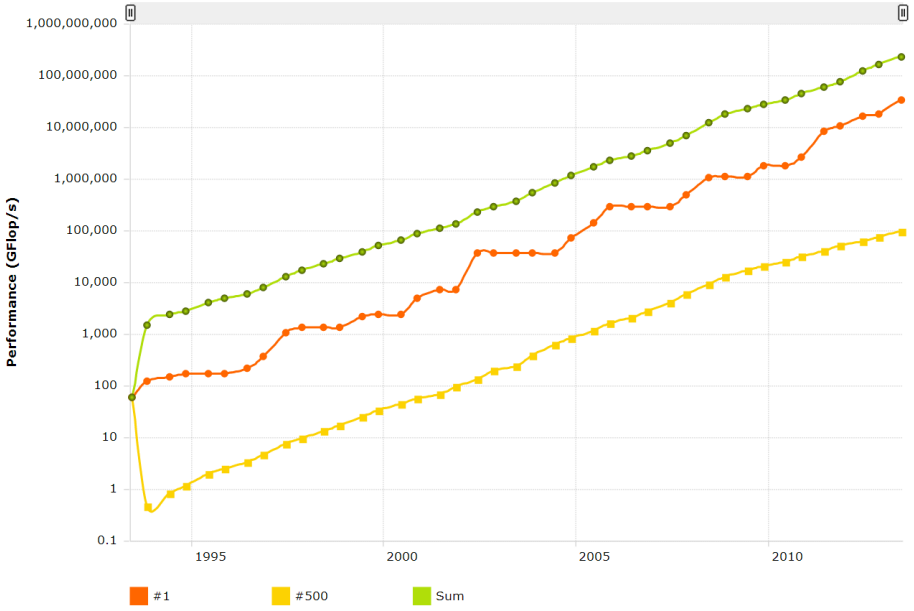

Figura 2.3: Rendimiento Agregado del Top500 [Top500])

Este *benchmark* requiere conocimiento avanzado para una correcta configuración, por ejemplo el tamaño de bloque que se va a utilizar para la distribución de trabajo debe estar directamente relacionado con el tamaño del *cache* de memoria del procesador.

La salida en pantalla resume entonces los datos de entrada y los resultados conseguidos. Como último paso el programa valida que los resultados sean correctos.

```
================================================================================
HPLinpack 2.0 - High-Performance Linpack benchmark - Sep 10, 2008
```

[1] http://www.top500.org/

```
Written by A. Petitet and R. Clint Whaley
================================================================
The following parameter values will be used:
N      :   28888
NB     :     168
PMAP   : Row-major process mapping
P      :       4
Q      :       4
PFACT  :   Right
NBMIN  :       4
NDIV   :       2
RFACT  :   Crout
BCAST  :  1ringM
DEPTH  :       0
SWAP   : Mix (threshold = 64)
L1     : transposed form
U      : transposed form
EQUIL  : yes
ALIGN  : 8 double precision words
----------------------------------------------------------------
- The matrix A is randomly generated for each test.
- The relative machine precision (eps) is taken to be 1.110223e-16
- Computational tests pass if scaled residuals are less than 16.0

Column=000168 Fraction=0.005 Mflops=133122.97
...
Column=025872 Fraction=0.895 Mflops=98107.60
================================================================================
T/V                N   NB   P    Q             Time             Gflops
WR01C2R4       28888  168   4    4           165.83          9.693e+01
--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N) = 0.0043035 .. PASSED
================================================================================
Finished      1 tests with the following results:
1 tests completed and passed residual checks,
0 tests completed and failed residual checks,
0 tests skipped because of illegal input values.
```

Existe cierta controversia de que no es una buena forma de ejercitar un sistema de cómputo distribuido ya que no implica uso significativo de la red, sólo procesamiento intensivo de aritmética de punto flotante sobre la jerarquía local de memoria.

Intel MPI Benchmarks

Es un conjunto de *benchmarks* cuyo objetivo es ejercitar las funciones más importantes del estándar para librerías de paso de mensajes (MPI, por la sigla de *Message Passing Interface* en inglés) [24]. El más conocido es el popular ping-pong, el cual ejercita la transmisión de mensajes ida y vuelta entre dos nodos de cómputo con diferentes tamaños de mensajes [7].

Para obtener el máximo ancho de banda disponible, se ejercita la comunicación

a través de mensajes con datos grandes. Para obtener la mínima latencia, se ejercita la comunicación con mensajes vacíos.

```
# Intel (R) MPI Benchmark Suite V3.1, MPI-1 part
# Date                  : Wed Mar  3 10:45:16 2010
# Machine               : x86_64
# System                : Linux
# Release               : 2.6.16.46-0.12-smp
# Version               : #1 SMP Thu May 17 14:00:09 UTC 2007
# MPI Version           : 2.0
# MPI Thread Environment: MPI_THREAD_SINGLE
# Calling sequence was: ../IMB-MPI1 pingpong
# Minimum message length in bytes:   0
# Maximum message length in bytes:   4194304
#
# MPI_Datatype                   :   MPI_BYTE
# MPI_Datatype for reductions    :   MPI_FLOAT
# MPI_Op                         :   MPI_SUM
#
# List of Benchmarks to run: PingPong
#---------------------------------------------------
# Benchmarking PingPong
# #processes = 2
#---------------------------------------------------
#bytes     #repetitions  t[usec]     Mbytes/sec
0              1000           17.13        0.00
1              1000           17.89        0.05
2              1000           17.82        0.11
4              1000           17.95        0.21
...
1048576    40              8993.23    111.19
2097152    20              17919.20  111.61
4194304    10              35766.45  111.84
```

HPC Challenge

El *benchmark* HPC Challenge [25] (HPCC) está compuesto internamente por un conjunto de varios núcleos de cómputo: entre ellos STREAM, HPL, Ping Pong, Transformadas de *Fourier* y otros más simples ejercitando la red de comunicación.

Este benchmark muestra diferentes resultados que son representativos y puestos en consideración de acuerdo al tipo de aplicación en discusión. La mejor máquina depende de la aplicación específica a ejecutar, ya que algunas aplicaciones necesitan mejor ancho de banda de memoria, mejor canal de comunicación, o simplemente la mayor capacidad de cómputo de operaciones flotantes posible.

Una analogía interesante para entender cómo el *benchmark* se relaciona con diferentes núcleos de cómputo se muestra en la Figura 2.4. El eje de las X representa el grado de localidad espacial que posee un núcleo de cómputo. Esto se refiere a la concentración de las lecturas y escrituras en una región de memoria cercana. El eje de las Y representa el grado de localidad temporal que posee un

núcleo de cómputo. Esto se refiere al reuso para lectura o escritura de una región de memoria dada.

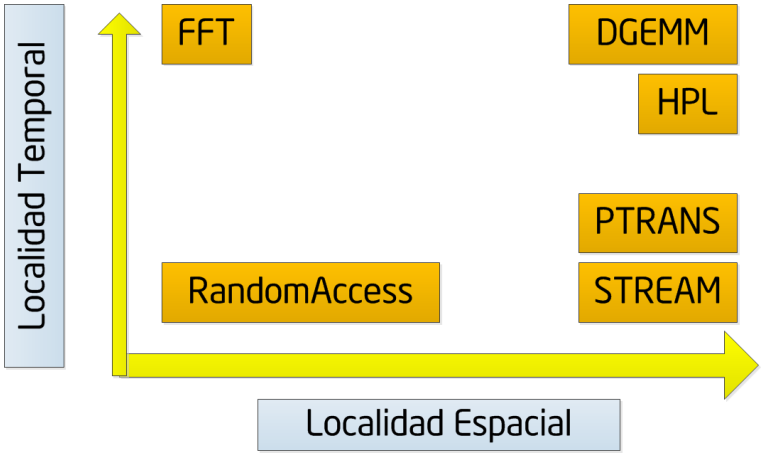

Figura 2.4: Localidad temporal versus espacial en resultados de HPCC

El núcleo *FFT* realiza transformadas de *Fourier* donde los cálculos se realizan sobre una misma zona de memoria. El núcleo *RandomAccess* es aleatorio en cuanto a su caracterización de acceso. Los núcleos *DGEMM* y *HPL* realizan multiplicaciones de matrices. El núcleo *PTRANS* incluya una rotación de una matriz densa. El núcleo *STREAM* realiza operaciones vectoriales para ejercitar la jerarquia de memoria.

Por ejemplo al tener un problema que utiliza principalmente acceso a memoria local, se puede suponer que un sistema con buenos resultados de *STREAM* va ser una buena alternativa para su ejecución.

Para una mejor comparación de resultados de HPCC se utilizan diagramas de tipo radar denominados *kiviats*, un ejemplo se muestra en la Figura 2.5. Los resultados están normalizados hacia uno de los sistemas, en este ejemplo se puede identificar mejor rendimiento en FLOPs por poseer mejores DGEMM y HPL en comparación.

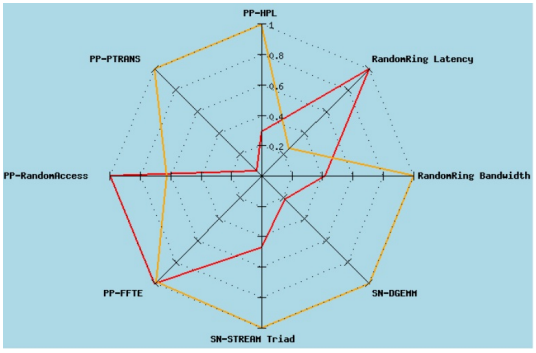

Figura 2.5: Diagrama Kiviat [Top500]

Un ejemplo de la salida que se muestra durante la ejecución se muestra a continuación.

```
This is the DARPA/DOE HPC Challenge Benchmark version 1.2.0 October 2003
Produced by Jack Dongarra and Piotr Luszczek
Innovative Computing Laboratory
University of Tennessee Knoxville and Oak Ridge National Laboratory

Begin of Summary section.

VersionMajor=1
VersionMinor=2
LANG=C
Success=1
CommWorldProcs=3
MPI_Wtick=1.000000e-06
HPL_Tflops=0.0674008
HPL_time=26.3165
HPL_eps=1.11022e-16
HPL_N=13856
HPL_NB=64
HPL_nprow=1
HPL_npcol=3
HPL_depth=2
HPL_nbdiv=2
HPL_nbmin=8
HPL_cpfact=C
HPL_crfact=R
HPL_ctop=1
HPL_order=R
dweps=1.110223e-16
sweps=5.960464e-08
HPLMaxProcs=3
HPLMinProcs=3
DGEMM_N=4618
StarDGEMM_Gflops=68.9053
SingleDGEMM_Gflops=70.2692
PTRANS_GBs=0.794254
PTRANS_time=0.479293
PTRANS_residual=0
PTRANS_n=6928
PTRANS_nb=64
PTRANS_nprow=1
PTRANS_npcol=3
MPIRandomAccess_N=134217728
MPIRandomAccess_time=30.4475
MPIRandomAccess_Check=14.0705
MPIRandomAccess_Errors=0
MPIRandomAccess_ErrorsFraction=0
MPIRandomAccess_ExeUpdates=536870912
MPIRandomAccess_GUPs=0.0176327
MPIRandomAccess_TimeBound=-1
MPIRandomAccess_Algorithm=0
RandomAccess_N=33554432
StarRandomAccess_GUPs=0.0186362
SingleRandomAccess_GUPs=0.0184568
STREAM_VectorSize=21332081
STREAM_Threads=8
StarSTREAM_Copy=4.34705
StarSTREAM_Scale=3.24366
StarSTREAM_Add=3.41196
StarSTREAM_Triad=3.46198
SingleSTREAM_Copy=4.53628
SingleSTREAM_Scale=3.38984
SingleSTREAM_Add=3.59073
SingleSTREAM_Triad=3.65083
FFT_N=8388608
StarFFT_Gflops=2.17339
SingleFFT_Gflops=2.26806
MPIFFT_N=8388608
MPIFFT_Gflops=1.7043
MPIFFT_maxErr=1.77722e-15
MPIFFT_Procs=2
MaxPingPongLatency_usec=5.37932
RandomRingLatency_usec=5.70686
MinPingPongBandwidth_GBytes=0.675574
NaturalRingBandwidth_GBytes=0.531278
RandomRingBandwidth_GBytes=0.529161
MinPingPongLatency_usec=5.24521
AvgPingPongLatency_usec=5.30978
MaxPingPongBandwidth_GBytes=0.682139
AvgPingPongBandwidth_GBytes=0.678212
NaturalRingLatency_usec=5.79357
FFTEnblk=16
FFTEnp=8
FFTEl2size=1048576

End of Summary section.
End of HPC Challenge tests.
```

2.2.2. Utilización de las Herramientas

Se recomienda un proceso de aplicación gradual empezando por herramientas generales de alto nivel que analizan la aplicación como un todo; terminando con herramientas de bajo nivel que proveen detalles complejos de granularidad más fina en partes específicas del código. Esto permite ir analizando el rendimiento sin tener que enfrentar la dificultad de un análisis complejo y extensivo desde un principio. Una lista de las herramientas más conocidas se muestra en la Tabla 2.5.

Tabla 2.5: Aplicación Gradual de Herramientas

Característica	**Herramientas**
Capacidad del sistema	Benchmark HPCC
Medición de ejecución	`time`, `gettimeofday()`, `MPI_WTIME()`
Perfil de ejecución	profilers: `gprof`, `perf`
Comportamiento de la aplicación	profilers: `gprof, perf`
Comportamiento de librerías	profilers: valgrind, MPI vampir.
Comportamiento del sistema	profilers: oprofile, perf
Vectorización	compilador: gcc
Contadores en *hardware*	oprofile, PAPI, perf

A grandes rasgos el procedimiento es el siguiente:

1. Se establece una línea de comparación al ejecutar una prueba de rendimiento del sistema, *HPCC* brinda un conjunto de métricas muy completo.
2. Se utilizan herramientas para medir el tiempo de ejecución de la aplicación sobre diferentes escenarios. `time` permite una ejecución directa sin modificación de código, `gettimeofday()` requiere modificación de código pero puede ser utilizados con mayor libertad dentro de la aplicación. En el caso de estar utilizando la librería MPI, `MPI_WTime()` y la herramienta VAMPIR[2] proveen soporte específico para análisis de rendimiento.
3. Se dimensiona el comportamiento de la aplicación mediante un perfil de ejecución y un análisis de cuello de botella utilizando `gprof`.
4. Se analiza el comportamiento del sistema ejecutando la aplicación mediante `oprofile` [3] o `perf` [4].
5. Se revisa el reporte del compilador para comprobar que se estén vectorizando los ciclos de cálculo más intensivos.
6. Se analiza el comportamiento de las unidades de cómputo utilizando soporte de *hardware* mediante herramientas como `perf`, `oprofile` y *Performance Application Programming Interface* (PAPI) [5].

2.2.3. Tiempo de Ejecución

Esta sección revisa como medir el tiempo de ejecución global de una aplicación, incluyendo ejemplos.

[2] http://www.vampir.eu
[3] http://oprofile.sourceforge.net
[4] https://perf.wiki.kernel.org
[5] http://icl.cs.utk.edu/papi

Tiempo de ejecución global

Para medir el tiempo de ejecución de un comando en consola se utiliza `time(1)`. Aunque rudimentaria, esta simple herramienta no necesita de instrumentación de código y se encuentra disponible en cualquier distribución *GNU/Linux*. El intérprete de comandos tiene su propia versión embebida, sin embargo el del sistema brinda información del uso de otros recursos del sistema, usualmente localizado en `/usr/bin/time`. Un ejemplo se demuestra en el listado 1.

Listado 1: Ejecución del Programa

```
$ /usr/bin/time -v ./program
1
        Command being timed: "./program"
        User time (seconds): 0.61
        System time (seconds): 0.00
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.62
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 4560
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 668
        Voluntary context switches: 6
        Involuntary context switches: 2
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

Reloj del sistema

La librería principal de sistema permite acceder a llamadas al sistema operativo para obtener datos precisos del paso del tiempo. Las más utilizadas son `gettimeofday(3)` y `clock(3)`, aunque éste último se comporta de manera especial al utilizar multi-threading ya que suma el tiempo ejecutado en cada unidad de cómputo.

El código en el listado 2 ejemplifica como obtener un número de segundos en una representación de punto flotante de doble precisión, permitiendo una granularidad de medición adecuada.

Listado 2: Tiempo de Ejecución

```
double wtime(void)
{
  double sec;
  struct timeval tv;

  gettimeofday(&tv, NULL);
  sec = tv.tv_sec + tv.tv_usec/1000000.0;
  return sec;
}
```

2.2.4. Perfil de Ejecución Funcional

Las herramientas de perfilado denominadas *profilers* extraen el perfil dinámico de una aplicación en tiempo de ejecución. Se instrumenta la aplicación con una opción específica que incluye información de uso de las diferentes partes del programa y los recursos del sistema como por ejemplo procesador y memoria.

La aplicación debe ejecutarse con un conjunto de datos prefijado. El conjunto de datos debe ser representativo y debe también ejercitar la aplicación por una cantidad de tiempo suficiente como para intensificar el uso de los recursos. Los datos del perfil de una ejecución son luego obtenidos en la forma de un archivo de datos, luego se procede a procesar los datos acumulados con un analizador respectivo.

Provee un perfil plano que consiste en una simple lista de las funciones ejecutadas ordenadas por la cantidad acumulada de tiempo utilizado. También provee el gráfico de llamadas anidadas, que muestra el tiempo utilizado por cada función en llamadas sucesivas. Las funciones recursivas son manejadas de manera especial ya que imposibilitan el armado de relaciones de dependencias.

Ejemplo: `gprof`

El perfil de ejecución de `gprof` muestra el tiempo individual y el tiempo acumulado en segundos de cada línea de código de la aplicación. Los binarios deben ser compilados con información extra de depuración, en el caso de `gcc`, las opciones necesarias son `-g -pg`. Si `-g` no se encuentra presente entonces no se provee el reporte detallado por línea de ejecución. Esto permite identificar donde se está consumiendo tiempo durante la ejecución. La herramienta también muestra un cuadro de las llamadas entre funciones realizadas por el programa. Esto permite visualizar el esquema de dependencias durante la ejecución.

A continuación en el listado 3 se muestra cómo realizar la compilación incluyendo información de perfilado específica, además de un caso concreto contra una aplicación simulando el juego de la vida [26].

Listado 3: Compilación con Información de Depuración

```
1 $ gcc -g -pg program.c -o program
2 $ ./program
3 $ gprof program
4 ...
```

En el listado 4 se muestra la información de las funciones del programa ordenadas por mayor impacto en el tiempo de ejecución.

Listado 4: Perfil de Rendimiento

```
1 Flat profile:
2 Each sample counts as 0.01 seconds.
3 % cumulative self self total
4 time seconds seconds calls us/call us/call name
5 37.50 0.15 0.15 48000 3.12 3.12 Life::neighbor_count(int, int)
6 ...
```

En el listado 5 se muestra la información del gráfico de llamadas del programa.

Listado 5: Gráficos de Llamadas

```
Call graph
granularity: each sample hit covers 4 byte(s) for 2.50% of 0.40 seconds
index %time    self  children    called     name
      0.02 0.15 12/12 main [2]
[1] 42.5 0.02 0.15 12 Life::update(void) [1]
      0.15 0.00 48000/48000 Life::neighbor_count(int, int) [4]
--
          0.00    0.17       1/1           _start [3]
[2]     42.5    0.00    0.17       1         main [2]
          0.02    0.15      12/12          Life::update(void) [1]
          0.00    0.00      12/12          Life::print(void) [13]
          0.00    0.00      12/12          to_continue(void) [14]
          0.00    0.00       1/1           instructions(void) [16]
          0.00    0.00       1/1           Life::initialize(void) [15]
--
```

2.2.5. Perfil de Ejecución Asistido por *Hardware*

Un *profiler* puede utilizar el *hardware* para analizar el uso de los recursos disponibles a nivel de núcleo del sistema operativo. Actúa de forma transparente a nivel global. Utiliza contadores de *hardware* del CPU y además interrupciones de un temporizador cuando no logra detectar soporte específico en *hardware*. Aunque tiene un costo adicional inherente, la sobrecarga es mínima.

Para obtener un perfil de ejecución representativo, usualmente se recomienda detener toda aplicación o servicio no relevante en el sistema. La herramienta de por si no requiere acceder al código fuente de la aplicación, pero si esta disponible el código correspondiente se muestra anotado con contadores si hay símbolos de depuración en el binario.

Los registros de *hardware* implementando contadores más utilizados son los siguientes:

1. Cantidad total de ciclos de procesador
2. Cantidad total de instrucciones ejecutadas
3. Cantidad de ciclos detenidos por espera de acceso a memoria
4. Cantidad de instrucciones de punto flotante
5. Cantidad de fallos de cache de nivel uno (L1)
6. Cantidad de instrucciones de carga y descarga

En núcleos *Linux* más nuevos que la versión 2.6, en lugar de `oprofile` se recomienda utilizar `perf`. Al estar implementados a nivel de núcleo, éstos evitan las llamadas al sistema y tienen una sobrecarga de un orden de magnitud menor que los *profilers* a nivel de aplicación. Las herramientas propietarias suelen tener acceso a contadores más específicos e incluso programables para funciones determinadas de medición.

Ejemplo: `perf`

A continuación en el listado 6 se demuestra la información provista por `perf` en sus diferentes modos de ejecución: estadísticas de contadores, perfil de sistema y por último perfil de aplicación.

Listado 6: Estadísticas de Contadores

```
$ perf stat -B program
Performance counter stats for 'program':
          5,099 cache-misses  0.005 M/sec (scaled from 66.58%)
        235,384 cache-references 0.246 M/sec (scaled from 66.56%)
      9,281,660 branch-misses 3.858 %     (scaled from 33.50%)
    240,609,766 branches 251.559 M/sec (scaled from 33.66%)
  1,403,561,257 instructions  0.679 IPC   (scaled from 50.23%)
  2,066,201,729 cycles 2160.227 M/sec (scaled from 66.67%)
            217 page-faults 0.000 M/sec
              3 CPU-migrations 0.000 M/sec
             83 context-switches 0.000 M/sec
    956.474238 task-clock-msecs 0.999 CPUs
      0.957617512  seconds time elapsed
```

En el listado 7 se muestra la salida del perfil de rendimiento. Notar que se incluye incluso la información del comportamiento del núcleo del sistema.

Listado 7: Perfil de Rendimiento

```
$ perf record ./mm
$ perf report
# Events: 1K cycles
# Overhead Command Shared Object Symbol
28.15% main mm [.] 0xd10b45
4.45% swapper  [kernel.kallsyms] [k] mwait_idle_with_hints
4.26% swapper  [kernel.kallsyms] [k] read_hpet
...
```

En el listado 8 se muestra la salida del perfil de código anotado con las instrucciones del ensamblador respectivas.

Listado 8: Código Anotado

```
 Percent |   Source code & Disassembly of program
         :   Disassembly of section .text:
         :   08048484 <main>:
         :   #include <string.h>
         :   #include <unistd.h>
         :   #include <sys/time.h>
         :
         :   int main(int argc, char **argv)
         :   {
0.00:    8048484:       55                      push   %ebp
0.00:    8048485:       89 e5                   mov    %esp,%ebp
...
0.00:    8048530:       eb 0b                   jmp 804853d <main+0xb9>
         :                           count++;
14.22:    8048532:       8b 44 24 2c             mov    0x2c(%esp),%eax
0.00:    8048536:       83 c0 01                add    $0x1,%eax
14.78:    8048539:       89 44 24 2c             mov    %eax,0x2c(%esp)
         :           memcpy(&tv_end, &tv_now, sizeof(tv_now));
         :           tv_end.tv_sec += strtol(argv[1], NULL, 10);
         :           while (tv_now.tv_sec < tv_end.tv_sec ||
         :                  tv_now.tv_usec < tv_end.tv_usec) {
         :                    count = 0;
         :                    while (count < 100000000UL)
14.78:    804853d:       8b 44 24 2c             mov    0x2c(%esp),%eax
56.23:    8048541:       3d ff e0 f5 05          cmp    $0x5f5e0ff,%eax
0.00:    8048546:       76 ea                   jbe 8048532 <main+0xae>
...
```

Este punto de análisis requiere conocimiento avanzado de como funciona el CPU utilizado, su acceso a memoria y los costos de las diferentes instrucciones soportadas. Una fuente de consulta debe incluir conceptos generales de arquitectura de procesadores [27] e información de los fabricantes [28].

2.2.6. Reporte de Vectorización

Una herramienta de bajo nivel para analizar rendimiento es el mismo compilador que debería estar vectorizando los ciclos de cómputo intensivo. Esto es muy útil para detectar si los cuellos de botella ya se encuentran optimizados o no.

Por ejemplo, GCC provee opciones específicas que deben ser provistas para mostrar el reporte. En el listado 9 se muestra la información incluída.

Listado 9: Información de Vectorización

```
$ gcc -c -O3 -ftree-vectorizer-verbose=1 ex.c
ex.c:7: note: LOOP VECTORIZED.
ex.c:3: note: vectorized 1 loops in function.
$ gcc -c -O3 -ftree-vectorizer-verbose=2 ex.c
ex.c:10: note: not vectorized: complicated access pattern.
ex.c:10: note: not vectorized: complicated access pattern.
ex.c:7: note: LOOP VECTORIZED.
ex.c:3: note: vectorized 1 loops in function.
$ gcc -c -O3 -fdump-tree-vect-details ex.c
...
```

En el caso de existir código recursivo, podemos comprobar que no suele estar soportado por los compiladores actuales. La información de vectorización sobre un código que posee recursividad se muestra en el listado 10

Listado 10: Vectorización de Código Recursivo

```
$ gcc -Wall -Wextra -O3 -ftree-vectorizer-verbose=4 -g queen.c
queen.c:22: note: vectorized 0 loops in function.
queen.c:35: note: vectorized 0 loops in function.
```

2.3. Trabajo Previo

Esta tesis se realiza como extensión al Trabajo Final Integrador *Herramientas para el Soporte de Análisis de Rendimiento* de la Especialización en Cómputo de Altas Prestaciones y Tecnología Grid [6].

Al ser un tema que es relevante en toda disciplina que realice simulaciones computarizadas, existen innumerables trabajos relacionados. Una introducción puede ser consultada en [29]. Una revisión general en [30]. Una propuesta de hacia donde va el estado del arte en [31]. Patrones útiles para ser reusados en [32]. Como capturar la información necesaria en [33]. Un esfuerzo de optimización automática en [34]. Más detalles del estado del arte del análisis del rendimiento en Cómputo de Altas Prestaciones en [35].

[6] http://sedici.unlp.edu.ar/handle/10915/29858

Capítulo 3

Descripción del Problema

Este capítulo introduce el problema a resolver. Se describen las dificultades enfrentadas al realizar análisis de rendimiento y el posterior esfuerzo de optimización. Se listan las capacidades que debe poseer una infraestructura que de soporte a estos procesos de forma automatica.

3.1. Análisis de Rendimiento

El problema concreto sobre el que se trabaja es brindar automatización para ahorrar esfuerzo y minimizar el nivel de conocimiento requerido durante el desarrollo de aplicaciones de cómputo de altas prestaciones.

3.1.1. Problemas Usuales

Los desafíos usuales que se enfrentan durante el análisis de rendimiento de un programa son: interacción humana, manejo de herramientas, recopilación de datos y su representación resumida, optimización temprana e implementación teórica de algoritmos.

Interacción Humana

La interacción humana siempre es fuente de errores involuntarios, además de malgastar el tiempo de un investigador o desarrollador en tareas factibles de ser automatizadas. Usualmente una persona es requerida para ejecutar los programas, tabular los resultados y generar gráficos para su resumen. A continuación se detalla cada uno en particular.

El análisis de rendimiento requiere de una disciplina absoluta. Una de las tareas más demandantes de tiempo es la ejecución de un programa bajo distintas configuraciones para entender su comportamiento.

Manejo de Herramientas

El aprendizaje del correcto uso y aplicación de las herramientas demanda valioso tiempo. Sin embargo las herramientas adecuadas permiten extraer información

de rendimiento de mayor granularidad y calidad de la información, posibilitando tomar una mejor decisión a la hora de focalizar los esfuerzos de optimización.

El análisis requiere el correcto uso de matemática estadística para promediar resultados, descartar ejecuciones problemáticas y establecer límites en las mejoras. Es frecuente la utilización de una hoja de cálculo para centralizar los cálculos una vez tabulados los tiempos de ejecución y demás métricas de rendimiento.

Recopilación de Datos y Representación de Resultados

El análisis requiere la recopilación de datos relevantes al rendimiento sobre el comportamiento del programa y el sistema ejecutando el mismo. Cuáles son estas métricas, cómo obtenerlas y cómo representarlas para el análisis es una tarea no trivial que requiere tiempo y experiencia en el tema.

Optimización Temprana

La optimización temprana sin tener en cuenta datos cuantitativos puede implicar que un esfuerzo importante no tenga impacto alguno en el resultado global del tiempo de ejecución de un programa.

Implementación Teórica de Algoritmos

La implementación directa de un algoritmo matemático puede garantizar su correctitud pero no su eficiencia durante su ejecución. La reutilización de librerías de dominio público ofrecidas por los fabricantes es siempre preferida ya que garantizan eficiencia con un mínimo esfuerzo de aprender como utilizarlas correctamente.

3.1.2. Métodos de Optimización

Los métodos usuales de optimización de un programa que ya utiliza un algoritmo adecuado son usualmente los siguientes:

Código: se analiza el código fuente para mejorar la predicción de saltos, realizar *inlining* de rutinas muy utilizadas, alinear código y realizar *unrolling* de ciclos para posibilitar vectorización.

Las posibles optimizaciones a nivel de código son diversas. Algunas pueden ser automáticamente introducidas por el compilador en casos donde es posible garantizar una mejora.

Ejecución: se analiza el código a nivel de ensamblador para comprobar el uso de instrucciones livianas o vectoriales, y además tratando de reducir el uso de registros.

Memoria: se analiza las estructuras de datos para incrementar la tasa de transferencia a memoria, minimizar fallas de cache, alinear datos y mejorar la localidad del acceso a los mismos.

Estas optimizaciones pueden ser realizadas manualmente a nivel de código o automáticamente por el compilador.

Precarga: se analiza el uso de instrucciones de *hardware* para precargar datos en cache cuando los algoritmos predictivos no son suficientes.

Punto Flotante: se considera el relajamiento de las reglas de redondeo y representación según los estándares, intercambiando un mayor error acumulado por mejor velocidad de procesamiento.

3.2. Infraestructura de Soporte

Los problemas usuales durante el análisis de rendimiento listados anteriormente reflejan la necesidad de utilizar soporte automátíco para la ejecución de pruebas de rendimiento, la recopilación de métricas relevantes y la generación de un reporte con gráficos comparativos e información de contexto. Los siguientes requerimientos son los necesarios para una infraestructura de análisis de rendimiento.

Reusabilidad

La infraestructura debe ser aplicable a un gran rango de programas, no requiriendo su modificación. Preferentemente, su instalación solo debe depender de la existencia previa de las mismas herramientas que un usuario podría ejecutar para obtener información relacionada al rendimiento de un programa. La infraestructura debe ser eficiente, no debe requerir ejecutar pruebas largas y tediosas sin un objetivo concreto.

Configurabilidad

La aplicación de las herramientas debe ser configurable. Los parámetros a configurar deberían incluir entre otros: la forma de compilar y ejecutar un programa, el rango de valores de entrada, el número de repeticiones de las diferentes ejecuciones del programa.

Portabilidad

La infraestructura debe ser implementada con un lenguaje portable, de ser posible basado en código abierto de forma que pueda ser revisada fácilmente por los usuarios para incluir nuevas fuentes de información o cambiar la forma en que las herramientas base son utilizadas.

Extensibilidad

La infraestructura debe poseer un diseño de fácil extensión, la incorporación de nuevas herramientas, gráficos o secciones dentro de un reporte debe ser una tarea trivial asumiendo que ya se conoce la forma manual de obtener la información requerida para ello.

Simplicidad

La infraestructura debe reutilizar las mismas herramientas disponibles en el sistema, de tal forma el usuario puede continuar el análisis de forma directa. También debe generar archivos de soporte con la información pura de los comandos ejecutados y su salida sin depurar. Debe ser posible completar un reporte entre un día de trabajo y otro sin la interacción con un usuario.

Capítulo 4

Propuesta de Solución

Este capítulo muestra la propuesta de solución, incluyendo el diseño de la misma a diferentes niveles.

4.1. Procedimiento

Un procedimiento de análisis de rendimiento debe ser iterativo de modo de garantizar un progreso continuo asociado a la cantidad de tiempo dedicado. Se propone entonces un proceso que empieza analizando las capacidades del sistema en uso, y luego incluye múltiples ciclos de recabado de resultados, identificación de cuellos de botella y la inclusión de optimizaciones para mejorar el rendimiento.

La Figura 4.1 muestra a grandes rasgos las etapas del proceso a automatizar.

Figura 4.1: Procedimiento de Análisis

Primero se establece una línea base de rendimiento del sistema utilizando pruebas de rendimiento conocidas. Luego se procede a trabajar en etapas iterativas asegurando en cada paso la estabilidad de los resultados, revisando la utilización de recursos y utilizando un perfil de ejecución para encontrar un punto de enfoque. Luego de optimizar y comprobar la mejora, se vuelve a repetir el ciclo.

A continuación se detallan los pasos a realizar, junto con preguntas que guían el análisis de rendimiento de una aplicación. La infraestructura solo implementa los pasos específicos de ejecución de herramientas para la recolección de información.

Pruebas de Referencia: el análisis de resultados de pruebas de referencia ayuda a dimensionar la capacidad máxima del sistema bajo prueba. Para ello se propone ejecutar pruebas de rendimiento sobre el sistema a utilizar para poder entender sus capacidades máximas en contraste con las teóricas.

Un ejemplo de los pasos a seguir se muestra a continuación.

Listado 11: Instalación de HPCC

```
1 $ sudo apt-get install hpcc
2 $ mpirun -n `grep -c proc /proc/cpuinfo` ./hpcc
3 $ cat hpccoutf.txt
```

Las interrogantes a resolver son las siguientes.

1. ¿Los resultados reflejan las capacidades esperadas del sistema?
2. ¿Los FLOPS se aproximan al rendimiento de un sistema similar?
3. ¿La latencia y ancho de banda de la memoria es la esperada?

Compilación del Binario: la correcta compilación del programa a optimizar es una tarea no trivial, ya que puede ser necesario incluir opciones extra de diagnóstico para soportar el análisis. Un binario requiere ser instrumentado para obtener datos de depuración detallados utilizados en el análisis de comportamiento a nivel de código.

Un ejemplo se lista a continuación.

Listado 12: Compilación del Binario

```
1 $ CFLAGS='-Wall -Wextra -O3 -g -pg' make
```

Estabilidad de Resultados: se debe comprobar la estabilidad de los resultados obtenidos para asegurarse de que las decisiones tomadas al comparar los mismos sean acertadas. De no ser así hay que revisar la configuración del sistema o el caso de prueba a utilizar como escenario principal de análisis.

Para ello se propone primero comprobar desviacion estándar de resultados para conocer la estabilidad de los mismos. Segundo, establecer cuál es el promedio geométrico a usar como referencia para comparaciones futuras.

Un ejemplo se lista a continuación.

Listado 13: Estabilidad de Resultados

```
1 $ for i in `seq 1 32`; do /usr/bin/time -v ./program >> time.csv; done
```

Las interrogantes a resolver son entonces:

1. ¿Son los resultados estables?
2. ¿La desviación estándar es significativa?
3. ¿Cuál es el promedio geométrico para comparaciones futuras?
4. ¿Es necesario incrementar el problema para mejorar la desviación?
5. ¿Es posible reducir la duración de cada prueba sin afectar la desviación?

Utilización de los Recursos: se deben ejecutar disciplinadamente múltiples pruebas variando el tamaño del problema y la cantidad de unidades de cómputo para entender el patrón de escalamiento del programa.

Para ello se propone primero escalar el problema para dimensionar la cantidad de trabajo según el tamaño del problema. Segundo, escalar cómputo para luego calcular límite de mejoras según *Amdalah* y *Gustafson*.

Un ejemplo de los pasos a seguir se muestra a continuación.

Listado 14: Escalamiento de Problema

```
1 $ for size in `seq 1024 1024 10240`; do /usr/bin/time -v ./program $size
      >> size.log; done
```

Listado 15: Escalamiento de Cómputo

```
1 $ for threads in `grep -c proc /proc/cpuinfo | xargs seq 1`; do
    OMP_NUM_THREADS=$threads ./program >> threads.log; done
```

Las interrogantes son entonces:

1. ¿Cuál es la relación entre el tiempo de las diferentes ejecuciones?
2. ¿Es la incremento del tiempo de ejecución lineal o constante?
3. ¿Cuál es la relación entre el tiempo de las diferentes ejecuciones?
4. ¿Es la relación lineal o constante?
5. ¿Qué porcentaje de la aplicación se estima paralelo?
6. ¿Cual es la mejora máxima posible?

Perfil de Código: la aplicación de herramientas de bajo nivel permiten entender con gran detalle el comportamiento del programa: ya sea la secuencia de llamadas que utiliza, el mapeo de porciones de código a utilización de recursos, y cuales optimizaciones automáticas se realizan.

Para ello se propone inicialmente generar el perfil de llamadas a funciones dentro de la aplicación para revisar el diseño de la misma y los posibles cuellos de botella a resolver. Luego utilizar el profiler a nivel de sistema y por último comprobar vectorizaciones.

Un ejemplo de los pasos a seguir se muestra a continuación.

Listado 16: Generación de Perfil de Rendimiento

```
$ gcc -g -pg program.c -o program
$ ./program
...
$ gprof --flat-profile --graph --annotated-source app
...
```

Listado 17: Generación de Perfil de Sistema

```
$ prof stat ./program
$ prof record ./program
$ prof report
```

Listado 18: Información de Vectorización

```
$ gcc -Wall -Wextra -O3 --report-loop
```

Las interrogantes son entonces:

1. ¿Cómo está diseñada la aplicación?
2. ¿Que dependencias en librerías externas tiene?
3. ¿Implementa algún núcleo de cómputo conocido encuadrado dentro de librerías optimizadas como BLAS?
4. ¿En que archivos, funciones y líneas se concentra la mayor cantidad de tiempo de cómputo?
5. ¿Cómo se comporta el sistema durante la ejecución de la aplicación?
6. ¿Son las métricas de contadores de *hardware* las esperadas?
7. ¿Es la aplicación la gran concentradora de los recursos disponibles?
8. ¿Qué instrucciones de *hardware* son las mayormente utilizadas?
9. ¿Hay ciclos que no pueden ser automáticamente vectorizados?
10. ¿Pueden los ciclos no optimizados ser modificados?

Optimización: este paso del proceso propuesto es el único que necesita intervención del investigador ya que la optimización es una tarea creativa imposible de automatizar en términos generales.

Luego de realizar cualquier optimización se reinicia el ciclo para mantener o descartar la optimización hecha.

4.2. Infraestructura

El procedimiento anterior se implementó como una infraestructura automática de generación de reportes de rendimiento denominada `hotspot` por ser la traduccion de cuello de botella [1].

La automatización se comporta del mismo modo que un usuario realizando un procedimiento sistemático de análisis de rendimiento. Es decir que ejecuta utilidades del sistema como `gcc`, `make`, `prof`, `gprof`, `pidstat` para obtener la información relevante. LaTeX se utiliza para la generación del reporte final.

[1] El proyecto `hotspot` está disponible en https://github.com/moreandres/hotspot

Las limitaciones *a-priori* que posee la infraestructura son en materia de portabilidad y aplicación. Por el lado de la portabilidad, sólo se soportan sistemas *GNU/Linux recientes*, siendo necesarias el conjunto de utilidades de soporte especificadas anteriormente. Por el lado de la aplicación, sólo se soportan programan utilizando tecnología *OpenMP*.

4.3. Teoría de Operación

Esta sección detalla el funcionamiento de la infraestructura relacionando los componentes.

4.3.1. Arquitectura

A grandes rasgos el usuario utiliza la infraestructura para analizar un programa. La infraestructura ejercita el programa múltiples veces hasta obtener la información necesaria para generar un reporte resumiendo los resultados.

La interacción con la infraestructura se detalla en la Figura 4.2 a continuación.

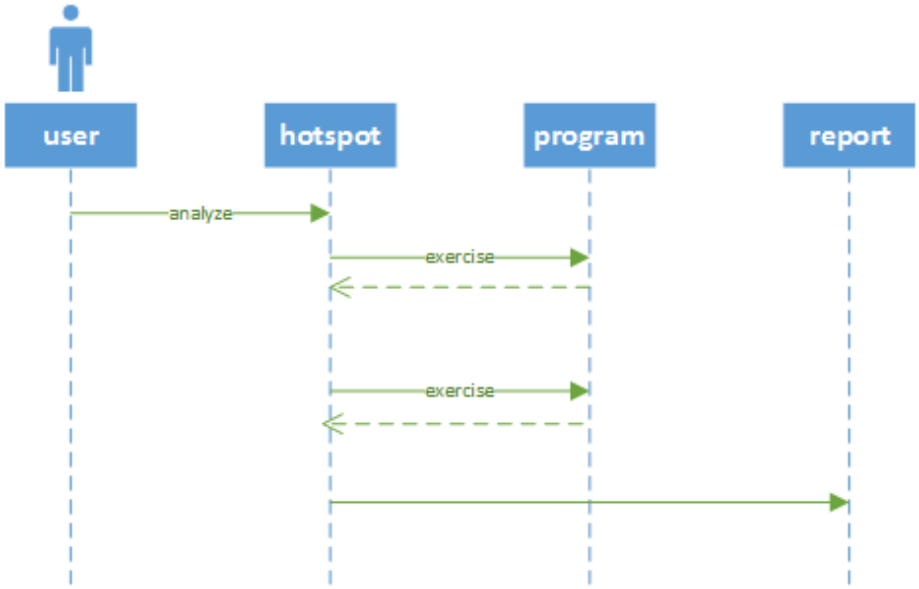

Figura 4.2: Diagrama de Secuencia

4.3.2. Funcionamiento Interno

Se utiliza un directorio escondido dentro de la carpeta que contiene la aplicación y se guardan en sub-directorios por fecha y hora las diferentes ejecuciones. Esta información puede ser usada para una comparación histórica de resultados.

Inicialmente se ejecuta la aplicación múltiples veces para validar que los resultados poseen una desviación estable. Se resume esta información con un histograma y se hace una aproximación a una distribución de resultados normales como

comparación. Se toma como referencia la media geométrica de los resultados. La primer ejecución se descarta.

Se ejercita la aplicación dentro del rango de tamaño del problema especificado, por cada punto en el rango se ejecuta múltiples veces para luego tomar un promedio. Se grafican los resultados en un gráfico de escalamiento donde se sobrepone una curva ideal suponiendo que un problema del doble de tamaño va a necesitar el doble de tiempo de cómputo.

Se detecta cuantas unidades de procesamiento hay en el sistema y se realizan pruebas utilizando más unidades incrementalmente. Se utiliza una y luego se itera hasta utilizar todas, se ejecuta múltiples veces la aplicación y se promedia el resultado. También se sobrepone una curva ideal suponiendo escalamiento ideal como en el caso anterior.

Utilizando la información anterior, se calcula el porcentaje de ejecución en serie y en paralelo. Con esta información se calculan también los limites según leyes de *Amdalah* y *Gustafson* para los procesadores disponibles y para un número grande como para visualizar el caso de poseer infinitas unidades de cómputo.

Se recompila la aplicación con información extra de depuración y se obtiene información del perfil de ejecución. Se detallan las funciones, las líneas de código y el *assembler* implementado por las mismas.

4.4. Diseño de Alto Nivel

El diseño de la infraestructura refleja la interacción manual con el sistema. Se depende de la presencia de herramientas de línea de comando, del programa a analizar en particular, y de LaTeX. Estos componentes se detallan en la Figura 4.3.

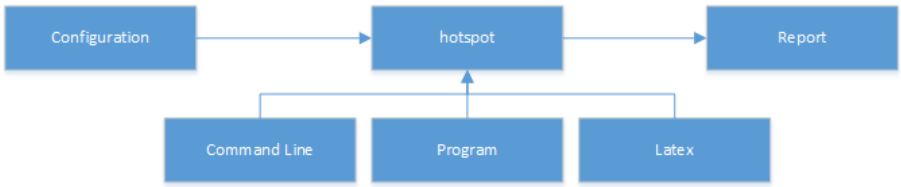

Figura 4.3: Diseño de Alto Nivel

1. **Configuración**: el componente lee información de configuración y la deja disponible para los demás componentes.
2. **Reporte**: el componente guarda valores de variables que luego utiliza para generar el reporte final.
3. **Infraestructura**: este componente utiliza información de configuración para generar ejercitar el programa y generar un reporte con los resultados.

La siguiente sección revisa más en detalle como la infraestructura está compuesta internamente.

4.5. Diseño de Bajo Nivel

La infraestructura se implementa mediante una jerarquía de clases bastante simple, siendo fácil de extender de ser necesario. Las diferentes clases disponibles se muestran en la Figura 4.4.

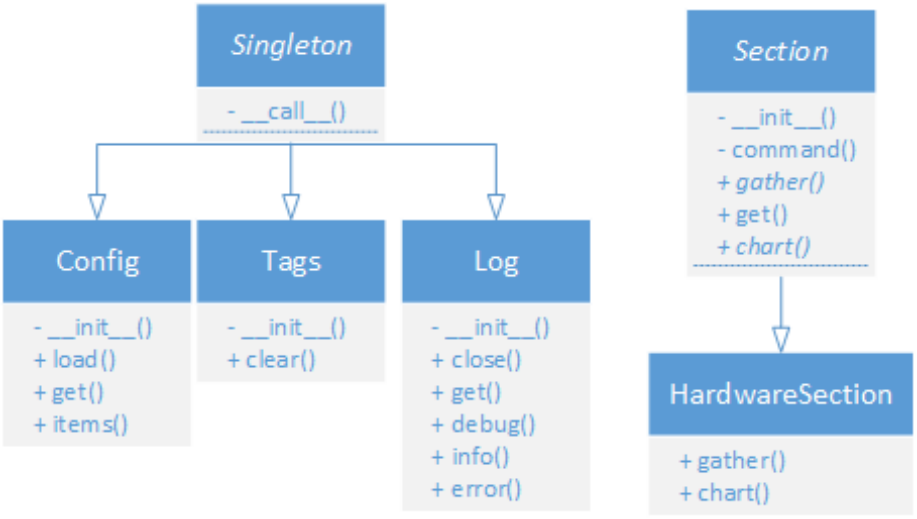

Figura 4.4: Diseño de Bajo Nivel

Aunque solo se incluye *HardwareSection* como ejemplo, existe un objeto sección por cada sección dentro del reporte.
A continuación se listan la totalidad de las clases junto con una descripción de las mismas.

1. *Singleton*: Patrón de Instancia Única. Es utilizado por otras clases para garantizar que solo existe una instancia única.
2. *Tags*: Almacenamiento de Palabras Clave. Es utilizado para guardar palabras clave que son utilizadas para generar el reporte.
3. *Log*: Generador de Mensajes. Es utilizado para estructurar los mensajes durante la ejecución.
4. *Config*: Administración de la Configuración. Es utilizado para leer y consultar la configuración.
5. *Section*: Sección del Reporte. Es utilizado como base de otras secciones.
6. *HardwareSection*: Descripción del *Hardware*. Es utilizado para obtener información del *hardware* disponible.
7. *ProgramSection*: Detalles sobre el Programa. Es utilizado para obtener información del programa a analizar.
8. *SoftwareSection*: Descripción del *Software*, Es utilizado para obtener información del *software* disponible.
9. *SanitySection*: Chequeo Base. Es utilizado para comprobaciones básicas del programa.
10. *FootprintSection*: Chequeo de Estructuras. Es utilizada para controlar la composición de estructuras.

11. *BenchmarkSection*: Pruebas de Rendimiento, Es utilizada para obtener información de rendimiento.
12. *WorkloadSection*: Caso de Prueba. Es utilizada para obtener información del caso de prueba.
13. *ScalingSection*: Escalamiento de Problema. Es utilizada para entender como escala el tamaño del problema.
14. *ThreadsSection*: Hilos. Es utilizada para entender como escalan las unidades de cómputo.
15. *OptimizationSection*: Optimización. Es utilizada para entender el efecto de los diferentes niveles de optimización del compilador.
16. *ProfileSection*: Sección sobre el Perfil de Rendimiento. Permite entender en que partes del programa se invierte el tiempo de ejecución.
17. *ResourcesSection*: Utilización de Recursos. Permite entender como se utilizan los recursos del sistema.
18. *AnnotatedSection*: Código Anotado. Incluye código ensamblador de los cuellos de botella.
19. *VectorizationSection*: Vectorización de Ciclos. Es utilizado para entender cuales ciclos fueron vectorizados o no.
20. *CountersSection*, Información de Contadores de *Hardware*. Es utilizado para resumir la información de los contadores de *hardware*.
21. *ConfigSection*, Configuración. Es utilizado para revisar la configuración del análisis.

4.5.1. Implementación

La implementación se desarrolló sobre una plataforma *GNU/Linux*, utilizando la distribución `Ubuntu 14.04.1 LTS`. El lenguaje utilizado `Python 2.7.6`, publicado en *Python Software Foundation Package Index* como `hotspot` versión 0.3. El repositorio de código también se encuentra disponible [2]. La licencia del código fuente es GPLv2 de modo de fomentar la colaboración de sus usuarios.

Se reutilizan librerías como `matplotlib.pyplot` [36] y `numpy` [37] para gráficar los resultados obtenidos durante las pruebas de rendimiento. El código fuente sigue convenciones estandares del lenguaje, el analizador estático `pylint` da un puntaje de `8.4` sobre `10` en cuanto a buenas prácticas.

4.5.2. Archivos

Una vez instalado el paquete en el sistema, cualquier modificación es posible ya que se utiliza un lenguaje interpretado. Como referencia rápida, la ubicación de los archivos suele ser:
`/usr/local/lib/python2.7/dist-packages/hotspot-0.0.2-py2.7.egg`.

Los archivos más importantes según su contenido son los siguientes:

Lógica Principal : La lógica que obtiene información para ser incluida en el reporte se puede encontrar en `hotspot/hotspot.py`.

Plantilla del Reporte : La plantilla LaTeXque contiene el formato del reporte se puede encontrar en `config/hotspot.tex`.

[2] https://github.com/moreandres/hotspot

Configuración : Un archivo de configuración de ejemplo puede encontrarse en `config/hotspot.cfg`.

4.5.3. Configuración

El listado 19 muestra la pantalla de ayuda de la infraestructura. La herramienta de línea de comando toma como parámetro principal un archivo de configuración donde se describen las características del programa a analizar.

Listado 19: Ayuda de `hotspot`

```
$ hotspot --help
usage: hotspot [-h] [-v] [--config CONFIG] [--debug]

Generate performance report for OpenMP programs.

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  --config CONFIG, -c CONFIG
                        path to configuration
  --debug, -d           enable verbose logging

Check https://github.com/moreandres/hotspot for details.
```

Entre las características necesarias se incluye la forma de compilar la aplicación, ya que para algunos pasos de análisis se necesita incorporar información extra de depuración. El archivo de configuración también permite la definición de tareas como configuración y ejecución del programa con parámetros específicos de tamaño o resolución del problema de entrada. Otro parámetro necesario es el rango de tamaños de problema a utilizar durante las ejecuciones, definidos como una secuencia de formato compatible con la herramienta *Unix* `seq` [3].

Un ejemplo del archivo de configuración requerido se muestra en el listado 20.

Listado 20: Configuración de `hotspot`

```
# hotspot configuration file

[hotspot]
# python format method is used to pass parameters

# range is a seq-like definition for problem size
range=1024,2048,256

# cflags are the compiler flags to use when building
cflags=-O3 -Wall -Wextra

# build is the command used to build the program
build=CFLAGS='{0}' make

# clean is the cleanup command to execute
clean=make clean

# run is the program execution command
run=OMP_NUM_THREADS={0} N={1} ./{2}

# count is the number of runs used to check for workload stabilization
count=16
```

[3] http://man.cat-v.org/unix_8th/1/seq

4.6. Reporte Generado

A continuación se explica el contenido de cada sección del reporte generado en particular. En el Apéndice B se adjuntan ejemplos utilizando núcleos de cómputo conocidos como la múltiplicación de matrices, la distribución de calor en dos dimensiones y la generación de fractales.

4.6.1. Consideraciones Generales

Las consideraciones generales tenidas en cuenta en todas las secciones fueron las siguientes:

Formato portable similar a un artículo de investigación: Esto resulta en un documento que cualquiera puede consultar, con un formato familiar.

Hipervínculos a archivos con la información pura: Esto permite que al identificar algun punto interesante, se pueda consultar la información tal como fue producida por la herramienta en cuestión.

Explicación breve del objetivo de la sección y/o gráficos: Esto permite entender rápidamente la información de cada sección.

Inclusión de líneas de tendencía y comportamiento ideal en gráficos: Esto permite una rápida comparación con el escenario ideal.

Referencias a material de consulta: Esto permite ahondar en detalles en caso de ser necesario.

Utilización del inglés: Esto permite compartir los resultados fácilmente entre grupos de trabajos distribuídos internacionalmente.

4.6.2. Consideraciones Particulares

Resumen

El resumen incluído en la primer página introduce el reporte junto con información sobre la infraestructura utilizada y la ubicación específica de la información de ejecución para su consulta de ser necesaria.

Contenido

Contiene un índice de secciones en formato reducido de dos columnas con hipervínculos, de modo de facilitar la búsqueda rápida de la información.

Programa

Se incluyen detalles sobre el programa bajo análisis. Esto permite recordar la versión del programa, la fecha y hora exacta del análisis, y los parámetros de entrada utilizados con el mismo.

Capacidad del Sistema

Se incluyen detalles sobre la configuración de *hardware* y *software* del sistema. En cuanto a *hardware* se detallan cantidad de memoria, modelo de CPU, y además la descripción de los controladores de placa madre, almacenamiento y red existentes. En cuanto a *software* se detallan la versión del sistema operativo y su núcleo, junto con las versiones del compilador y librería principal del sistema predeterminados. Esto permite comparar los resultados contra sistemas similares, o validar que una distinta configuración impacta o no en la ejecución del programa.

Además se incluye información de referencia obtenida de la prueba de rendimiento *HPCC*, detallando las pruebas individuales realizadas por la misma. Esto permite rápidamente entender las capacidades del sistema en núcleos de cómputo conocidos.

Carga de Trabajo

Se incluye información sobre el caso de prueba ejecutado. Se detalla el tamaño del mismo en memoria y la composición de sus estructuras ya que esto impacta directamente sobre la utilización de la jerarquía de memoria del sistema.

Se incluye también un resumen estadístico de la estabilidad del caso de prueba luego de repetirlo varias veces. Un histograma muestra la distribución de los resultados y el promedio geométrico resultante a ser utilizado como línea base.

También se incluye un gráfico demostrando los tiempos de ejecución del caso de prueba bajo diferentes niveles de optimización en el compilador. Esto permite entender el grado de optimización que ya posee un programa en materia de vectorización.

Escalabilidad

Se incluyen gráficos resumiendo la escalabilidad del programa al aumento el tamaño del problema y también por separado la cantidad de unidades de cómputo. Se superpone el escalamiento ideal como punto de comparación. Esto permite entender el grado de optimización del programa.

Se estima la proporción paralelo/lineal del programa de los datos de escalabilidad, y junto con ello se calculan mejoras teóricas máximas bajo las leyes de *Amdalah* y *Gustafson*. Este dato es importante para entender el tope de mejora posible.

Perfil de Ejecución

Se incluye información sobre las funciones y líneas de código más usadas del programa durante su ejecución. Esto nos focaliza explícitamente los futuros esfuerzos de optimización, ya que garantiza un alto impacto global.

En una sección aparte se incluyen gráficos sobre el uso de los recursos del sistema durante una ejecución del programa. Incluyendo utilización de CPU, memoria y lectura/escritura de disco.

Se listan también cuellos de botella, junto con el código de ensamblador respectivo. Esta valiosa información nos permite entender que tipo de instrucciones se están utilizando para llevar a cabo el trabajo más intenso del programa.

Bajo Nivel

Se incluye el reporte de vectorización emitido por el compilador. Nos permite comprobar si todos los ciclos dentro del código fuente están siendo vectorizados. En caso negativo, se muestra información relevante de la razón.

Se incluye además el reporte de contadores de *hardware* relacionados con el rendimiento de la unidad de procesamiento. Aunque requiere un entendimiento avanzado de la arquitectura del sistema [27], muchos problemas pueden ser identificados con esta información.

Referencias

Se incluye una lista de publicaciones relacionadas, las cuales son referenciadas en las secciones anteriores. Esto permite a los usuarios del reporte conocer más de los puntos relevantes de cada sección si así lo desean.

Capítulo 5

Casos de Aplicación

Este capítulo contiene casos de estudio mostrando los resultados de aplicar la infraestructura desarrollada a aplicaciones sintéticas implementando núcleos de cómputo de alto rendimiento conocidos. Los reportes generados están incluidos en su totalidad dentro del apéndice B, en particular secciones B.1, B.2 y B.3.

5.1. Sistema de Prueba

La configuración del sistema obtenida en todos los casos de aplicación es la misma ya que todas las pruebas fueron realizadas en el mismo sistema.

La infrastructura obtuvo correctamente información de *hardware* del sistema incluyendo cantidad de memoria disponible (9784 MiB), modelo de la unidad de procesamiento (i5-320M @ 2.6 GHz), modelo del controlador de la interfaz de red (82540EM) y modelo de placa madre más sus componentes (440FX). Las especificaciones listadas de memoria, procesador, placa madre y almacenamiento pueden ser utilizadas para buscar información de rendimiento publicada oficialmente por cada fabricante [1].

La infraestructura informó correctamente que el sistema corre un núcleo *Linux* versión 3.16.0 en un sistema *Ubuntu* versión 14.04, con un compilador *GCC* versión 4.8.2 y una librería *GNU C* versión 2.19. La identificación del *software* utilizado en el sistema permite ser más específicos al comparar los resultados obtenidos.

La infrastructura ejecutó exitosamente pruebas de rendimiento donde entre otras se incluyen *DGEMM* (con 1.8 mflops) y *STREAM* (con 6.96 MB/s). Estos datos sirven como límite de capacidad del sistema en la práctica ya que son núcleos de cómputo denso diseñados para obtener rendimiento bruto.

[1] Por ejemplo el procesador utilizado posee 2 cores permitiendo 4 hilos corriendo en paralelo, con frecuencia base de 2.6 GHz y máxima de 3.3 GHz según Automated Relational Knowledgebase (ARK)

5.2. Código de Prueba

Esta sección incluye una revisión de los resultados específicos obtenidos al aplicar la infraestructura a códigos de prueba. Se seleccionaron implementaciones de núcleos de cómputo simples pero suficientemente interesantes para demostrar las capacidades de la infrastructura. Entre ellos una implementación directa (no optimizada) de multiplicación de matrices, una simulación de transferencia de calor con un componente de ejecucion serial significativo y por último una implementación de cálculo de fractales sobre números complejos.

5.2.1. Multiplicación de Matrices

La multiplicación de matrices es una operación fundamental en diversos campos de aplicación científica como la resolución de ecuaciones lineales y la representación de grafos y espacios dimensionales. Por ello existe abundante material sobre el tema. Se tomó como ejemplo una implementación utilizando OpenMP [2]. El código utilizado se muestra a continuación en el listado 21, y la configuración del caso de prueba en el listado 22.

Listado 21: Código de Multiplicación de Matrices

```
1  int main()
2  {
3    int size = 1024;
4    if (getenv("N"))
5      size = atoi(getenv("N"));
6
7    float *a = malloc(sizeof(float) * size * size);
8    float *b = malloc(sizeof(float) * size * size);
9    float *c = malloc(sizeof(float) * size * size);
10
11   int i, j, k;
12
13   for (i = 0; i < size; ++i) {
14     for (j = 0; j < size; ++j) {
15       a[i+j*size] = (float) (i + j);
16       b[i+j*size] = (float) (i - j);
17       c[i+j*size] = 0.0f;
18     }
19   }
20
21 #pragma omp parallel for shared(a,b,c)
22   for (i = 0; i < size; ++i) {
23     for (j = 0; j < size; ++j) {
24       for (k = 0; k < size; ++k) {
25         c[i+j*size] += a[i+k*size] * b[k+j*size];
26       }
27     }
28   }
29
30   return 0;
31 }
```

Listado 22: Caso de Prueba de Multiplicación de Matrices

```
1  # hotspot configuration file
2  [hotspot]
3  range=1024,2048,256
4  cflags=-O3 -Wall -Wextra
5  build=CFLAGS='{0}' make
```

[2] http://blog.speedgocomputing.com/2010/08/parallelizing-matrix-multiplication.html.

```
6 clean=make clean
7 run=OMP_NUM_THREADS={0} N={1} ./{2}
8 count=8
```

El reporte generado por la infraestructura puede consultarse en el apéndice B.1.

El reporte de rendimiento nos demuestra lo siguiente:

1. El caso de prueba es muy estable, con una desviación estandar mucho menor a 1.

   Esto implica que el tamaño de los datos de entrada son apropiados para realizar un análisis de rendimiento sin introducir datos aleatorios de ejecución.
2. Las optimizaciones que realiza el compilador demuestran un impacto importante en el tiempo de ejecución.

   Esto implica que las optimizaciones automáticas realizadas por el compilador encuentran en el código oportunidades claras de optimización.
3. Los tiempos de ejecución al escalar el problema descubren un pico que posiblemente implique un error de implementación de algoritmo.
4. Los tiempos de ejecución al incrementar las unidades de cómputo revelan un problema ya que no muestra mejoras constantes.
5. Se estima un 75 % de paralelismo, con una mejora teórica máxima de 3.5 veces sin escalar el tamaño del problema si utilizamos el mismo algoritmo.

   Los últimos tres items muestran que la implementación del código no está optimizada para los recursos de cómputo disponibles.

   En este caso es recomendable la revisión de publicaciones en el tema y la reutilización de librerías ya optimizadas.
6. El cuello de botella se localiza en el código que obtiene el valor parcial de un elemento de la matriz resultante.
7. No hay lecturas ni escrituras al disco, por lo que no se está perdiendo rendimiento al utilizar memoria virtual.

   Esto implica que el tamaño elegido de problema puede ser cálculado integramente utilizando la memoria principal.
8. El cuello de botella principal utiliza la instrucción `add`, la cual no es vectorial.

   Esto implica que el código no está siendo totalmente vectorizado, desaprovechando la capacidad de cómputo disponible.
9. Las vectorizaciones no son realizadas exitosamente por el compilador en ningún ciclo dentro del código.

   Esto implica que los ciclos de cómputos no están siendo vectorizados, desaprovechando la capacidad de cómputo disponible.
   Una comparación de rendimiento de diferentes algoritmos de multiplicación de matrices puede encontrarse en [6].

5.2.2. Transmisión de Calor en 2 Dimensiones

Otro problema interesante es la simulación de transferencia de calor en un plano bidimensional. Se utiliza una grilla donde cada celda transfiere calor a sus vecinos en una serie finita de ciclos simulando el paso del tiempo [3]. El código utilizado

[3] http://www.rblasch.org/studies/cs580/pa5/#Source+Code-N100AC.

es incluido a continuación en el listado 23, la configuración del caso de prueba en el listado 24.

Listado 23: Código de Transmisión de Calor en 2 Dimensiones

```
1  #define CRESN    302                /* x, y resolution */
2  #define RESN     300
3  #define MAX_COLORS 8
4  #define MAX_HEAT   20.0
5
6  double solution[2][CRESN][CRESN], diff_constant;
7  int cur_gen, next_gen;
8  double sim_time, final = 10000.0, time_step = 0.1, diff = 10000.0;
9
10 void compute_one_iteration();
11 void setup();
12
13 int main()
14 {
15   final = 1024;
16   if (getenv("N"))
17     final = atoi(getenv("N"));
18
19   int temp;
20   setup ();
21   for (sim_time = 0; sim_time < final; sim_time += time_step)
22     {
23       compute_one_iteration (sim_time);
24       temp = cur_gen;
25       cur_gen = next_gen;
26       next_gen = temp;
27     }
28   return 0;
29 }
30
31 void setup()
32 {
33   int i, j;
34 #pragma omp parallel for shared(solution) private(i,j)
35   for (i = 0; i < CRESN; i++)
36     for (j = 0; j < CRESN; j++)
37       solution[0][i][j] = solution[1][i][j] = 0.0;
38
39   cur_gen = 0;
40   next_gen = 1;
41   diff_constant = diff * time_step / ((float) RESN * (float) RESN);
42 }
43
44 void compute_one_iteration()
45 {
46   int i, j;
47   /* set boundary values */
48   for (i = 0; i < CRESN; i++) {
49       if (i < 256 || i > 768)
50         solution[cur_gen][i][0] = solution[cur_gen][i][1];
51       else
52         solution[cur_gen][i][0] = MAX_HEAT;
53     }
54   for (i = 0; i < CRESN; i++)
55       solution[cur_gen][i][CRESN - 1] = solution[cur_gen][i][CRESN - 2];
56
57   for (i = 0; i < CRESN; i++) {
58       solution[cur_gen][0][i] = solution[cur_gen][1][i];
59       solution[cur_gen][CRESN - 1][i] = solution[cur_gen][CRESN - 2][i];
60     }
61
62 #pragma omp parallel for shared(solution,cur_gen,next_gen,diff_constant)
       private(i,j)
63   for (i = 1; i <= RESN; i++)
64     for (j = 1; j <= RESN; j++)
65       solution[next_gen][i][j] = solution[cur_gen][i][j] +
66         (solution[cur_gen][i + 1][j] +
67          solution[cur_gen][i - 1][j] +
68          solution[cur_gen][i][j + 1] +
69          solution[cur_gen][i][j - 1] -
70          4.0 * solution[cur_gen][i][j]) * diff_constant;
71 }
```

Listado 24: Caso de Prueba de Distribución de Calor en 2 Dimensiones

```
1 [hotspot]
2 range=16384,32768,1024
3 cflags=-O3 -Wall -Wextra
4 build=CFLAGS='{0}' make
5 clean=make clean
6 run=OMP_NUM_THREADS={0} N={1} ./{2}
7 count=8
```

El reporte generado por la infraestructura puede consultarse en el apéndice B.2.

Algunas conclusiones que pueden obtenerse son:

1. El caso de prueba es estable.
2. Las optimizaciones del compilador tienen impacto acotado.
3. El tiempo de ejecución al incrementar el tamaño del problema no crece monotónicamente.
4. El tiempo de ejecución al sumar más unidades de cómputo no decrece sistemáticamente.
5. El paralelismo en el programa solo alcanza el 50 %, por lo tanto el límite máximo de mejora es de 2 veces.
6. Hay dos cuellos de botella con un 25 % y un 15 % del tiempo de ejecución.
7. El 35 % del tiempo de ejecución lo ocupa la instrucción `mov`.
8. Los ciclos no están siendo vectorizados.

5.2.3. Conjunto de *Mandelbrot*

Los conjuntos de *Mandelbrot* resultan de un cálculo de una secuencia de operaciones sobre números complejos que no tienden a infinito. Si se supone que los componentes real y complejo corresponden a las coordenadas de un gráfico de dos dimensiones, las imágenes resultantes simulan figuras fractales [38]. El código utilizado se incluye a continuación en el listado 25, la configuración utilizada para el caso de prueba en el listado 26.

Listado 25: Código de Conjunto de Mandelbrot

```
#define X_RESN 1024
#define Y_RESN 1024
#define X_MIN -2.0
#define X_MAX 2.0
#define Y_MIN -2.0
#define Y_MAX 2.0

typedef struct complextype
{
  float real, imag;
} Compl;

int main()
{
  int i, j, k;
  Compl z, c;
  float lensq, temp;
  int iters;
  int res[X_RESN][Y_RESN];

  iters = 1024;
  if (getenv("N"))
    iters = atoi(getenv("N"));

#pragma omp parallel for shared(res, iters) private(i, j, z, c, k, temp,
    lensq)
  for (i = 0; i < Y_RESN; i++)
    for (j = 0; j < X_RESN; j++)
      {
        z.real = z.imag = 0.0;
        c.real = X_MIN + j * (X_MAX - X_MIN) / X_RESN;
        c.imag = Y_MAX - i * (Y_MAX - Y_MIN) / Y_RESN;
        k = 0;

        do
          {
            temp = z.real * z.real - z.imag * z.imag + c.real;
            z.imag = 2.0 * z.real * z.imag + c.imag;
            z.real = temp;
            lensq = z.real * z.real + z.imag * z.imag;
            k++;
          }
        while (lensq < 4.0 && k < iters);

        if (k >= iters)
          res[i][j] = 0;
        else
          res[i][j] = 1;

      }

  assert(res[0][0]);

  return 0;
}
```

Listado 26: Caso de Prueba de Conjunto de Mandelbrot

```
[hotspot]
range=16384,32768,1024
cflags=-O3 -Wall -Wextra
build=CFLAGS='{0}' make
clean=make clean
run=OMP_NUM_THREADS={0} N={1} ./{2}
count=8
```

El reporte generado puede consultarse en el apéndice B.3.

Algunas conclusiones que pueden obtenerse al ejecutar la infraestructura desarrollada sobre una implementación básica son:

1. La estructura utilizada para representar números complejos está alineada y no posee huecos.
2. El caso de prueba es muy estable.
3. La proporción del trabajo en paralelo es demasiado alta e incluso mejor que un caso ideal. Esto refleja un problema ya que es imposible.
4. Cerca del 50 % del tiempo se invierte en una única línea de código.
5. Algunos ciclos ya se encuentran optimizados, se identifica el uso de instrucciones vectoriales como `movss, addss`.

Capítulo 6

Conclusiones y Trabajo Futuro

Este capítulo concluye revisando los objetivos propuestos y posibles líneas de investigación como continuación.

6.1. Conclusiones

La optimización del rendimiento de una aplicación es algo no trivial. Es preciso realizar un análisis disciplinado del comportamiento y del uso de los recursos antes de empezar a optimizar. Las mejoras pueden ser no significativas si son realizadas en el lugar incorrecto.

Esta tesis soporta los primeros pasos de análisis de rendimiento para expertos del dominio de un problema científico utilizando computación de altas prestaciones. Se provee una metodología de uso de herramientas de soporte para principiantes, que puede ser utilizada como una lista de pasos resumidos para usuarios casuales, e incluso como referencia de consulta diaria para expertos.

Se resume también el estado del arte del análisis de rendimiento en aplicaciones de cómputo de altas prestaciones. Respecto a herramientas de soporte, se detallan diferentes opciones y se demuestra su aplicación en varios problemas simples aunque suficientemente interesantes para mostrar los beneficios de un análisis automático.

Este estudio propone un proceso de análisis de rendimiento gradual e iterativo para cualquier investigador. Incluyendo soporte automático para la aplicación de las herramientas y la generación integrada de reportes de rendimiento.

Se desarrolló como estaba planeado una infraestructura de soporte que permite a un desarrollador especialista en el dominio de un problema obtener rápidamente información cuantitativa del comportamiento de un programa *OpenMP*, incluyendo utilización de recursos y cuellos de botella para así dirigir los esfuerzos de optimización.

La infraestructura fue construída sobre los comandos que un desarrollador utilizaría. Realizar cualquier modificación es entonces trivial para contemplar nuevas herramientas o agregar nuevas secciones en los reportes generados como etapa final.

6.2. Trabajo Futuro

Las posibilidades de extensión de esta infraestructura son muchas, cada sección del reporte final puede incluir información más detallada o utilizar otras herramientas. En particular la sección de contadores de *hardware* solo contiene una lista de contadores generalizada para cualquier arquitectura; si se asume una arquitectura dada, se puede fácilmente proveer información más especifica y por lo tanto más útil.

La utilización de estas ideas en una aplicación del mundo real es materia pendiente, otra posibilidad es re-implementar desde cero alguna aplicación científica en colaboración con algún grupo de investigación y realizar varios ciclos de optimización para validar su utilidad.

Otra posibilidad consiste en incorporar soporte para programas basados en la librería de comunicación MPI. Una vez que los programas son optimizados para ejecutarse eficientemente sobre un solo sistema, el siguiente paso consiste en utilizar varios sistemas en conjunto como se detalla en [39] [40].

Por último, aunque la generación de un documento portable en formato PDF es simple y útil, se puede pensar como siguiente paso en utilizar una tecnología que permita la generación de reportes dinámicos que posibiliten a los expertos modificar la representación de la información. Por ejemplo, una implementación en HTML5 o en un formato compatible con planillas de cálculo utilizando tablas filtrables y gráficos configurables puede ser interesante y mejorar la calidad del análisis de rendimiento.

Bibliografía

[1] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38(3):451–460, June 2010.

[2] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM.

[3] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, 2010.

[4] Steven P. Levitan and Donald M. Chiarulli. Massively parallel processing: It's dÉjÀ vu all over again. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 534–538, New York, NY, USA, 2009. ACM.

[5] Intel Corporation. *Intel Math Kernel Library. Reference Manual.* Intel Corporation, 2009.

[6] Andres More. A Case Study on High Performance Matrix Multiplication. Technical report, Universidad Nacional de La Plata, 2008.

[7] Rafael Garabato, Victor Rosales, and Andres More. Optimizing Latency in Beowulf Clusters. *CLEI Electron. J.*, 15(3), 2012.

[8] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[9] Andres More. Lessons Learned from Contrasting BLAS Kernel Implementations. In *XVIII Congreso Argentino de Ciencias de la Computacion*, 2013.

[10] Fernando Tinetti and Andres More. Hotspot: a Framework to Support Performance Optimization on Multiprocessors. *PDPTA'15: The 2015 International Conference on Parallel and Distributed Processing Techniques and Applications*, 2015.

[11] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[12] L. Yavits, A. Morad, and R. Ginosar. The effect of communication and synchronization on amdahl's law in multicore systems. *Parallel Comput.*, 40(1):1–16, January 2014.

[13] John L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31:532–533, 1988.

[14] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33(5):539–543, May 1990.

[15] Leonard J. Kazmier. *Schaum's outline of theory and problems of business statistics /.* McGraw-Hill,, New York :, c1976. Includes index.

[16] Friedrich Pukelsheim. The three sigma rule. *The American Statistician*, 48(2):88–91, 1994.

[17] Philip J. Fleming and John J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.

[18] Kendall Atkinson. *An Introduction to Numerical Analysis.* Wiley, 2 edition, 1989.

[19] Brendan Gregg. Linux Performance Analysis and Tools. Technical report, Joyent, February 2013.

[20] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec 1995.

[21] Ulrich Drepper. What Every Programmer Should Know About Memory. Technical report, Red Hat, November 2007.

[22] J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart. *LINPACK User's Guide.* SIAM, 1979.

[23] A. Petitet, R. C. Whaley, Jack Dongarra, and A. Cleary. HPL - a portable implementation of the High-Performance linpack benchmark for Distributed-Memory computers. Technical report, University of Tennessee, http://www.netlib.org/benchmark/hpl/, 2008.

[24] Message Passing Interface Forum. MPI: A Message-Passing interface standard. Technical report, University of Tennessee, May 1994.

[25] Piotr Luszczek, Jack J. Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John Mccalpin, David Bailey, and Daisuke Takahashi. Introduction to the HPC Challenge Benchmark Suite. Technical report, ., 2005.

[26] Martin Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game 'life'. *Scientific American*, pages 120–123, October 1970.

[27] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2002.

[28] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-018 in 1. Intel Press, March 2009.

[29] Connie U. Smith. Introduction to software performance engineering: origins and outstanding problems. In *Proceedings of the 7th international conference on Formal methods for performance evaluation*, SFM'07, pages 395–428, Berlin, Heidelberg, 2007. Springer-Verlag.

[30] J. C. Browne. A critical overview of computer performance evaluation. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 138–145, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[31] Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society.

[32] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[33] Kevin A. Huck, Oscar Hernandez, Van Bui, Sunita Chandrasekaran, Barbara Chapman, Allen D. Malony, Lois Curfman McInnes, and Boyana Norris. Capturing performance knowledge for automated analysis. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 49:1–49:10, Piscataway, NJ, USA, 2008. IEEE Press.

[34] Tomàs Margalef, Josep Jorba, Oleg Morajko, Anna Morajko, and Emilio Luque. Performance analysis and grid computing. In Vladimir Getov, Michael Gerndt, Adolfy Hoisie, Allen Malony, and Barton Miller, editors, *Performance analysis and grid computing*, chapter Different approaches to automatic performance analysis of distributed applications, pages 3–19. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[35] Felix Wolf and Bernd Mohr. Automatic performance analysis of hybrid mpi/openmp applications. *J. Syst. Archit.*, 49(10-11):421–439, November 2003.

[36] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[37] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science and Engg.*, 13(2):22–30, March 2011.

[38] Benoît B. Mandelbrot and Dann E. Passoja, editors. *Fractal aspects of materials: metal and catalyst surfaces, powders and aggregates: extended abstracts*, volume EA–4 of *Materials Research Society extended abstracts*, Pittsburgh, PA, USA, 1984. Materials Research Society.

[39] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[40] Fernando G. Tinetti. Cómputo Paralelo en Redes Locales de Computadoras. Technical report, Universidad Nacional de La Plata, 2004.

Apéndice A

Salida de Ejemplo

Listado 27: Salida de Ejemplo

```
20150330-170552: DEBUG: Logging to /home/amore/.hotspot/matrix/20150330-170552
20150330-170552: INFO: Starting hotspot
20150330-170552: INFO: Creating hardware section
20150330-170553: DEBUG: Run which lshw >/dev/null && lshw -short -sanitize
processor    Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
...
20150330-170553: DEBUG: Returning tags from hardware section
20150330-170553: INFO: Creating program section
20150330-170553: DEBUG: Returning tags from program section
20150330-170553: INFO: Creating software section
20150330-170553: DEBUG: Run which gcc >/dev/null && gcc --version for gcc (Ubuntu
     4.8.2-19ubuntu1) 4.8.2
Copyright (C) 2013 Free Software Foundation, Inc.
...
20150330-170553: DEBUG: Dumping /home/amore/.hotspot/matrix/software.1.output
20150330-170553: DEBUG: Run /lib/x86_64-linux-gnu/libc.so.6 for GNU C Library
Copyright (C) 2014 Free Software Foundation, Inc.
...
20150330-170553: DEBUG: Returning tags from software section
20150330-170553: INFO: Creating sanity section
20150330-170553: DEBUG: Dumping /home/amore/.hotspot/matrix/sanity.0.output
20150330-170558: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix && CFLAGS='
     -O3' make
 in 4.91553783417
20150330-170558: DEBUG: Returning tags from sanity section
20150330-170558: INFO: Creating footprint section
20150330-170558: DEBUG: Dumping /home/amore/.hotspot/matrix/footprint.0.output
20150330-170558: DEBUG: Run make clean && CFLAGS='-O3_-g' make for  in
     0.0740978717804
20150330-170558: DEBUG: Run file matrix for matrix: ELF 64-bit LSB  executable, x86
     -64,
 in 0.00443410873413
20150330-170558: DEBUG: Run which pahole >/dev/null && pahole matrix for struct
     _IO_FILE {
        int                        _flags;           /*     0     4 */
...
20150330-170558: DEBUG: Footprint report completed
20150330-170558: DEBUG: Returning tags from footprint section
20150330-170558: INFO: Creating benchmark section
20150330-170734: DEBUG: Run which mpirun >/dev/null && which hpcc >/dev/null &&
     mpirun -np 2 `which hpcc`
>>> cannot open file hpccinf.txt <<<
...
20150330-170734: DEBUG: System baseline completed
20150330-170734: DEBUG: Returning tags from benchmark section
20150330-170734: INFO: Creating workload section
20150330-170738: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix &&
     OMP_NUM_THREADS=2 N=1024 ./matrix
 in 4.35692501068
20150330-170738: INFO: Control 0 took 4.36 seconds
20150330-170743: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix &&
     OMP_NUM_THREADS=2 N=1024 ./matrix
 in 5.14309000969
20150330-170743: INFO: Control 1 took 5.14 seconds
20150330-170748: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix &&
     OMP_NUM_THREADS=2 N=1024 ./matrix
 in 4.63894510269
20150330-170748: INFO: Control 2 took 4.64 seconds
20150330-170752: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix &&
     OMP_NUM_THREADS=2 N=1024 ./matrix
 in 4.29476809502
20150330-170752: INFO: Control 3 took 4.29 seconds
20150330-170757: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix &&
     OMP_NUM_THREADS=2 N=1024 ./matrix
 in 4.77419805527
20150330-170757: INFO: Control 4 took 4.77 seconds
```

```
20150330-170802: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix &&
    OMP_NUM_THREADS=2 N=1024 ./matrix
 in 5.09612584114
20150330-170802: INFO: Control 5 took 5.10 seconds
20150330-170807: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix &&
    OMP_NUM_THREADS=2 N=1024 ./matrix
 in 4.48241996765
20150330-170807: INFO: Control 6 took 4.48 seconds
20150330-170812: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix &&
    OMP_NUM_THREADS=2 N=1024 ./matrix
 in 5.05654096603
20150330-170812: INFO: Control 7 took 5.06 seconds
20150330-170812: INFO: Deviation: gmean 4.72 std 0.32
20150330-170812: DEBUG: Plotted histogram
20150330-170812: DEBUG: Returning tags from workload section
20150330-170812: INFO: Creating scaling section
20150330-170812: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix; make clean
    ; CFLAGS='-O3 -Wall -Wextra' make for  in 0.0691559314728
20150330-170817: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix &&
    OMP_NUM_THREADS=2 N=1024 ./matrix
 in 4.2927801609
20150330-170817: INFO: Problem at 1024 took 4.29 seconds
20150330-170821: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix &&
    OMP_NUM_THREADS=2 N=1280 ./matrix
 in 4.30518698692
20150330-170821: INFO: Problem at 1280 took 4.31 seconds
20150330-170831: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix &&
    OMP_NUM_THREADS=2 N=1536 ./matrix
 in 10.3319211006
20150330-170831: INFO: Problem at 1536 took 10.33 seconds
20150330-170842: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix &&
    OMP_NUM_THREADS=2 N=1792 ./matrix
 in 10.7062511444
20150330-170842: INFO: Problem at 1792 took 10.71 seconds
20150330-170932: DEBUG: Run cd /home/amore/hotspot/tests/examples/matrix &&
    OMP_NUM_THREADS=2 N=2048 ./matrix
 in 50.0601859093
20150330-170932: INFO: Problem at 2048 took 50.06 seconds
20150330-170932: DEBUG: Plotted problem scaling
20150330-170932: DEBUG: Returning tags from scaling section
20150330-170932: INFO: Creating threading section
20150330-171042: DEBUG: Run OMP_NUM_THREADS=1 N=2048 ./matrix for  in 69.9276690483
20150330-171042: INFO: Threads at 1 took 69.93 seconds
20150330-171042: DEBUG: Dumping /home/amore/.hotspot/matrix/threading.1.output
20150330-171132: DEBUG: Run OMP_NUM_THREADS=2 N=2048 ./matrix for  in 50.0171101093
20150330-171132: INFO: Threads at 2 took 50.02 seconds
20150330-171132: DEBUG: Computed scaling laws
20150330-171132: DEBUG: Plotted thread scaling
20150330-171132: DEBUG: Returning tags from threading section
20150330-171132: INFO: Creating optimizations section
20150330-171749: DEBUG: Run CFLAGS='-O0' make && OMP_NUM_THREADS=2 N=2048 ./matrix
    for  in 376.509689093
20150330-171749: INFO: Optimizations at 0 took 376.51 seconds
20150330-172025: DEBUG: Run CFLAGS='-O1' make && OMP_NUM_THREADS=2 N=2048 ./matrix
    for  in 156.686037064
20150330-172025: INFO: Optimizations at 1 took 156.69 seconds
20150330-172129: DEBUG: Run CFLAGS='-O2' make && OMP_NUM_THREADS=2 N=2048 ./matrix
    for  in 64.02125597
20150330-172129: INFO: Optimizations at 2 took 64.02 seconds
20150330-172220: DEBUG: Run CFLAGS='-O3' make && OMP_NUM_THREADS=2 N=2048 ./matrix
    for  in 50.6308100224
20150330-172220: INFO: Optimizations at 3 took 50.63 seconds
20150330-172220: DEBUG: Plotted optimizations
20150330-172220: DEBUG: Returning tags from optimizations section
20150330-172220: INFO: Creating profile section
20150330-172311: DEBUG: Run CFLAGS='-O3 -g -pg' make && OMP_NUM_THREADS=2 N=2048 ./
    matrix && which gprof >/dev/null && gprof -l -b matrix
Each sample counts as 0.01 seconds.
...
20150330-172311: DEBUG: Profiling report completed
20150330-172311: DEBUG: Returning tags from profile section
20150330-172311: INFO: Creating resources section
20150330-172403: DEBUG: Run OMP_NUM_THREADS=2 N=2048 ./matrix& which pidstat >/dev/
    null && pidstat -s -r -d -u -h -p $! 1
1427761393,1000,20581,199.00,0.00,0.00,199.00,0,0.00,0.00,65968,50424
...
20150330-172403: DEBUG: Resource usage plotting completed
20150330-172403: DEBUG: Returning tags from resources section
20150330-172403: INFO: Creating annotated section
20150330-172540: DEBUG: Run CFLAGS='-O3 -g' make && which perf >/dev/null && echo "
    OMP_NUM_THREADS=2 N=2048 perf record -q -- ./matrix"
...
20150330-172540: DEBUG: Source annotation completed
20150330-172540: DEBUG: Returning tags from annotated section
20150330-172540: INFO: Creating vectorization section
20150330-172540: DEBUG: Run CFLAGS='-O3 -ftree-vectorizer-verbose=2' make 2>&1 |
    grep -v "^$" for Analyzing loop at matrix.c:26
matrix.c:26: note: not vectorized: multiple nested loops.
...
20150330-172540: DEBUG: Vectorization report completed
20150330-172540: DEBUG: Returning tags from vectorization section
20150330-172540: INFO: Creating counters section
20150330-172924: DEBUG: Run which perf >/dev/null && N=2048 perf stat -r 3 ./matrix
    for
 Performance counter stats for './matrix' (3 runs):
...
20150330-172924: DEBUG: Hardware counters gathering completed
20150330-172924: DEBUG: Showing counters section
matrix.c:26: note: not vectorized: multiple nested loops.
```

```
...
20150330-172924: DEBUG: Returning tags from counters section
20150330-172924: INFO: Generating report
20150330-172924: DEBUG: Replacing macro amdalah with 2.32267
20150330-172924: DEBUG: Replacing macro annotation with          :       #pragma omp parallel for shared(a,b,c)
          :         for (i = 0; i < size; ++i) {
...
20150330-172924: DEBUG: Replacing macro average with 4.73038
20150330-172924: DEBUG: Replacing macro build with CFLAGS='{0}' make
20150330-172924: DEBUG: Replacing macro cflags with -O3 -Wall -Wextra
20150330-172924: DEBUG: Replacing macro clean with make clean
20150330-172924: DEBUG: Replacing macro compiler with gcc (Ubuntu 4.8.2-19ubuntu1) 4.8.2
20150330-172924: DEBUG: Replacing macro cores with 2
20150330-172924: DEBUG: Replacing macro count with 8
20150330-172924: DEBUG: Replacing macro counters with
 Performance counter stats for './matrix' (3 runs):
...
20150330-172924: DEBUG: Replacing macro cwd with /home/amore/hotspot/tests/examples/matrix
20150330-172924: DEBUG: Replacing macro distro with Ubuntu, 14.04, trusty
20150330-172924: DEBUG: Replacing macro first with 1024
20150330-172924: DEBUG: Replacing macro geomean with 4.71964
20150330-172924: DEBUG: Replacing macro gustafson with 583.55915
20150330-172924: DEBUG: Replacing macro hardware with memory       3953MiB System memory
processor       Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
...
20150330-172924: DEBUG: Replacing macro host with ubuntu
20150330-172924: DEBUG: Replacing macro hpcc-dgemm with 1.79842 GFlops
20150330-172924: DEBUG: Replacing macro hpcc-fft with 2.03598 GFlops
20150330-172924: DEBUG: Replacing macro hpcc-hpl with 0.00385463 TFlops
20150330-172924: DEBUG: Replacing macro hpcc-ptrans with 1.8496 GBs
20150330-172924: DEBUG: Replacing macro hpcc-random with 0.0523812 GUPs
20150330-172924: DEBUG: Replacing macro hpcc-stream with 6.96329 MBs
20150330-172924: DEBUG: Replacing macro hpcc-success with 1 None
20150330-172924: DEBUG: Replacing macro increment with 256
20150330-172924: DEBUG: Replacing macro last with 2048
20150330-172924: DEBUG: Replacing macro libc with GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6.6) stable release version 2.19
20150330-172924: DEBUG: Replacing macro log with /home/amore/.hotspot/matrix/20150330-170552
20150330-172924: DEBUG: Replacing macro max with 5.14309
20150330-172924: DEBUG: Replacing macro min with 4.29477
20150330-172924: DEBUG: Replacing macro pahole with struct _IO_FILE {
        int                        _flags;               /*     0     4 */
...
20150330-172924: DEBUG: Replacing macro parallel with 0.56946
20150330-172924: DEBUG: Replacing macro platform with Linux-3.16.0-30-generic-x86_64-with-Ubuntu-14.04-trusty
20150330-172924: DEBUG: Replacing macro profile with Flat profile:
Each sample counts as 0.01 seconds.
...
20150330-172924: DEBUG: Replacing macro program with matrix
20150330-172924: DEBUG: Replacing macro range with [1024, 1280, 1536, 1792]
20150330-172924: DEBUG: Replacing macro resources with 1427761392,1000,20581,184.00,1.00,0.00,185.00,0,914.00,0.00,65968,50424,1.25
...
20150330-172924: DEBUG: Replacing macro run with OMP_NUM_THREADS={0} N={1} ./{2}
20150330-172924: DEBUG: Replacing macro serial with 0.43054
20150330-172924: DEBUG: Replacing macro stddev with 0.31849
20150330-172924: DEBUG: Replacing macro strip with matrix: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs)
20150330-172924: DEBUG: Replacing macro timestamp with 20150330-170552
20150330-172924: DEBUG: Replacing macro vectorizer with Analyzing loop at matrix.c:26
matrix.c:26: note: not vectorized: multiple nested loops.
...
20150330-172925: INFO: Completed execution in 1412.88 seconds
```

Apéndice B

Reportes de Ejemplo

B.1. Multiplicación de Matrices

`matrix` Performance Report

20150608-190308

Abstract

This performance report is intended to support performance analysis and optimization activities. It includes details on program behavior, system configuration and capabilities, including support data from well-known performance analysis tools.

This report was generated using `hotspot` version `0.1`. Homepage `http://www.github.com/moreandres/hotspot`. Full execution log can be found at `~/.hotspot/matrix/20150608-190308/hotspot.log`.

Contents

1 Program 1
2 System Capabilities 1
2.1 System Configuration . . . 1
2.2 System Performance Baseline . . . 2
3 Workload 2
3.1 Workload Footprint . . . 2
3.2 Workload Stability . . . 3
3.3 Workload Optimization . . . 4
4 Scalability 4
4.1 Problem Size Scalability . . . 4
4.2 Computing Scalability . . . 5
5 Profile 5
5.1 Program Profiling . . . 5
5.1.1 Flat Profile . . . 6
5.2 System Profiling . . . 6
5.2.1 System Resources Usage . . . 6
5.3 Hotspots . . . 7
6 Low Level 8
6.1 Vectorization Report . . . 8
6.2 Counters Report . . . 9

1 Program

This section provides details about the program being analyzed.

1. Program: `matrix`.
   Program is the name of the program.
2. Timestamp: `20150608-190308`.
   Timestamp is a unique identifier used to store information on disk.
3. Parameters Range: `[2048, 2304, 2560, 2816, 3072, 3328, 3584, 3840]`.
   Parameters range is the problem size set used to scale the program.

2 System Capabilities

This section provides details about the system being used for the analysis.

2.1 System Configuration

This subsection provides details about the system configuration.

The hardware in the system is summarized using a hardware lister utility. It reports exact memory configuration, firmware version, mainboard configuration, CPU version and speed, cache configuration, bus speed and others.

The hardware configuration can be used to contrast the system capabilities to well-known benchmarks results on similar systems.

```
memory      7984MiB System memory
processor   Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
bridge      440FX - 82441FX PMC [Natoma]
bridge      82371SB PIIX3 ISA [Natoma/Triton II]
storage     82371AB/EB/MB PIIX4 IDE
network     82540EM Gigabit Ethernet Controller
bridge      82371AB/EB/MB PIIX4 ACPI
storage     82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode]
```

The software in the system is summarized using the GNU/Linux platform string.

```
Linux-3.16.0-30-generic-x86_64-with-Ubuntu-14.04-trusty
```

The software toolchain is built upon the following components.

1. Host: `ubuntu`

2. Distribution: `Ubuntu, 14.04, trusty`.
   This codename provides LSB (Linux Standard Base) and distribution-specific information.
3. Compiler: `gcc (Ubuntu 4.8.2-19ubuntu1) 4.8.2`.
   Version number of the compiler program.
4. C Library: `GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6.6) stable release version 2.19`.
   Version number of the C library.

The software configuration can be used to contrast the system capabilities to well-known benchmark results on similar systems.

2.2 System Performance Baseline

This subsection provides details about the system capabilities.

A set of performance results is included as a reference to contrast systems and to verify hardware capabilities using well-known synthetic benchmarks.

The HPC Challenge benchmark [1] consists of different tests:

1. HPL: the Linpack TPP benchmark which measures the floating point rate of execution for solving a linear system of equations.
2. DGEMM: measures the floating point rate of execution of double precision real matrix-matrix multiplication.
3. PTRANS (parallel matrix transpose): exercises the communications where pairs of processors communicate with each other simultaneously.
4. RandomAccess: measures the rate of integer random updates of memory (GUPS).
5. STREAM: a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s).
6. FFT: measures the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT).

Table 1: Benchmarks

Benchmark	Value	Unit
hpl	0.00385463 TFlops	tflops
dgemm	1.79842 GFlops	mflops
ptrans	1.8496 GBs	MB/s
random	0.0523812 GUPs	MB/s
stream	6.96329 MBs	MB/s
fft	2.03598 GFlops	MB/s

Most programs will have a dominant compute kernel that can be approximated by the ones above, the results helps to understand the available capacity.

3 Workload

This section provides details about the workload behavior.

3.1 Workload Footprint

The workload footprint impacts on memory hierarchy usage.

```
matrix: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux
```

Binaries should be stripped to better fit inside cache.

```
struct _IO_FILE {
int                        _flags;              /*     0     4 */

/* XXX 4 bytes hole, try to pack */

char *                     _IO_read_ptr;        /*     8     8 */
char *                     _IO_read_end;        /*    16     8 */
char *                     _IO_read_base;       /*    24     8 */
char *                     _IO_write_base;      /*    32     8 */
char *                     _IO_write_ptr;       /*    40     8 */
char *                     _IO_write_end;       /*    48     8 */
char *                     _IO_buf_base;        /*    56     8 */
/* --- cacheline 1 boundary (64 bytes) --- */
char *                     _IO_buf_end;         /*    64     8 */
char *                     _IO_save_base;       /*    72     8 */
char *                     _IO_backup_base;     /*    80     8 */
```

```
char *                    _IO_save_end;         /*    88     8 */
struct _IO_marker *       _markers;             /*    96     8 */
struct _IO_FILE *         _chain;               /*   104     8 */
int                       _fileno;              /*   112     4 */
int                       _flags2;              /*   116     4 */
__off_t                   _old_offset;          /*   120     8 */
/* --- cacheline 2 boundary (128 bytes) --- */
short unsigned int        _cur_column;          /*   128     2 */
signed char               _vtable_offset;       /*   130     1 */
char                      _shortbuf[1];         /*   131     1 */

/* XXX 4 bytes hole, try to pack */

_IO_lock_t *              _lock;                /*   136     8 */
__off64_t                 _offset;              /*   144     8 */
void *                    __pad1;               /*   152     8 */
void *                    __pad2;               /*   160     8 */
void *                    __pad3;               /*   168     8 */
void *                    __pad4;               /*   176     8 */
size_t                    __pad5;               /*   184     8 */
/* --- cacheline 3 boundary (192 bytes) --- */
int                       _mode;                /*   192     4 */
char                      _unused2[20];         /*   196    20 */

/* size: 216, cachelines: 4, members: 29 */
/* sum members: 208, holes: 2, sum holes: 8 */
/* last cacheline: 24 bytes */
};
struct _IO_marker {
struct _IO_marker *       _next;                /*     0     8 */
struct _IO_FILE *         _sbuf;                /*     8     8 */
int                       _pos;                 /*    16     4 */

/* size: 24, cachelines: 1, members: 3 */
/* padding: 4 */
/* last cacheline: 24 bytes */
};
```

The in-memory layout of data structures can be used to identify issues. Reorganizing data to remove alignment holes will improve CPU cache utilization.

More information `https://www.kernel.org/doc/ols/2007/ols2007v2-pages-35-44.pdf`

3.2 Workload Stability

This subsection provides details about workload stability.

1. Execution time:
   (a) problem size range: `2048 - 4096`
   (b) geomean: `42.80017` seconds
   (c) average: `42.80048` seconds
   (d) stddev: `0.16483`
   (e) min: `42.41161` seconds
   (f) max: `43.15019` seconds
   (g) repetitions: `16` times

   The histogram plots the elapsed times and shows how they fit in a normal distribution sample.

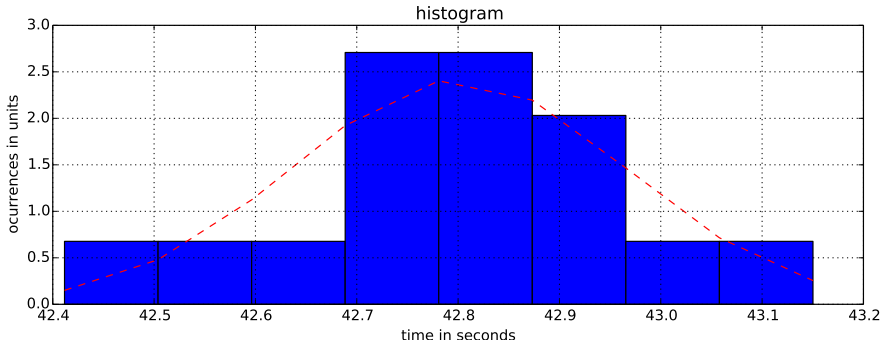

Figure 1: Results Distribution

The workload should run for at least one minute to fully utilize system resources. The execution time of the workload should be stable and the standard deviation less than 3 units.

3.3 Workload Optimization

This section shows how the program reacts to different optimization levels.

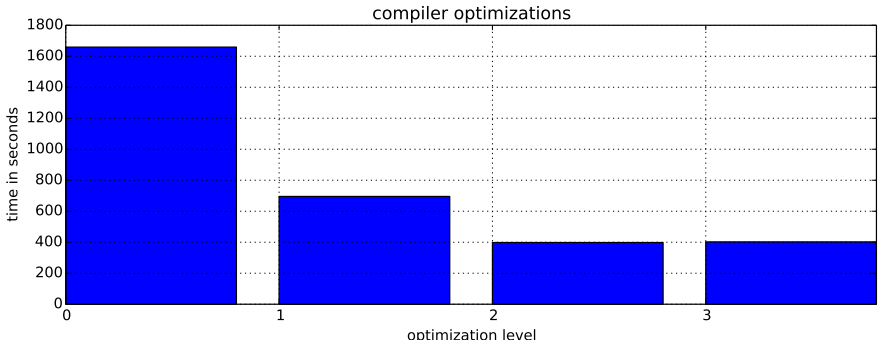

Figure 2: Optimization Levels

4 Scalability

This section provides details about the scaling behavior of the program.

4.1 Problem Size Scalability

A chart with the execution time when scaling the problem size.

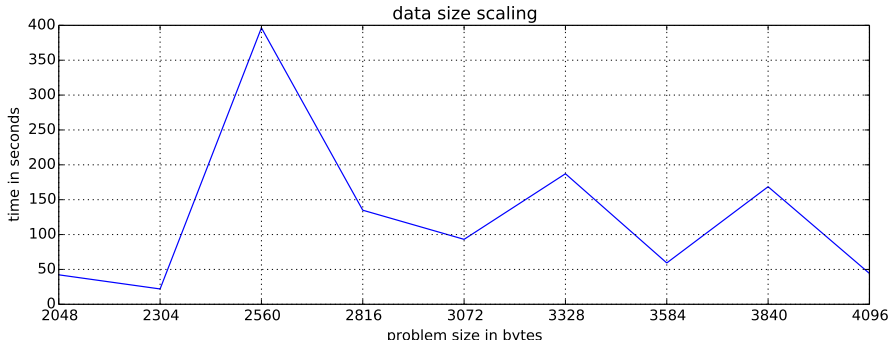

Figure 3: Problem size times

The chart will show how computing time increases when increasing problem size. There should be no valleys or bumps if processing properly balanced across computational units.

4.2 Computing Scalability

A chart with the execution time when scaling computation units.

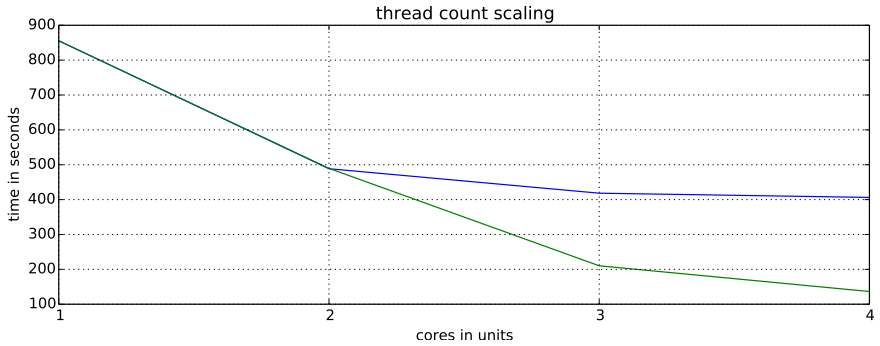

Figure 4: Thread count times

The chart will show how computing time decreases when increasing processing units. An ideal scaling line is provided for comparison.

The parallel and serial fractions of the program can be estimated using the information above.

1. Parallel Fraction: `0.85788`.
   Portion of the program doing parallel work.
2. Serial: `0.14212`.
   Portion of the program doing serial work.

Optimization limits can be estimated using scaling laws.

1. Amdalah Law for 1024 procs: `7.03614 times`.
   Optimizations are limited up to this point when scaling problem size. [2]
2. Gustafson Law for 1024 procs: `878.60771 times`.
   Optimizations are limited up to this point when not scaling problem size. [3]

5 Profile

This section provides details about the execution profile of the program and the system.

5.1 Program Profiling

This subsection provides details about the program execution profile.

5.1.1 Flat Profile

The flat profile shows how much time your program spent in each function, and how many times that function was called.

```
Flat profile:
Each sample counts as 0.01 seconds.
     cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 90.00    659.87   659.87                             main._omp_fn.0 (matrix.c:28 @ 400b48)
  9.96    732.90    73.03                             main._omp_fn.0 (matrix.c:28 @ 400b5b)
```

The table shows where to focus optimization efforts to maximize impact.

5.2 System Profiling

This subsection provide details about the system execution profile.

5.2.1 System Resources Usage

The following charts shows the state of system resources during the execution of the program.

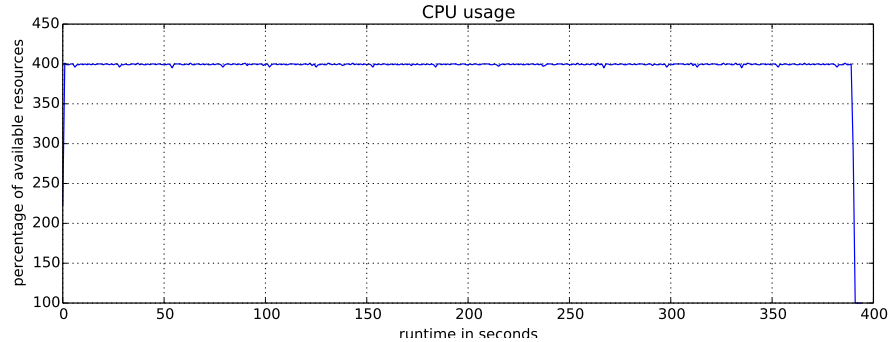

Figure 5: CPU Usage

Note that this chart is likely to show as upper limit a multiple of 100% in case a multicore system is being used.

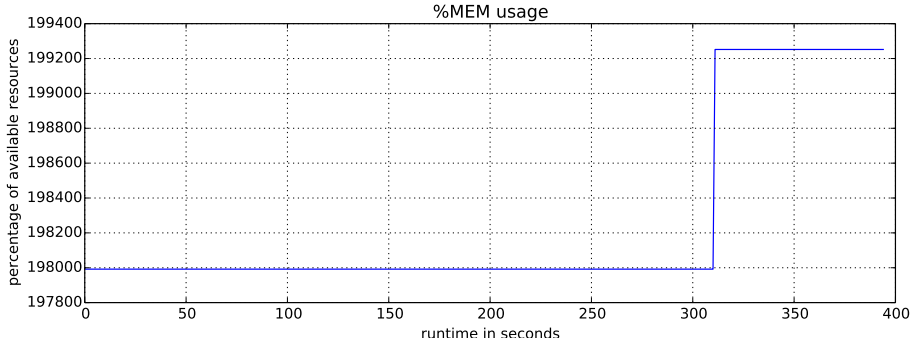

Figure 6: Memory Usage

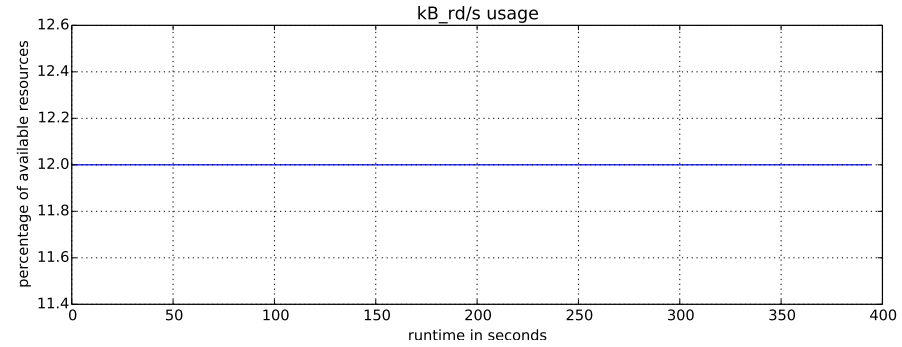

Figure 7: Reads from Disk

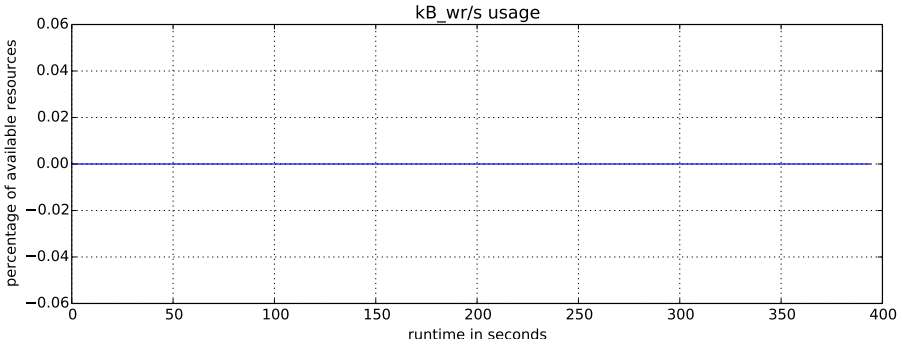

Figure 8: Writes to Disk

5.3 Hotspots

This subsection shows annotated code guiding the optimization efforts.

```
         : #pragma omp parallel for shared(a,b,c)
         :   for (i = 0; i < size; ++i) {
         :     for (j = 0; j < size; ++j) {
         :       for (k = 0; k < size; ++k) {
         :         c[i+j*size] += a[i+k*size] * b[k+j*size];
    0.20 :   4009e8:       movss  (rcx),xmm0
   77.69 :   4009ec:       add    r9,rcx
    9.81 :   4009ef:       mulss  (r8,rdx,4),xmm0
    6.46 :   4009f5:       add    $0x1,rdx
         :   }
         : #pragma omp parallel for shared(a,b,c)
         :   for (i = 0; i < size; ++i) {
         :     for (j = 0; j < size; ++j) {
         :       for (k = 0; k < size; ++k) {
         :         c[i+j*size] += a[i+k*size] * b[k+j*size];
    0.91 :   4009fb:       addss  xmm0,xmm1
    3.52 :   4009ff:       movss  xmm1,(rsi)
         :   }
         : #pragma omp parallel for shared(a,b,c)
         :   for (i = 0; i < size; ++i) {
         :     for (j = 0; j < size; ++j) {
         :       for (k = 0; k < size; ++k) {
--
         :       b[i+j*size] = (float) (i - j);
         :   int i, j, k;
         :   for (i = 0; i < size; ++i) {
         :     for (j = 0; j < size; ++j) {
```

```
       :       a[i+j*size] = (float) (i + j);
  64.82 :   4007c3:       cvtsi2ss edi,xmm0
       :   float *c = malloc(sizeof(float) * size * size);
       :   int i, j, k;
       :   for (i = 0; i < size; ++i) {
       :     for (j = 0; j < size; ++j) {
       :       a[i+j*size] = (float) (i + j);
   2.24 :   4007ce:       movss  xmm0,(r10,rcx,1)
       :       b[i+j*size] = (float) (i - j);
  29.06 :   4007d4:       cvtsi2ss edi,xmm0
   1.85 :   4007d8:       movss  xmm0,(r9,rcx,1)
   0.97 :   4007de:       add    r11,rcx
       :   float *c = malloc(sizeof(float) * size * size);
       :   int i, j, k;
       :   for (i = 0; i < size; ++i) {
       :     for (j = 0; j < size; ++j) {
       :   float *b = malloc(sizeof(float) * size * size);
```

6 Low Level

This section provide details about low level details such as vectorization and performance counters.

6.1 Vectorization Report

This subsection provide details about vectorization status of the program loops.

```
Analyzing loop at matrix.c:26
matrix.c:26: note: not vectorized: multiple nested loops.
matrix.c:26: note: bad loop form.
Analyzing loop at matrix.c:26
matrix.c:26: note: not vectorized: not suitable for strided load _41 = *_40;
matrix.c:26: note: bad data references.
Analyzing loop at matrix.c:27
matrix.c:27: note: step unknown.
matrix.c:27: note: reduction used in loop.
matrix.c:27: note: Unknown def-use cycle pattern.
matrix.c:27: note: Unsupported pattern.
matrix.c:27: note: not vectorized: unsupported use in stmt.
matrix.c:27: note: unexpected pattern.
matrix.c:24: note: vectorized 0 loops in function.
matrix.c:24: note: not consecutive access _10 = .omp_data_i_9(D)->size;
matrix.c:24: note: Failed to SLP the basic block.
matrix.c:24: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
matrix.c:26: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: not consecutive access .omp_data_i_9(D)->j = .omp_data_i__j_lsm.9_106;
matrix.c:24: note: Failed to SLP the basic block.
matrix.c:24: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: Failed to SLP the basic block.
matrix.c:24: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:24: note: not consecutive access pretmp_123 = *pretmp_122;
matrix.c:24: note: Failed to SLP the basic block.
matrix.c:24: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:26: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: not consecutive access .omp_data_i_9(D)->k = _10;
matrix.c:24: note: Failed to SLP the basic block.
matrix.c:24: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
matrix.c:28: note: can't determine dependence between *_40 and *pretmp_122
matrix.c:28: note: can't determine dependence between *_46 and *pretmp_122
matrix.c:28: note: SLP: step doesn't divide the vector-size.
matrix.c:28: note: Unknown alignment for access: *(pretmp_113 + (sizetype) ((long unsigned int) pretmp_118 * 4))
matrix.c:28: note: not consecutive access _41 = *_40;
matrix.c:28: note: not consecutive access *pretmp_122 = _49;
matrix.c:28: note: Failed to SLP the basic block.
```

```
matrix.c:28: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: not vectorized: not enough data-refs in basic block.
Analyzing loop at matrix.c:16
matrix.c:16: note: not vectorized: not suitable for strided load *_27 = _29;
matrix.c:16: note: bad data references.
Analyzing loop at matrix.c:17
matrix.c:17: note: not vectorized: not suitable for strided load *_27 = _29;
matrix.c:17: note: bad data references.
matrix.c:4: note: vectorized 0 loops in function.
matrix.c:7: note: not vectorized: not enough data-refs in basic block.
matrix.c:8: note: not vectorized: not enough data-refs in basic block.
matrix.c:4: note: not vectorized: not enough data-refs in basic block.
matrix.c:18: note: not consecutive access *_27 = _29;
matrix.c:18: note: not consecutive access *_32 = _34;
matrix.c:18: note: not consecutive access *_36 = 0.0;
matrix.c:18: note: Failed to SLP the basic block.
matrix.c:18: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:16: note: not vectorized: not enough data-refs in basic block.
matrix.c:4: note: not vectorized: not enough data-refs in basic block.
matrix.c:4: note: not vectorized: not enough data-refs in basic block.
matrix.c:24: note: misalign = 0 bytes of ref .omp_data_o.1.c
matrix.c:24: note: misalign = 8 bytes of ref .omp_data_o.1.b
matrix.c:24: note: misalign = 0 bytes of ref .omp_data_o.1.a
matrix.c:24: note: misalign = 8 bytes of ref .omp_data_o.1.size
matrix.c:24: note: misalign = 12 bytes of ref .omp_data_o.1.j
matrix.c:24: note: misalign = 0 bytes of ref .omp_data_o.1.k
matrix.c:24: note: Build SLP failed: unrolling required in basic block SLP
matrix.c:24: note: Build SLP failed: unrolling required in basic block SLP
matrix.c:24: note: Failed to SLP the basic block.
matrix.c:24: note: not vectorized: failed to find SLP opportunities in basic block.
matrix.c:4: note: not vectorized: not enough data-refs in basic block.
matrix.c:10: note: not vectorized: not enough data-refs in basic block.
matrix.c:4: note: not vectorized: not enough data-refs in basic block.
```

The details above shows the list of loops in the program and if they are being vectorized or not. These reports can pinpoint areas where the compiler cannot apply vectorization and related optimizations. It may be possible to modify your code or communicate additional information to the compiler to guide the vectorization and/or optimizations.

6.2 Counters Report

This subsection provides details about software and hardware counters.

```
 Performance counter stats for './matrix' (3 runs):

    1568409.582702      task-clock (msec)         #    3.954 CPUs utilized            ( +-  0.16 )
            2,863      context-switches          #    0.002 K/sec                    ( +-  0.80 )
                3      cpu-migrations            #    0.000 K/sec                    ( +- 10.00 )
            1,703      page-faults               #    0.001 K/sec                    ( +-  0.04 )
   <not supported>      cycles
   <not supported>      stalled-cycles-frontend
   <not supported>      stalled-cycles-backend
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

     396.643251906 seconds time elapsed                                          ( +-  0.31 )
```

The details above shows counters that provide low-overhead access to detailed performance information using internal registers of the CPU.

References

[1] Piotr Luszczek and Jack J. Dongarra and David Koester and Rolf Rabenseifner and Bob Lucas and Jeremy Kepner and John Mccalpin and David Bailey and Daisuke Takahashi, *Introduction to the HPC Challenge Benchmark Suite*. Technical Report, 2005.

[2] Amdahl, Gene M., *Validity of the single processor approach to achieving large scale computing capabilities.* Communications of the ACM, Proceedings of the April 18-20, 1967, spring joint computer conference Pages 483-485, 1967.

[3] John L. Gustafson, *Reevaluating Amdahl's Law.* Communications of the ACM, Volume 31 Pages 532-533, 1988.

[4] OpenMP Architecture Review Board, *OpenMP Application Program Interface. http://www.openmp.org*, 3.0, May 2008.

B.2. Propagación de Calor en 2 Dimensiones

`heat2d` Performance Report

20150607-141823

Abstract

This performance report is intended to support performance analysis and optimization activities. It includes details on program behavior, system configuration and capabilities, including support data from well-known performance analysis tools.

This report was generated using `hotspot` version `0.1`. Homepage `http://www.github.com/moreandres/hotspot`. Full execution log can be found at `~/.hotspot/heat2d/20150607-141823/hotspot.log`.

Contents

1 Program **1**
2 System Capabilities **1**
2.1 System Configuration 1
2.2 System Performance Baseline 2
3 Workload **2**
3.1 Workload Footprint 2
3.2 Workload Stability 3
3.3 Workload Optimization 4
4 Scalability **4**
4.1 Problem Size Scalability 4
4.2 Computing Scalability 5
5 Profile **5**
5.1 Program Profiling 5
5.1.1 Flat Profile 6
5.2 System Profiling 6
5.2.1 System Resources Usage 6
5.3 Hotspots 7
6 Low Level **12**
6.1 Vectorization Report 12
6.2 Counters Report 14

1 Program

This section provides details about the program being analyzed.

1. Program: `heat2d`.
   Program is the name of the program.
2. Timestamp: `20150607-141823`.
   Timestamp is a unique identifier used to store information on disk.
3. Parameters Range: `[32768, 33792, 34816, 35840, 36864, 37888, 38912, 39936, 40960, 41984, 43008, 44032, 45056, 46080, 47104, 48128, 49152, 50176, 51200, 52224, 53248, 54272, 55296, 56320, 57344, 58368, 59392, 60416, 61440, 62464, 63488, 64512]`.
   Parameters range is the problem size set used to scale the program.

2 System Capabilities

This section provides details about the system being used for the analysis.

2.1 System Configuration

This subsection provides details about the system configuration.

The hardware in the system is summarized using a hardware lister utility. It reports exact memory configuration, firmware version, mainboard configuration, CPU version and speed, cache configuration, bus speed and others.

The hardware configuration can be used to contrast the system capabilities to well-known benchmarks results on similar systems.

```
memory      7984MiB System memory
processor   Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
bridge      440FX - 82441FX PMC [Natoma]
bridge      82371SB PIIX3 ISA [Natoma/Triton II]
storage     82371AB/EB/MB PIIX4 IDE
network     82540EM Gigabit Ethernet Controller
bridge      82371AB/EB/MB PIIX4 ACPI
storage     82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode]
```

The software in the system is summarized using the GNU/Linux platform string.

```
Linux-3.16.0-30-generic-x86_64-with-Ubuntu-14.04-trusty
```

The software toolchain is built upon the following components.

1. Host: `ubuntu`
2. Distribution: `Ubuntu, 14.04, trusty`.
   This codename provides LSB (Linux Standard Base) and distribution-specific information.
3. Compiler: `gcc (Ubuntu 4.8.2-19ubuntu1) 4.8.2`.
   Version number of the compiler program.
4. C Library: `GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6.6) stable release version 2.19`.
   Version number of the C library.

The software configuration can be used to contrast the system capabilities to well-known benchmark results on similar systems.

2.2 System Performance Baseline

This subsection provides details about the system capabilities.

A set of performance results is included as a reference to contrast systems and to verify hardware capabilities using well-known synthetic benchmarks.

The HPC Challenge benchmark [1] consists of different tests:

1. HPL: the Linpack TPP benchmark which measures the floating point rate of execution for solving a linear system of equations.
2. DGEMM: measures the floating point rate of execution of double precision real matrix-matrix multiplication.
3. PTRANS (parallel matrix transpose): exercises the communications where pairs of processors communicate with each other simultaneously.
4. RandomAccess: measures the rate of integer random updates of memory (GUPS).
5. STREAM: a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s).
6. FFT: measures the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT).

Table 1: Benchmarks

Benchmark	Value	Unit
hpl	0.00365811 TFlops	tflops
dgemm	1.72977 GFlops	mflops
ptrans	1.58619 GBs	MB/s
random	0.0544077 GUPs	MB/s
stream	7.00091 MBs	MB/s
fft	2.32577 GFlops	MB/s

Most programs will have a dominant compute kernel that can be approximated by the ones above, the results helps to understand the available capacity.

3 Workload

This section provides details about the workload behavior.

3.1 Workload Footprint

The workload footprint impacts on memory hierarchy usage.

```
heat2d: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux
```

Binaries should be stripped to better fit inside cache.

```
struct _IO_FILE {
int                        _flags;               /*     0     4 */

/* XXX 4 bytes hole, try to pack */

char *                     _IO_read_ptr;         /*     8     8 */
char *                     _IO_read_end;         /*    16     8 */
char *                     _IO_read_base;        /*    24     8 */
char *                     _IO_write_base;       /*    32     8 */
char *                     _IO_write_ptr;        /*    40     8 */
char *                     _IO_write_end;        /*    48     8 */
char *                     _IO_buf_base;         /*    56     8 */
/* --- cacheline 1 boundary (64 bytes) --- */
char *                     _IO_buf_end;          /*    64     8 */
```

```
char *                    _IO_save_base;        /*    72     8 */
char *                    _IO_backup_base;      /*    80     8 */
char *                    _IO_save_end;         /*    88     8 */
struct _IO_marker *       _markers;             /*    96     8 */
struct _IO_FILE *         _chain;               /*   104     8 */
int                       _fileno;              /*   112     4 */
int                       _flags2;              /*   116     4 */
__off_t                   _old_offset;          /*   120     8 */
/* --- cacheline 2 boundary (128 bytes) --- */
short unsigned int        _cur_column;          /*   128     2 */
signed char               _vtable_offset;       /*   130     1 */
char                      _shortbuf[1];         /*   131     1 */

/* XXX 4 bytes hole, try to pack */

_IO_lock_t *              _lock;                /*   136     8 */
__off64_t                 _offset;              /*   144     8 */
void *                    __pad1;               /*   152     8 */
void *                    __pad2;               /*   160     8 */
void *                    __pad3;               /*   168     8 */
void *                    __pad4;               /*   176     8 */
size_t                    __pad5;               /*   184     8 */
/* --- cacheline 3 boundary (192 bytes) --- */
int                       _mode;                /*   192     4 */
char                      _unused2[20];         /*   196    20 */

/* size: 216, cachelines: 4, members: 29 */
/* sum members: 208, holes: 2, sum holes: 8 */
/* last cacheline: 24 bytes */
};
struct _IO_marker {
struct _IO_marker *       _next;                /*     0     8 */
struct _IO_FILE *         _sbuf;                /*     8     8 */
int                       _pos;                 /*    16     4 */

/* size: 24, cachelines: 1, members: 3 */
/* padding: 4 */
/* last cacheline: 24 bytes */
};
```

The in-memory layout of data structures can be used to identify issues. Reorganizing data to remove alignment holes will improve CPU cache utilization.

More information `https://www.kernel.org/doc/ols/2007/ols2007v2-pages-35-44.pdf`

3.2 Workload Stability

This subsection provides details about workload stability.

1. Execution time:
   (a) problem size range: `32768 - 65536`
   (b) geomean: `23.32382` seconds
   (c) average: `23.32607` seconds
   (d) stddev: `0.32522`
   (e) min: `22.90810` seconds
   (f) max: `24.26887` seconds
   (g) repetitions: `32` times

   The histogram plots the elapsed times and shows how they fit in a normal distribution sample.

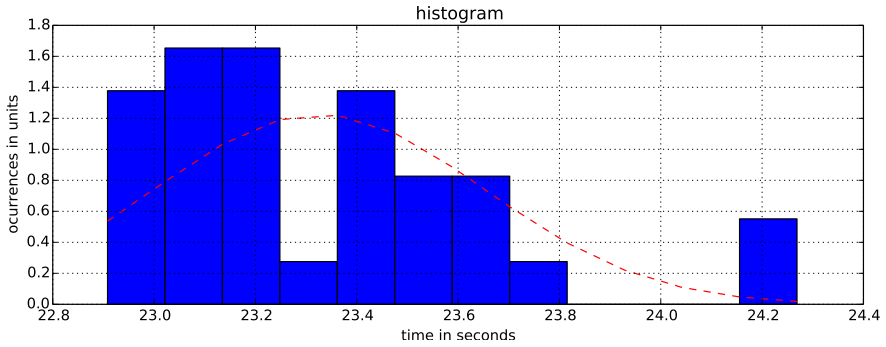

Figure 1: Results Distribution

The workload should run for at least one minute to fully utilize system resources. The execution time of the workload should be stable and the standard deviation less than 3 units.

3.3 Workload Optimization

This section shows how the program reacts to different optimization levels.

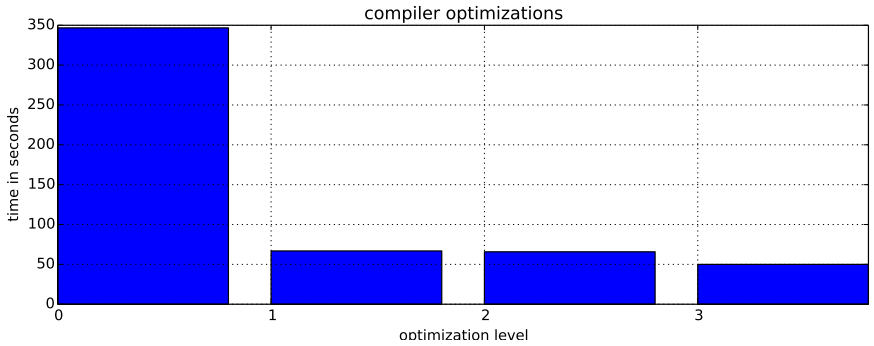

Figure 2: Optimization Levels

4 Scalability

This section provides details about the scaling behavior of the program.

4.1 Problem Size Scalability

A chart with the execution time when scaling the problem size.

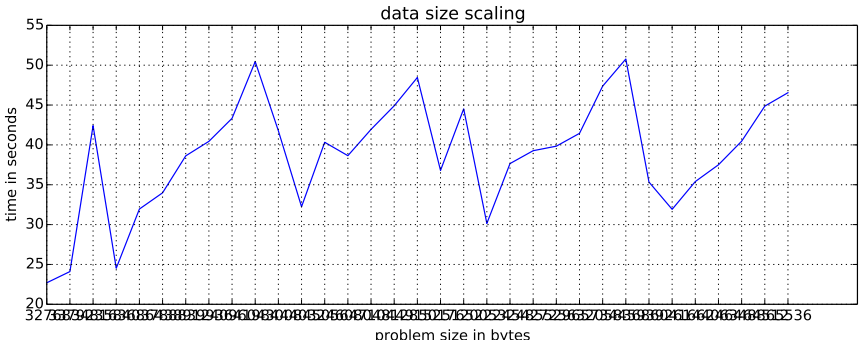

Figure 3: Problem size times

The chart will show how computing time increases when increasing problem size. There should be no valleys or bumps if processing properly balanced across computational units.

4.2 Computing Scalability

A chart with the execution time when scaling computation units.

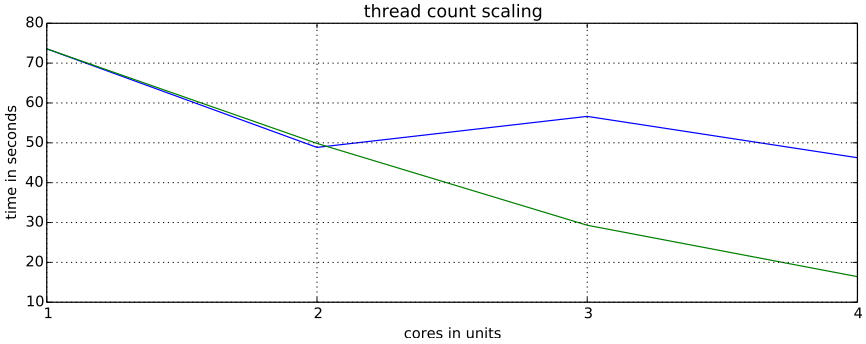

Figure 4: Thread count times

The chart will show how computing time decreases when increasing processing units. An ideal scaling line is provided for comparison.

The parallel and serial fractions of the program can be estimated using the information above.

1. Parallel Fraction: `0.67235`.
   Portion of the program doing parallel work.
2. Serial: `0.32765`.
   Portion of the program doing serial work.

Optimization limits can be estimated using scaling laws.

1. Amdalah Law for 1024 procs: `3.05200 times`.
   Optimizations are limited up to this point when scaling problem size. [2]
2. Gustafson Law for 1024 procs: `688.80974 times`.
   Optimizations are limited up to this point when not scaling problem size. [3]

5 Profile

This section provides details about the execution profile of the program and the system.

5.1 Program Profiling

This subsection provides details about the program execution profile.

5.1.1 Flat Profile

The flat profile shows how much time your program spent in each function, and how many times that function was called.

```
Flat profile:
Each sample counts as 0.01 seconds.
     cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 29.65     41.38    41.38                             compute_one_iteration._omp_fn.1 (heat2d.c:78 @ 400d97)
 12.89     59.37    17.99                             compute_one_iteration._omp_fn.1 (heat2d.c:83 @ 400d93)
  9.89     73.18    13.81                             compute_one_iteration._omp_fn.1 (heat2d.c:82 @ 400d8f)
  9.66     86.67    13.49                             compute_one_iteration._omp_fn.1 (heat2d.c:83 @ 400d70)
  9.35     99.72    13.05                             compute_one_iteration._omp_fn.1 (heat2d.c:79 @ 400d50)
  6.74    109.13     9.41                             compute_one_iteration._omp_fn.1 (heat2d.c:81 @ 400d89)
  4.86    115.91     6.78                             compute_one_iteration._omp_fn.1 (heat2d.c:80 @ 400d69)
  4.40    122.06     6.15                             compute_one_iteration._omp_fn.1 (heat2d.c:80 @ 400d83)
  4.20    127.91     5.86                             compute_one_iteration._omp_fn.1 (heat2d.c:83 @ 400d63)
  3.65    133.01     5.09                             compute_one_iteration._omp_fn.1 (heat2d.c:75 @ 400bd0)
  2.08    135.91     2.90                             compute_one_iteration._omp_fn.1 (heat2d.c:79 @ 400d5c)
  2.08    138.81     2.90                             compute_one_iteration._omp_fn.1 (heat2d.c:83 @ 400d7b)
Call graph
granularity: each sample hit covers 2 byte(s) for 0.01 of 140.15 seconds
index  time    self  children    called     name
                                                 <spontaneous>
[19]     0.0    0.03    0.00                 compute_one_iteration (heat2d.c:54 @ 400ee0) [19]
[23]     0.0    0.00    0.00  484290         frame_dummy [23]
```

The table shows where to focus optimization efforts to maximize impact.

5.2 System Profiling

This subsection provide details about the system execution profile.

5.2.1 System Resources Usage

The following charts shows the state of system resources during the execution of the program.

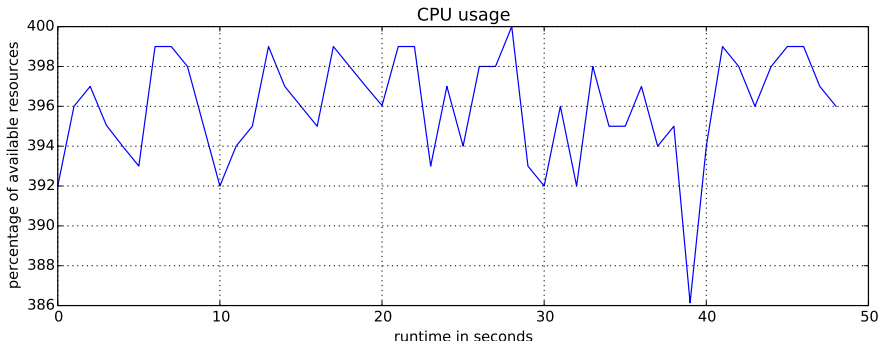

Figure 5: CPU Usage

Note that this chart is likely to show as upper limit a multiple of 100% in case a multicore system is being used.

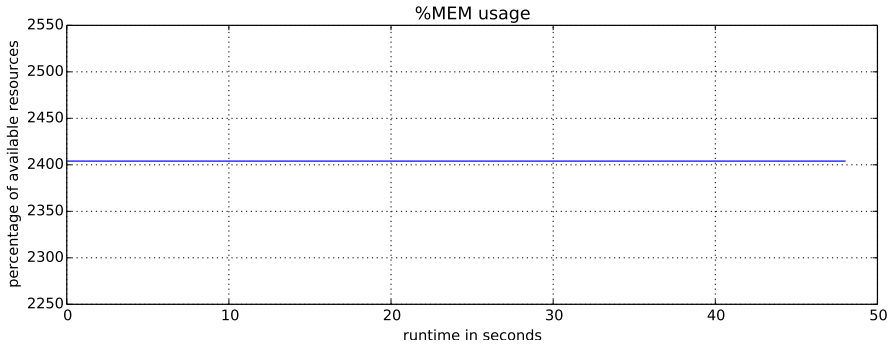

Figure 6: Memory Usage

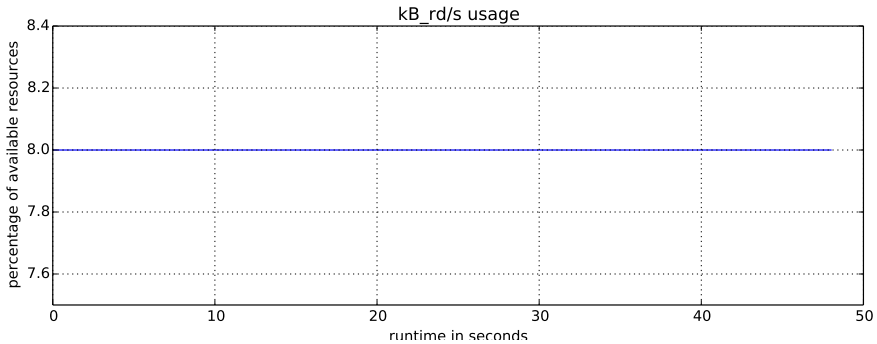

Figure 7: Reads from Disk

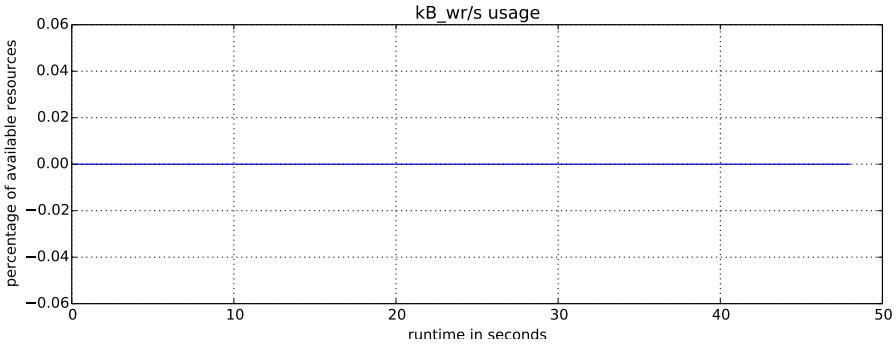

Figure 8: Writes to Disk

5.3 Hotspots

This subsection shows annotated code guiding the optimization efforts.

```
      :       solution[cur_gen][0][i] = solution[cur_gen][1][i];
      :       solution[cur_gen][CRESN - 1][i] = solution[cur_gen][CRESN - 2][i];
      :     }
      :   /* corners ? */
      : #pragma omp parallel for shared(solution,cur_gen,next_gen,diff_constant) private(i,j)
 3.52 :   400af3:       movsd  0x365a15(rip),xmm5        # 766510 <diff_constant>
 0.19 :   400b80:       lea    0x10(rcx),r11
```

```
         :   for (i = 1; i <= RESN; i++)
         :     for (j = 1; j <= RESN; j++)
         :       solution[next_gen][i][j] = solution[cur_gen][i][j] +
         :         (solution[cur_gen][i + 1][j] +
         :          solution[cur_gen][i - 1][j] +
         : #pragma omp parallel for shared(solution,cur_gen,next_gen,diff_constant) private(i,j)
         :   for (i = 1; i <= RESN; i++)
         :     for (j = 1; j <= RESN; j++)
         :       solution[next_gen][i][j] = solution[cur_gen][i][j] +
         :         (solution[cur_gen][i + 1][j] +
    0.23 :   400c13:       movhpd 0x8(r10,rax,1),xmm0
         :          solution[cur_gen][i - 1][j] +
         :          solution[cur_gen][i][j + 1] +
         :          solution[cur_gen][i][j - 1] -
         :          4.0 * solution[cur_gen][i][j]) * diff_constant;
    1.84 :   400c1a:       movsd  (rdx,rax,1),xmm1
         : #pragma omp parallel for shared(solution,cur_gen,next_gen,diff_constant) private(i,j)
         :   for (i = 1; i <= RESN; i++)
         :     for (j = 1; j <= RESN; j++)
         :       solution[next_gen][i][j] = solution[cur_gen][i][j] +
         :         (solution[cur_gen][i + 1][j] +
         :          solution[cur_gen][i - 1][j] +
    8.31 :   400c1f:       movhpd 0x8(rax,rsi,1),xmm2
         :          solution[cur_gen][i][j + 1] +
         :          solution[cur_gen][i][j - 1] -
         :          4.0 * solution[cur_gen][i][j]) * diff_constant;
    1.51 :   400c25:       movhpd 0x8(rax,rdx,1),xmm1
         : #pragma omp parallel for shared(solution,cur_gen,next_gen,diff_constant) private(i,j)
         :   for (i = 1; i <= RESN; i++)
         :     for (j = 1; j <= RESN; j++)
         :       solution[next_gen][i][j] = solution[cur_gen][i][j] +
         :         (solution[cur_gen][i + 1][j] +
    9.15 :   400c2b:       addpd  xmm2,xmm0
         :          solution[cur_gen][i - 1][j] +
         :          solution[cur_gen][i][j + 1] +
         :          solution[cur_gen][i][j - 1] -
         :          4.0 * solution[cur_gen][i][j]) * diff_constant;
    0.33 :   400c2f:       movapd xmm1,xmm2
    1.12 :   400c33:       mulpd  xmm3,xmm2
         : #pragma omp parallel for shared(solution,cur_gen,next_gen,diff_constant) private(i,j)
         :   for (i = 1; i <= RESN; i++)
         :     for (j = 1; j <= RESN; j++)
         :       solution[next_gen][i][j] = solution[cur_gen][i][j] +
         :         (solution[cur_gen][i + 1][j] +
         :          solution[cur_gen][i - 1][j] +
    0.92 :   400c37:       addpd  (rdi,rax,1),xmm0
         :          solution[cur_gen][i][j + 1] +
    8.45 :   400c3c:       addpd  (r8,rax,1),xmm0
         :          solution[cur_gen][i][j - 1] -
    2.49 :   400c42:       subpd  xmm2,xmm0
         :          4.0 * solution[cur_gen][i][j]) * diff_constant;
    9.57 :   400c46:       mulpd  xmm4,xmm0
         :   /* corners ? */
         : #pragma omp parallel for shared(solution,cur_gen,next_gen,diff_constant) private(i,j)
         :   for (i = 1; i <= RESN; i++)
         :     for (j = 1; j <= RESN; j++)
         :       solution[next_gen][i][j] = solution[cur_gen][i][j] +
   12.58 :   400c4a:       addpd  xmm1,xmm0
   16.32 :   400c4e:       movlpd xmm0,(rcx,rax,1)
    8.25 :   400c59:       add    $0x10,rax
    1.10 :   400c5d:       cmp    $0x960,rax
    0.12 :   400c6c:       add    $0x970,r8
         :       solution[cur_gen][0][i] = solution[cur_gen][1][i];
         :       solution[cur_gen][CRESN - 1][i] = solution[cur_gen][CRESN - 2][i];
         :     }
         :   /* corners ? */
         : #pragma omp parallel for shared(solution,cur_gen,next_gen,diff_constant) private(i,j)
--
         :     {
         :       compute_one_iteration (sim_time);
```

```
         :   int temp;
         :   setup ();
         :   for (sim_time = 0; sim_time < final; sim_time += time_step)
   99.81 :   40079a:       movsd  0x365d5e(rip),xmm0        # 766500 <sim_time>
         :     {
         :       compute_one_iteration (sim_time);
         :       temp = cur_gen;
    0.19 :   4007a2:       mov    0x365d64(rip),eax        # 76650c <cur_gen>
         :   int temp;
         :   setup ();
         :   for (sim_time = 0; sim_time < final; sim_time += time_step)
         :     {
         :       compute_one_iteration (sim_time);
--
         :   int i, j;
         :   /* set boundary values */
         :   for (i = 0; i < CRESN; i++)
         :     {
         :       if (i < 256 || i > 768)
    5.64 :   400da0:       cmp    $0x200,eax
         :         solution[cur_gen][i][0] = solution[cur_gen][i][1];
         :       else
         :         solution[cur_gen][i][0] = MAX_HEAT;
    1.13 :   400dab:       movsd  0x135(rip),xmm1        # 400ee8 <_IO_stdin_used+0x18>
    0.28 :   400db3:       movsd  xmm1,(rdx)
    9.41 :   400db7:       add    $0x1,eax
    0.19 :   400dba:       add    $0x970,rdx
         : void compute_one_iteration()
         : {
         :   int i, j;
         :   /* set boundary values */
         :   for (i = 0; i < CRESN; i++)
    0.38 :   400dc1:       cmp    $0x2e,eax
         :       else
         :         solution[cur_gen][i][0] = MAX_HEAT;
         :     }
         :   for (i = 0; i < CRESN; i++)
         :     {
         :       solution[cur_gen][i][CRESN - 1] = solution[cur_gen][i][CRESN - 2];
    2.63 :   400dd0:       movsd  0x602a20(rbx,rax,1),xmm0
    4.23 :   400de2:       add    $0x970,rax
         :       if (i < 256 || i > 768)
         :         solution[cur_gen][i][0] = solution[cur_gen][i][1];
         :       else
         :         solution[cur_gen][i][0] = MAX_HEAT;
         :     }
         :   for (i = 0; i < CRESN; i++)
    1.79 :   400de8:       cmp    $0xb2220,rax
         :     {
         :       solution[cur_gen][i][CRESN - 1] = solution[cur_gen][i][CRESN - 2];
         :     }
         :   for (i = 0; i < CRESN; i++)
         :     {
--
         :   /* set boundary values */
         :   for (i = 0; i < CRESN; i++)
         :     {
         :       if (i < 256 || i > 768)
         :         solution[cur_gen][i][0] = solution[cur_gen][i][1];
    6.77 :   400e3b:       movsd  0x8(rdx),xmm0
   36.41 :   400e40:       movsd  xmm0,(rdx)
    2.92 :   400e44:       jmpq   400db7 <compute_one_iteration+0x37>
 Percent | Source code & Disassembly of libc-2.19.so for cpu-clock
-----------------------------------------------------------------------
         : Disassembly of section .text:
    0.27 :   147f70:       mov    rdi,rax
    0.55 :   147fb0:       movdqu (rsi),xmm0
    0.27 :   147fb4:       mov    rdi,rcx
    0.27 :   147fbf:       mov    rcx,r8
    0.27 :   147fcb:       mov    0x27625e(rip),rcx        # 3be230 <obstack_exit_failure+0x38>
```

```
    1.92 :   147fd2:       cmp    rcx,rdx
    0.27 :   148006:       add    rdx,rsi
    0.27 :   14807b:       movdqu xmm0,(r8)
    0.82 :   1480d7:       lea    -0x80(rdx),rdx
    0.55 :   1480e1:       movdqa (rsi),xmm4
    5.48 :   1480e9:       movaps 0x20(rsi),xmm2
    1.37 :   1480ed:       movaps 0x30(rsi),xmm3
    6.85 :   1480f1:       movdqa xmm4,(rdi)
    0.82 :   1480f5:       movaps xmm1,0x10(rdi)
    0.82 :   1480f9:       movaps xmm2,0x20(rdi)
    1.10 :   148101:       sub    $0x80,rdx
    0.55 :   148108:       movaps 0x40(rsi),xmm4
    0.82 :   14810c:       movaps 0x50(rsi),xmm5
    4.66 :   148110:       movaps 0x60(rsi),xmm6
    0.55 :   148114:       movaps 0x70(rsi),xmm7
    7.67 :   148118:       lea    0x80(rsi),rsi
    0.82 :   14811f:       movaps xmm4,0x40(rdi)
    0.27 :   148123:       movaps xmm5,0x50(rdi)
    0.82 :   148127:       movaps xmm6,0x60(rdi)
    1.10 :   14812b:       movaps xmm7,0x70(rdi)
    1.10 :   14812f:       lea    0x80(rdi),rdi
    0.27 :   148152:       movdqa xmm4,(rdi)
    3.29 :   148188:       lea    (r11,rdx,1),rdx
    0.27 :   148280:       sub    $0x10,rdx
    0.82 :   148284:       movdqa -0x10(rsi),xmm1
    0.82 :   148301:       movdqa -0x10(rsi),xmm0
    9.59 :   148306:       movaps -0x20(rsi),xmm1
    1.10 :   14830a:       movaps -0x30(rsi),xmm2
    8.77 :   14830e:       movaps -0x40(rsi),xmm3
    0.55 :   148312:       movdqa xmm0,-0x10(rdi)
    1.10 :   148317:       movaps xmm1,-0x20(rdi)
    0.55 :   14831b:       movaps xmm2,-0x30(rdi)
    2.19 :   14831f:       movaps xmm3,-0x40(rdi)
    0.27 :   148323:       sub    $0x80,rdx
    9.59 :   14832e:       movaps -0x60(rsi),xmm5
   11.23 :   148336:       movaps -0x80(rsi),xmm7
    1.10 :   14833a:       lea    -0x80(rsi),rsi
    1.92 :   14833e:       movaps xmm4,-0x50(rdi)
    1.10 :   148342:       movaps xmm5,-0x60(rdi)
    0.55 :   148346:       movaps xmm6,-0x70(rdi)
    0.82 :   14834a:       movaps xmm7,-0x80(rdi)
    1.10 :   14834e:       lea    -0x80(rdi),rdi
    0.27 :   148352:       jae    148301 <__nss_hosts_lookup+0x11081>
    0.27 :   148383:       sub    $0x40,rsi
    0.55 :   1483a0:       lea    (r11,rdx,1),rdx
    0.27 :   1499d0:       mov    -0x10(rsi),r11
------------------------------------------------------------------------------
         : Disassembly of section .text:
   19.35 :   74d0:       mov    0x206af1(rip),rax        # 20dfc8 <omp_in_final_+0x203c18>
    4.30 :   74d7:       mov    fs:0x10(rax),rax
    2.15 :   74dc:       test   rax,rax
    2.15 :   74e1:       mov    (rax),eax
   72.04 :   74e3:       retq
 Percent | Source code & Disassembly of libc-2.19.so for cpu-clock
------------------------------------------------------------------------
         : Disassembly of section .text:
    1.37 :   7fc82:       push   r14
    1.37 :   7fc89:       push   rbx
    1.37 :   7fc95:       mov    rsi,0x8(rsp)
    5.48 :   7fcbe:       cmp    0x340e7b(rip),rbp        # 3c0b40 <__malloc_initialize_hook+0x120>
    2.74 :   7fcc5:       ja     7fd39 <_IO_str_seekoff+0x2b99>
    5.48 :   7fd40:       ja     7fdbf <_IO_str_seekoff+0x2c1f>
    6.85 :   7fdbf:       mov    rbp,r9
    1.37 :   7fe08:       lea    0x58(rbx),r14
   15.07 :   7fe3f:       shr    $0xc,rax
    4.11 :   7fe43:       mov    rax,0x48(rsp)
    1.37 :   7fe5b:       add    $0x77,eax
    1.37 :   7fe6e:       add    $0x7c,eax
    1.37 :   7fe80:       sub    rax,0x10(rsp)
    2.74 :   7fe97:       mov    $0x2710,r15d
```

```
    1.37 :   7fe9d:       jmp    7fee6 <_IO_str_seekoff+0x2d46>
   16.44 :   7fee6:       mov    0x70(rbx),r12
    1.37 :   7feea:       cmp    r14,r12
    4.11 :   7fef3:       mov    0x8(r12),rsi
    1.37 :   7fefd:       cmp    $0x10,rsi
    1.37 :   80187:       cmp    $0x3ff,rbp
    2.74 :   801bf:       seta   cl
    1.37 :   8022c:       lea    0x20(rbp),rdx
    5.48 :   80234:       mov    rax,r12
    2.74 :   80314:       cmp    rax,rbx
    8.22 :   80365:       cmp    rax,rdx
    1.37 :   80858:       lea    0x33df01(rip),rax        # 3be760 <__malloc_hook+0x20>
 Percent | Source code & Disassembly of libc-2.19.so for cpu-clock
-------------------------------------------------------------------------
         : Disassembly of section .text:
    1.96 :   7f058:       mov    0x10(rsp),esi
    1.96 :   7f07b:       je     7fb0c <_IO_str_seekoff+0x296c>
    1.96 :   7f081:       testb  $0x2,0x4(r12)
    3.92 :   7f0b8:       mov    0x341a72(rip),esi        # 3c0b30 <__malloc_initialize_hook+0x110>
    1.96 :   7f0be:       test   esi,esi
    1.96 :   7f0c6:       testb  $0x1,0x8(rbx)
    3.92 :   7f16a:       lea    0x58(r12),rdx
    7.84 :   7f180:       mov    rax,0x10(rbx)
    1.96 :   7f184:       mov    rdx,0x18(rbx)
    1.96 :   7f1a3:       mov    rbp,rax
    1.96 :   7f1aa:       mov    rax,0x8(rbx)
    1.96 :   7f1b2:       cmp    $0xffff,rbp
   21.57 :   7f3b8:       mov    $0x1,esi
    7.84 :   7f3bd:       mov    0x10(rsp),eax
    3.92 :   7f3c8:       je     7f3d8 <_IO_str_seekoff+0x2238>
    9.80 :   7f3d0:       jne    84bbb <malloc_info+0x2bb>
    5.88 :   7f4b6:       jne    84bd7 <malloc_info+0x2d7>
    3.92 :   7f4bc:       jmp    7f4c8 <_IO_str_seekoff+0x2328>
    1.96 :   7f54d:       add    r14,rbp
    3.92 :   7f588:       jb     7f490 <_IO_str_seekoff+0x22f0>
    1.96 :   7f595:       lea    0x33fa3c(rip),rdx        # 3befd8 <__malloc_hook+0x898>
    5.88 :   7faf0:       and    $0xfffffffffffffff8,rdx
 Percent | Source code & Disassembly of libc-2.19.so for cpu-clock
-------------------------------------------------------------------------
         : Disassembly of section .text:
   12.00 :   82750:       push   rbp
   12.00 :   82760:       mov    (rax),rax
   16.00 :   82763:       test   rax,rax
    4.00 :   8276c:       mov    0x33b60d(rip),rax        # 3bdd80 <_IO_file_jumps+0x6e0>
   12.00 :   82777:       test   rbx,rbx
   16.00 :   82794:       jne    84dbb <malloc_info+0x4bb>
    4.00 :   827a8:       mov    rbx,rdi
    4.00 :   827b3:       mov    rax,rdx
    4.00 :   827bf:       je     827cc <__libc_malloc+0x7c>
   16.00 :   827c4:       jne    84dd6 <malloc_info+0x4d6>
 Percent | Source code & Disassembly of libc-2.19.so for cpu-clock
-------------------------------------------------------------------------
         : Disassembly of section .text:
   31.58 :   82df0:       mov    0x33b0f1(rip),rax        # 3bdee8 <_IO_file_jumps+0x848>
   10.53 :   82df7:       mov    (rax),rax
   47.37 :   82dfa:       test   rax,rax
   10.53 :   82e10:       lea    -0x10(rdi),rsi
-------------------------------------------------------------------------------
         : Disassembly of section .text:
   87.50 :   7490:       cmpq   $0xffffffffffffffff,0x206cf8(rip)        # 20e190 <omp_in_final_+0x203de0>
   12.50 :   7498:       jne    749f <GOMP_parallel_end+0xf>
 Percent | Source code & Disassembly of libc-2.19.so for cpu-clock
-------------------------------------------------------------------------
         : Disassembly of section .text:
   28.57 :   85094:       callq  f1280 <__sbrk>
   28.57 :   850a3:       add    $0x8,rsp
   28.57 :   850aa:       retq
-------------------------------------------------------------------------------
         : Disassembly of section .text:
   33.33 :   7460:       push   r12
```

```
  16.67 :   747e:       mov    rbp,rdi
  16.67 :   7488:       mov    rax,rcx
  33.33 :   748b:       jmpq   85f0 <omp_in_final+0x390>
 Percent | Source code & Disassembly of libc-2.19.so for cpu-clock
-------------------------------------------------------------------------
         : Disassembly of section .text:
 Percent | Source code & Disassembly of libc-2.19.so for cpu-clock
-------------------------------------------------------------------------
--
 Percent | Source code & Disassembly of heat2d for cpu-clock
-----------------------------------------------------------
         : Disassembly of section .plt:
-----------------------------------------------------------------------------
         : Disassembly of section .text:
  25.00 :   74f7:       mov    fs:0x28(rax),eax
  25.00 :   74fb:       retq
 Percent | Source code & Disassembly of heat2d for cpu-clock
-----------------------------------------------------------
         : Disassembly of section .plt:
 Percent | Source code & Disassembly of heat2d for cpu-clock
-----------------------------------------------------------
```

6 Low Level

This section provide details about low level details such as vectorization and performance counters.

6.1 Vectorization Report

This subsection provide details about vectorization status of the program loops.

```
Analyzing loop at heat2d.c:46
heat2d.c:46: note: not vectorized: loop contains function calls or data references that cannot be analyzed
heat2d.c:46: note: bad data references.
heat2d.c:43: note: vectorized 0 loops in function.
heat2d.c:43: note: not vectorized: not enough data-refs in basic block.
heat2d.c:43: note: not vectorized: not enough data-refs in basic block.
heat2d.c:43: note: not vectorized: not enough data-refs in basic block.
heat2d.c:46: note: not vectorized: not enough data-refs in basic block.
heat2d.c:43: note: not vectorized: not enough data-refs in basic block.
heat2d.c:43: note: not vectorized: not enough data-refs in basic block.
heat2d.c:43: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: versioning for alias required: can't determine dependence between solution[pretmp_108][pretmp_1
heat2d.c:75: note: versioning not yet supported for outer-loops.
heat2d.c:75: note: bad data dependence.
Analyzing loop at heat2d.c:77
heat2d.c:77: note: versioning for alias required: can't determine dependence between solution[pretmp_108][pretmp_1
heat2d.c:77: note: versioning for alias required: can't determine dependence between solution[pretmp_108][pretmp_1
heat2d.c:77: note: versioning for alias required: can't determine dependence between solution[pretmp_108][i_3][j_2
heat2d.c:77: note: versioning for alias required: can't determine dependence between solution[pretmp_108][i_3][_29
heat2d.c:77: note: versioning for alias required: can't determine dependence between solution[pretmp_108][i_3][j_3
heat2d.c:77: note: misalign = 8 bytes of ref solution[pretmp_108][pretmp_112][j_39]
heat2d.c:77: note: misalign = 8 bytes of ref solution[pretmp_108][pretmp_113][j_39]
heat2d.c:77: note: misalign = 0 bytes of ref solution[pretmp_108][i_3][j_26]
heat2d.c:77: note: misalign = 0 bytes of ref solution[pretmp_108][i_3][_29]
heat2d.c:77: note: misalign = 8 bytes of ref solution[pretmp_108][i_3][j_39]
heat2d.c:77: note: misalign = 8 bytes of ref solution[pretmp_106][i_3][j_39]
heat2d.c:77: note: num. args = 4 (not unary/binary/ternary op).
heat2d.c:77: note: not ssa-name.
heat2d.c:77: note: use not simple.
heat2d.c:77: note: num. args = 4 (not unary/binary/ternary op).
heat2d.c:77: note: not ssa-name.
heat2d.c:77: note: use not simple.
heat2d.c:77: note: num. args = 4 (not unary/binary/ternary op).
heat2d.c:77: note: not ssa-name.
heat2d.c:77: note: use not simple.
heat2d.c:77: note: num. args = 4 (not unary/binary/ternary op).
heat2d.c:77: note: not ssa-name.
heat2d.c:77: note: use not simple.
heat2d.c:77: note: num. args = 4 (not unary/binary/ternary op).
```

```
heat2d.c:77: note: not ssa-name.
heat2d.c:77: note: use not simple.
Vectorizing loop at heat2d.c:77
heat2d.c:75: note: vectorized 1 loops in function.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not consecutive access pretmp_106 = next_gen;
heat2d.c:75: note: not consecutive access pretmp_108 = cur_gen;
heat2d.c:75: note: not consecutive access pretmp_110 = diff_constant;
heat2d.c:75: note: Failed to SLP the basic block.
heat2d.c:75: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:79: note: can't determine dependence between solution[pretmp_108][pretmp_112][j_140] and solution[pretmp_
heat2d.c:79: note: can't determine dependence between solution[pretmp_108][pretmp_113][j_140] and solution[pretmp_
heat2d.c:79: note: can't determine dependence between solution[pretmp_108][i_3][j_146] and solution[pretmp_106][i_
heat2d.c:79: note: can't determine dependence between solution[pretmp_108][i_3][_149] and solution[pretmp_106][i_3
heat2d.c:79: note: can't determine dependence between solution[pretmp_108][i_3][j_140] and solution[pretmp_106][i_
heat2d.c:79: note: SLP: step doesn't divide the vector-size.
heat2d.c:79: note: Unknown alignment for access: solution
heat2d.c:79: note: SLP: step doesn't divide the vector-size.
heat2d.c:79: note: Unknown alignment for access: solution
heat2d.c:79: note: SLP: step doesn't divide the vector-size.
heat2d.c:79: note: Unknown alignment for access: solution
heat2d.c:79: note: SLP: step doesn't divide the vector-size.
heat2d.c:79: note: Unknown alignment for access: solution
heat2d.c:79: note: SLP: step doesn't divide the vector-size.
heat2d.c:79: note: Unknown alignment for access: solution
heat2d.c:79: note: Failed to SLP the basic block.
heat2d.c:79: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:79: note: not vectorized: no vectype for stmt: vect_var_.72_169 = MEM[(double[2][302][302] *)vect_psoluti
 scalar_type: vector(2) double
heat2d.c:79: note: Failed to SLP the basic block.
heat2d.c:79: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:75: note: not vectorized: not enough data-refs in basic block.
heat2d.c:43: note: not vectorized: not enough data-refs in basic block.
Analyzing loop at heat2d.c:64
heat2d.c:64: note: misalign = 0 bytes of ref solution[pretmp_34][i_41][300]
heat2d.c:64: note: misalign = 8 bytes of ref solution[pretmp_34][i_41][301]
heat2d.c:64: note: not consecutive access _19 = solution[pretmp_34][i_41][300];
heat2d.c:64: note: not vectorized: complicated access pattern.
heat2d.c:64: note: bad data access.
Analyzing loop at heat2d.c:57
heat2d.c:57: note: not vectorized: control flow in loop.
heat2d.c:57: note: bad loop form.
heat2d.c:53: note: vectorized 0 loops in function.
heat2d.c:53: note: not consecutive access pretmp_34 = cur_gen;
heat2d.c:53: note: Failed to SLP the basic block.
heat2d.c:53: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:59: note: not vectorized: not enough data-refs in basic block.
heat2d.c:60: note: misalign = 8 bytes of ref solution[pretmp_34][i_40][1]
heat2d.c:60: note: misalign = 0 bytes of ref solution[pretmp_34][i_40][0]
heat2d.c:60: note: not consecutive access _12 = solution[pretmp_34][i_40][1];
heat2d.c:60: note: not consecutive access solution[pretmp_34][i_40][0] = _12;
heat2d.c:60: note: Failed to SLP the basic block.
heat2d.c:60: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:62: note: misalign = 0 bytes of ref solution[pretmp_34][i_40][0]
heat2d.c:62: note: not consecutive access solution[pretmp_34][i_40][0] = 2.0e+1;
heat2d.c:62: note: Failed to SLP the basic block.
heat2d.c:62: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:57: note: not vectorized: not enough data-refs in basic block.
heat2d.c:53: note: not vectorized: not enough data-refs in basic block.
```

```
heat2d.c:53: note: not vectorized: not enough data-refs in basic block.
heat2d.c:66: note: misalign = 0 bytes of ref solution[pretmp_34][i_41][300]
heat2d.c:66: note: misalign = 8 bytes of ref solution[pretmp_34][i_41][301]
heat2d.c:66: note: not consecutive access _19 = solution[pretmp_34][i_41][300];
heat2d.c:66: note: not consecutive access solution[pretmp_34][i_41][301] = _19;
heat2d.c:66: note: Failed to SLP the basic block.
heat2d.c:66: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:53: note: not vectorized: not enough data-refs in basic block.
heat2d.c:70: note: not vectorized: not enough data-refs in basic block.
Analyzing loop at heat2d.c:28
heat2d.c:28: note: not vectorized: number of iterations cannot be computed.
heat2d.c:28: note: bad loop form.
heat2d.c:18: note: vectorized 0 loops in function.
heat2d.c:20: note: misalign = 0 bytes of ref final
heat2d.c:20: note: not consecutive access final = 1.024e+3;
heat2d.c:20: note: Failed to SLP the basic block.
heat2d.c:20: note: not vectorized: failed to find SLP opportunities in basic block.
heat2d.c:22: note: not vectorized: not enough data-refs in basic block.
heat2d.c:26: note: not vectorized: not enough data-refs in basic block.
heat2d.c:36: note: not vectorized: not enough data-refs in basic block.
heat2d.c:18: note: not vectorized: not enough data-refs in basic block.
heat2d.c:30: note: not vectorized: not enough data-refs in basic block.
heat2d.c:18: note: not vectorized: not enough data-refs in basic block.
```

The details above shows the list of loops in the program and if they are being vectorized or not. These reports can pinpoint areas where the compiler cannot apply vectorization and related optimizations. It may be possible to modify your code or communicate additional information to the compiler to guide the vectorization and/or optimizations.

6.2 Counters Report

This subsection provides details about software and hardware counters.

```
 Performance counter stats for './heat2d' (3 runs):

     199110.369461      task-clock (msec)         #    3.944 CPUs utilized            ( +-  0.87 )
            3,562      context-switches          #    0.018 K/sec                    ( +- 11.48 )
                8      cpu-migrations            #    0.000 K/sec                    ( +- 25.00 )
              427      page-faults               #    0.002 K/sec                    ( +-  0.08 )
   <not supported>      cycles
   <not supported>      stalled-cycles-frontend
   <not supported>      stalled-cycles-backend
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

      50.489125220 seconds time elapsed                                          ( +-  0.97 )
```

The details above shows counters that provide low-overhead access to detailed performance information using internal registers of the CPU.

References

[1] Piotr Luszczek and Jack J. Dongarra and David Koester and Rolf Rabenseifner and Bob Lucas and Jeremy Kepner and John Mccalpin and David Bailey and Daisuke Takahashi, *Introduction to the HPC Challenge Benchmark Suite.* Technical Report, 2005.

[2] Amdahl, Gene M., *Validity of the single processor approach to achieving large scale computing capabilities.* Communications of the ACM, Proceedings of the April 18-20, 1967, spring joint computer conference Pages 483-485, 1967.

[3] John L. Gustafson, *Reevaluating Amdahl's Law.* Communications of the ACM, Volume 31 Pages 532-533, 1988.

[4] OpenMP Architecture Review Board, *OpenMP Application Program Interface. http://www.openmp.org*, 3.0, May 2008.

B.3. Conjunto de Mandelbrot

`mandel` Performance Report

20150607-151437

Abstract

This performance report is intended to support performance analysis and optimization activities. It includes details on program behavior, system configuration and capabilities, including support data from well-known performance analysis tools.

This report was generated using `hotspot` version `0.1`. Homepage `http://www.github.com/moreandres/hotspot`. Full execution log can be found at `~/.hotspot/mandel/20150607-151437/hotspot.log`.

Contents

1 Program **1**
2 System Capabilities **1**
2.1 System Configuration 1
2.2 System Performance Baseline 2
3 Workload **2**
3.1 Workload Footprint 2
3.2 Workload Stability 3
3.3 Workload Optimization 4
4 Scalability **4**
4.1 Problem Size Scalability 4
4.2 Computing Scalability 5
5 Profile **5**
5.1 Program Profiling 5
5.1.1 Flat Profile 6
5.2 System Profiling 6
5.2.1 System Resources Usage 6
5.3 Hotspots 7
6 Low Level **8**
6.1 Vectorization Report 8
6.2 Counters Report 9

1 Program

This section provides details about the program being analyzed.

1. Program: `mandel`.
   Program is the name of the program.
2. Timestamp: `20150607-151437`.
   Timestamp is a unique identifier used to store information on disk.
3. Parameters Range: `[32768, 33792, 34816, 35840, 36864, 37888, 38912, 39936, 40960, 41984, 43008, 44032, 45056, 46080, 47104, 48128, 49152, 50176, 51200, 52224, 53248, 54272, 55296, 56320, 57344, 58368, 59392, 60416, 61440, 62464, 63488, 64512]`.
   Parameters range is the problem size set used to scale the program.

2 System Capabilities

This section provides details about the system being used for the analysis.

2.1 System Configuration

This subsection provides details about the system configuration.

The hardware in the system is summarized using a hardware lister utility. It reports exact memory configuration, firmware version, mainboard configuration, CPU version and speed, cache configuration, bus speed and others.

The hardware configuration can be used to contrast the system capabilities to well-known benchmarks results on similar systems.

```
memory      7984MiB System memory
processor   Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz
bridge      440FX - 82441FX PMC [Natoma]
bridge      82371SB PIIX3 ISA [Natoma/Triton II]
storage     82371AB/EB/MB PIIX4 IDE
network     82540EM Gigabit Ethernet Controller
bridge      82371AB/EB/MB PIIX4 ACPI
storage     82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode]
```

The software in the system is summarized using the GNU/Linux platform string.

```
Linux-3.16.0-30-generic-x86_64-with-Ubuntu-14.04-trusty
```

The software toolchain is built upon the following components.

1. Host: `ubuntu`
2. Distribution: `Ubuntu, 14.04, trusty`.
   This codename provides LSB (Linux Standard Base) and distribution-specific information.
3. Compiler: `gcc (Ubuntu 4.8.2-19ubuntu1) 4.8.2`.
   Version number of the compiler program.
4. C Library: `GNU C Library (Ubuntu EGLIBC 2.19-0ubuntu6.6) stable release version 2.19`.
   Version number of the C library.

The software configuration can be used to contrast the system capabilities to well-known benchmark results on similar systems.

2.2 System Performance Baseline

This subsection provides details about the system capabilities.

A set of performance results is included as a reference to contrast systems and to verify hardware capabilities using well-known synthetic benchmarks.

The HPC Challenge benchmark [1] consists of different tests:

1. HPL: the Linpack TPP benchmark which measures the floating point rate of execution for solving a linear system of equations.
2. DGEMM: measures the floating point rate of execution of double precision real matrix-matrix multiplication.
3. PTRANS (parallel matrix transpose): exercises the communications where pairs of processors communicate with each other simultaneously.
4. RandomAccess: measures the rate of integer random updates of memory (GUPS).
5. STREAM: a simple synthetic benchmark program that measures sustainable memory bandwidth (in GB/s).
6. FFT: measures the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT).

Table 1: Benchmarks

Benchmark	Value	Unit
hpl	0.00385977 TFlops	tflops
dgemm	1.03413 GFlops	mflops
ptrans	0.997656 GBs	MB/s
random	0.0274034 GUPs	MB/s
stream	5.19608 MBs	MB/s
fft	1.2658 GFlops	MB/s

Most programs will have a dominant compute kernel that can be approximated by the ones above, the results helps to understand the available capacity.

3 Workload

This section provides details about the workload behavior.

3.1 Workload Footprint

The workload footprint impacts on memory hierarchy usage.

```
mandel: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux
```

Binaries should be stripped to better fit inside cache.

```
struct _IO_FILE {
int                        _flags;               /*     0     4 */

/* XXX 4 bytes hole, try to pack */

char *                     _IO_read_ptr;         /*     8     8 */
char *                     _IO_read_end;         /*    16     8 */
char *                     _IO_read_base;        /*    24     8 */
char *                     _IO_write_base;       /*    32     8 */
char *                     _IO_write_ptr;        /*    40     8 */
char *                     _IO_write_end;        /*    48     8 */
char *                     _IO_buf_base;         /*    56     8 */
/* --- cacheline 1 boundary (64 bytes) --- */
char *                     _IO_buf_end;          /*    64     8 */
```

```
char *                    _IO_save_base;        /*    72     8 */
char *                    _IO_backup_base;      /*    80     8 */
char *                    _IO_save_end;         /*    88     8 */
struct _IO_marker *       _markers;             /*    96     8 */
struct _IO_FILE *         _chain;               /*   104     8 */
int                       _fileno;              /*   112     4 */
int                       _flags2;              /*   116     4 */
__off_t                   _old_offset;          /*   120     8 */
/* --- cacheline 2 boundary (128 bytes) --- */
short unsigned int        _cur_column;          /*   128     2 */
signed char               _vtable_offset;       /*   130     1 */
char                      _shortbuf[1];         /*   131     1 */

/* XXX 4 bytes hole, try to pack */

_IO_lock_t *              _lock;                /*   136     8 */
__off64_t                 _offset;              /*   144     8 */
void *                    __pad1;               /*   152     8 */
void *                    __pad2;               /*   160     8 */
void *                    __pad3;               /*   168     8 */
void *                    __pad4;               /*   176     8 */
size_t                    __pad5;               /*   184     8 */
/* --- cacheline 3 boundary (192 bytes) --- */
int                       _mode;                /*   192     4 */
char                      _unused2[20];         /*   196    20 */

/* size: 216, cachelines: 4, members: 29 */
/* sum members: 208, holes: 2, sum holes: 8 */
/* last cacheline: 24 bytes */
};
struct _IO_marker {
struct _IO_marker *       _next;                /*     0     8 */
struct _IO_FILE *         _sbuf;                /*     8     8 */
int                       _pos;                 /*    16     4 */

/* size: 24, cachelines: 1, members: 3 */
/* padding: 4 */
/* last cacheline: 24 bytes */
};
struct complextype {
float                     real;                 /*     0     4 */
float                     imag;                 /*     4     4 */

/* size: 8, cachelines: 1, members: 2 */
/* last cacheline: 8 bytes */
};
```

The in-memory layout of data structures can be used to identify issues. Reorganizing data to remove alignment holes will improve CPU cache utilization.

More information `https://www.kernel.org/doc/ols/2007/ols2007v2-pages-35-44.pdf`

3.2 Workload Stability

This subsection provides details about workload stability.

1. Execution time:
   (a) problem size range: `32768 - 65536`
   (b) geomean: `9.56445` seconds
   (c) average: `9.56486` seconds
   (d) stddev: `0.08794`
   (e) min: `9.40741` seconds
   (f) max: `9.72290` seconds
   (g) repetitions: `32` times

The histogram plots the elapsed times and shows how they fit in a normal distribution sample.

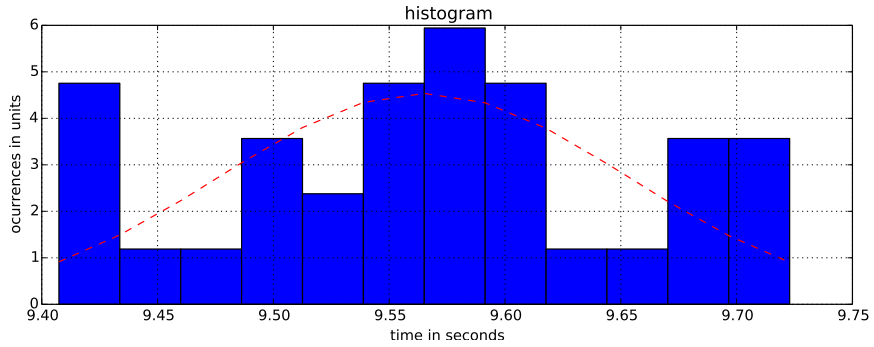

Figure 1: Results Distribution

The workload should run for at least one minute to fully utilize system resources. The execution time of the workload should be stable and the standard deviation less than 3 units.

3.3 Workload Optimization

This section shows how the program reacts to different optimization levels.

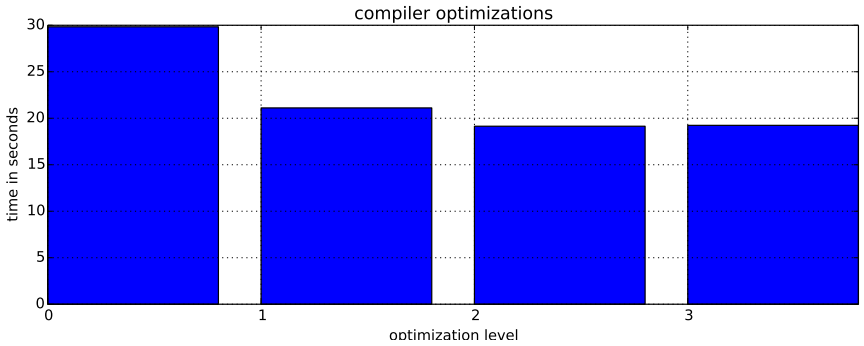

Figure 2: Optimization Levels

4 Scalability

This section provides details about the scaling behavior of the program.

4.1 Problem Size Scalability

A chart with the execution time when scaling the problem size.

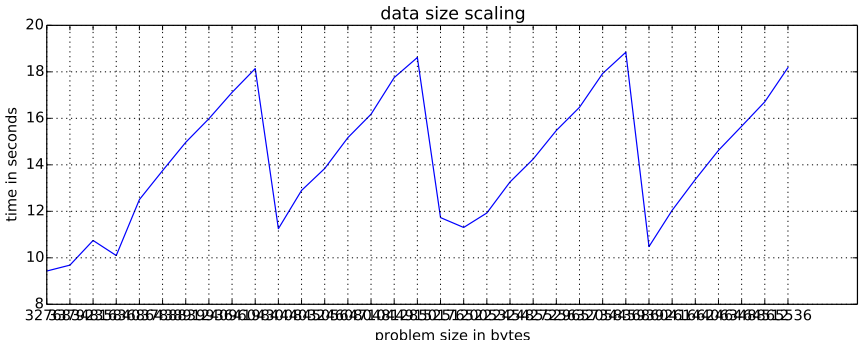

Figure 3: Problem size times

The chart will show how computing time increases when increasing problem size. There should be no valleys or bumps if processing properly balanced across computational units.

4.2 Computing Scalability

A chart with the execution time when scaling computation units.

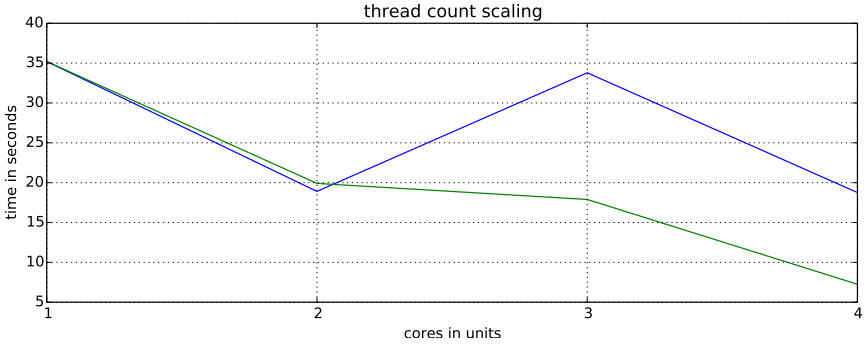

Figure 4: Thread count times

The chart will show how computing time decreases when increasing processing units. An ideal scaling line is provided for comparison.

The parallel and serial fractions of the program can be estimated using the information above.

1. Parallel Fraction: `0.92591`.
   Portion of the program doing parallel work.
2. Serial: `0.07409`.
   Portion of the program doing serial work.

Optimization limits can be estimated using scaling laws.

1. Amdalah Law for 1024 procs: `13.49756 times`.
   Optimizations are limited up to this point when scaling problem size. [2]
2. Gustafson Law for 1024 procs: `948.20853 times`.
   Optimizations are limited up to this point when not scaling problem size. [3]

5 Profile

This section provides details about the execution profile of the program and the system.

5.1 Program Profiling

This subsection provides details about the program execution profile.

5.1.1 Flat Profile

The flat profile shows how much time your program spent in each function, and how many times that function was called.

```
Flat profile:
Each sample counts as 0.01 seconds.
     cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 44.60     17.09    17.09                             main._omp_fn.0 (mandel.c:45 @ 400afd)
 16.52     23.42     6.33                             main._omp_fn.0 (mandel.c:48 @ 400b0f)
  9.76     27.16     3.74                             main._omp_fn.0 (mandel.c:43 @ 400aed)
  7.69     30.10     2.95                             main._omp_fn.0 (mandel.c:43 @ 400ad8)
  4.01     31.64     1.53                             main._omp_fn.0 (mandel.c:48 @ 400ac0)
  3.57     33.01     1.37                             main._omp_fn.0 (mandel.c:42 @ 400ae1)
  3.21     34.24     1.23                             main._omp_fn.0 (mandel.c:43 @ 400ad1)
  3.21     35.47     1.23                             main._omp_fn.0 (mandel.c:43 @ 400ae5)
  3.20     36.70     1.23                             main._omp_fn.0 (mandel.c:46 @ 400ace)
  3.06     37.87     1.17                             main._omp_fn.0 (mandel.c:36 @ 400aa4)
```

The table shows where to focus optimization efforts to maximize impact.

5.2 System Profiling

This subsection provide details about the system execution profile.

5.2.1 System Resources Usage

The following charts shows the state of system resources during the execution of the program.

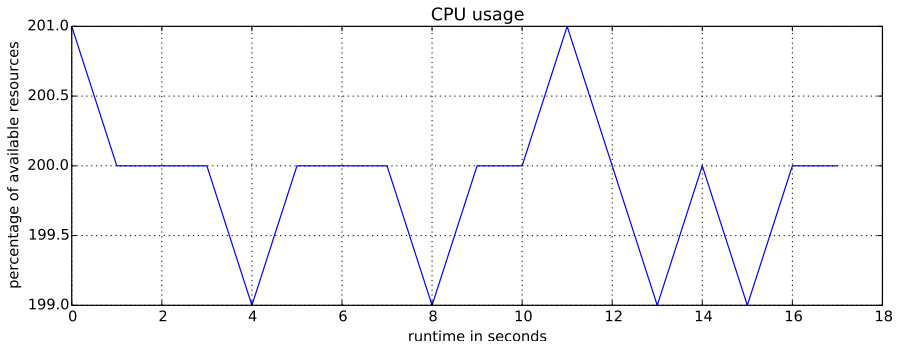

Figure 5: CPU Usage

Note that this chart is likely to show as upper limit a multiple of 100% in case a multicore system is being used.

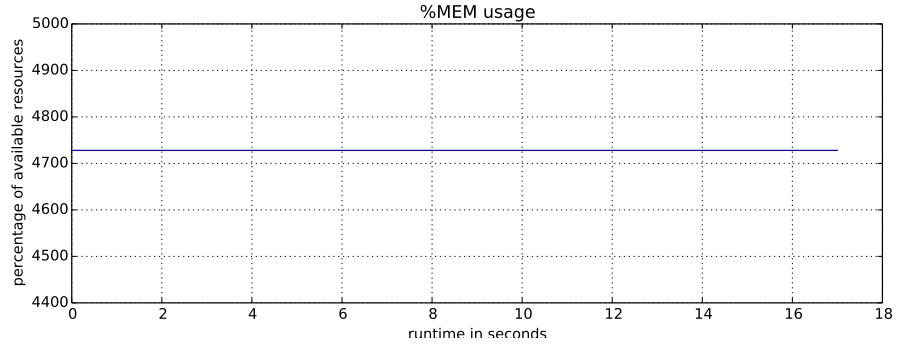

Figure 6: Memory Usage

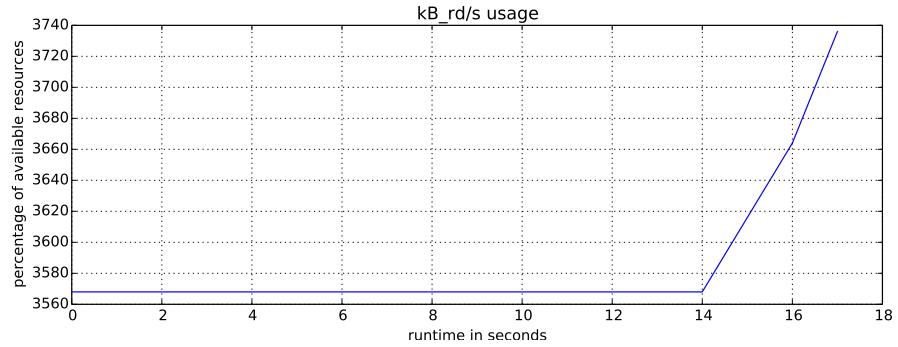

Figure 7: Reads from Disk

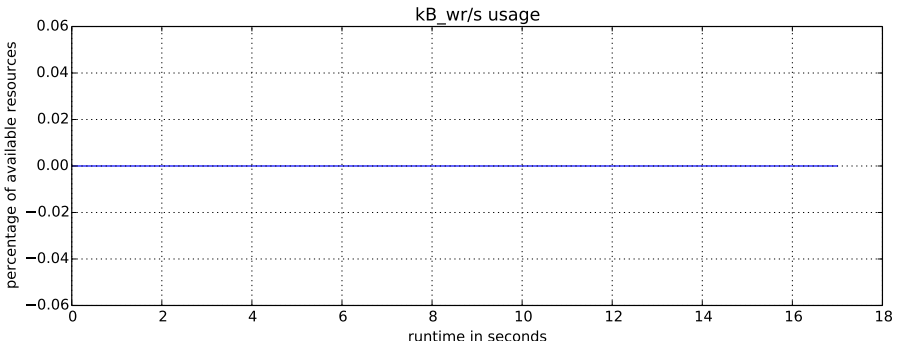

Figure 8: Writes to Disk

5.3 Hotspots

This subsection shows annotated code guiding the optimization efforts.

```
      :             z.real = temp;
      :             lensq = z.real * z.real + z.imag * z.imag;
      :             k++;
      :           }
      :         while (lensq < 4.0 && k < iters);
 6.11 :   400968:       cmp    edx,eax
      :         do
      :           {
      :             temp = z.real * z.real - z.imag * z.imag + c.real;
      :             z.imag = 2.0 * z.real * z.imag + c.imag;
      :             z.real = temp;
 1.58 :   400970:       movaps xmm1,xmm0
      :         c.imag = Y_MAX - i * (Y_MAX - Y_MIN) / Y_RESN;
      :         k = 0;
      :         do
      :           {
      :             temp = z.real * z.real - z.imag * z.imag + c.real;
      :             z.imag = 2.0 * z.real * z.imag + c.imag;
      :             z.real = temp;
      :             lensq = z.real * z.real + z.imag * z.imag;
      :             k++;
 0.55 :   400976:       add    $0x1,eax
      :         k = 0;
      :         do
      :           {
      :             temp = z.real * z.real - z.imag * z.imag + c.real;
      :             z.imag = 2.0 * z.real * z.imag + c.imag;
```

```
 6.64 :   400979:       unpcklps xmm2,xmm2
      :         c.imag = Y_MAX - i * (Y_MAX - Y_MIN) / Y_RESN;
      :         k = 0;
      :         do
      :           {
      :             temp = z.real * z.real - z.imag * z.imag + c.real;
      :             z.imag = 2.0 * z.real * z.imag + c.imag;
 0.63 :   400983:       cvtps2pd xmm2,xmm2
 6.54 :   400986:       cvtps2pd xmm0,xmm0
      :         c.imag = Y_MAX - i * (Y_MAX - Y_MIN) / Y_RESN;
      :         k = 0;
      :         do
      :           {
      :             temp = z.real * z.real - z.imag * z.imag + c.real;
 0.62 :   400989:       subss  xmm3,xmm1
      :             z.imag = 2.0 * z.real * z.imag + c.imag;
 6.21 :   40098d:       addsd  xmm0,xmm0
      :         c.imag = Y_MAX - i * (Y_MAX - Y_MIN) / Y_RESN;
      :         k = 0;
      :         do
      :           {
      :             temp = z.real * z.real - z.imag * z.imag + c.real;
      :             z.imag = 2.0 * z.real * z.imag + c.imag;
 0.57 :   400999:       addsd  xmm6,xmm0
 6.56 :   40099d:       unpcklpd xmm0,xmm0
      :             z.real = temp;
      :             lensq = z.real * z.real + z.imag * z.imag;
 5.17 :   4009a5:       movaps xmm1,xmm0
 2.26 :   4009a8:       mulss  xmm1,xmm0
 3.94 :   4009ac:       movaps xmm2,xmm3
 0.13 :   4009af:       mulss  xmm2,xmm3
19.29 :   4009b3:       addss  xmm3,xmm0
      :             k++;
      :           }
      :         while (lensq < 4.0 && k < iters);
17.24 :   4009b7:       ucomiss xmm0,xmm5
      :         if (k >= iters)
      :           res[i][j] = 0;
      :         else
      :           res[i][j] = 1;
      :   if (getenv("N"))
```

6 Low Level

This section provide details about low level details such as vectorization and performance counters.

6.1 Vectorization Report

This subsection provide details about vectorization status of the program loops.

```
mandel.c:31: note: not vectorized: multiple nested loops.
mandel.c:31: note: bad loop form.
Analyzing loop at mandel.c:33
mandel.c:33: note: not vectorized: control flow in loop.
mandel.c:33: note: bad inner-loop form.
mandel.c:33: note: not vectorized: Bad inner loop.
mandel.c:33: note: bad loop form.
Analyzing loop at mandel.c:42
mandel.c:42: note: not vectorized: control flow in loop.
mandel.c:42: note: bad loop form.
mandel.c:31: note: vectorized 0 loops in function.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not consecutive access pretmp_142 = .omp_data_i_50(D)->res;
mandel.c:31: note: Failed to SLP the basic block.
mandel.c:31: note: not vectorized: failed to find SLP opportunities in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
```

```
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:36: note: not consecutive access pretmp_129 = .omp_data_i_50(D)->iters;
mandel.c:36: note: Failed to SLP the basic block.
mandel.c:36: note: not vectorized: failed to find SLP opportunities in basic block.
mandel.c:42: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:50: note: not vectorized: not enough data-refs in basic block.
mandel.c:33: note: not vectorized: not enough data-refs in basic block.
mandel.c:53: note: SLP: step doesn't divide the vector-size.
mandel.c:53: note: Unknown alignment for access: *pretmp_142
mandel.c:53: note: Failed to SLP the basic block.
mandel.c:53: note: not vectorized: failed to find SLP opportunities in basic block.
mandel.c:51: note: SLP: step doesn't divide the vector-size.
mandel.c:51: note: Unknown alignment for access: *pretmp_142
mandel.c:51: note: Failed to SLP the basic block.
mandel.c:51: note: not vectorized: failed to find SLP opportunities in basic block.
mandel.c:48: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: not vectorized: not enough data-refs in basic block.
mandel.c:28: note: not vectorized: not enough data-refs in basic block.
mandel.c:29: note: not vectorized: not enough data-refs in basic block.
mandel.c:31: note: misalign = 0 bytes of ref .omp_data_o.1.res
mandel.c:31: note: misalign = 8 bytes of ref .omp_data_o.1.iters
mandel.c:31: note: not consecutive access .omp_data_o.1.res = &res;
mandel.c:31: note: not consecutive access .omp_data_o.1.iters = iters_1;
mandel.c:31: note: Failed to SLP the basic block.
mandel.c:31: note: not vectorized: failed to find SLP opportunities in basic block.
mandel.c:57: note: not vectorized: not enough data-refs in basic block.
mandel.c:19: note: not vectorized: no vectype for stmt: res ={v} {CLOBBER};
 scalar_type: int[1024][1024]
mandel.c:19: note: Failed to SLP the basic block.
mandel.c:19: note: not vectorized: failed to find SLP opportunities in basic block.
```

The details above shows the list of loops in the program and if they are being vectorized or not. These reports can pinpoint areas where the compiler cannot apply vectorization and related optimizations. It may be possible to modify your code or communicate additional information to the compiler to guide the vectorization and/or optimizations.

6.2 Counters Report

This subsection provides details about software and hardware counters.

```
 Performance counter stats for './mandel' (3 runs):

      37878.560427      task-clock (msec)         #    1.986 CPUs utilized            ( +-  0.75 )
               73      context-switches          #    0.002 K/sec                    ( +-  3.95 )
                5      cpu-migrations            #    0.000 K/sec                    ( +- 25.00 )
              582      page-faults               #    0.015 K/sec
   <not supported>      cycles
   <not supported>      stalled-cycles-frontend
   <not supported>      stalled-cycles-backend
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

      19.072601918 seconds time elapsed                                          ( +-  0.80 )
```

The details above shows counters that provide low-overhead access to detailed performance information using internal registers of the CPU.

References

[1] Piotr Luszczek and Jack J. Dongarra and David Koester and Rolf Rabenseifner and Bob Lucas and Jeremy Kepner and John Mccalpin and David Bailey and Daisuke Takahashi, *Introduction to the HPC Challenge Benchmark Suite*. Technical Report, 2005.

[2] Amdahl, Gene M., *Validity of the single processor approach to achieving large scale computing capabilities.* Communications of the ACM, Proceedings of the April 18-20, 1967, spring joint computer conference Pages 483-485, 1967.

[3] John L. Gustafson, *Reevaluating Amdahl's Law.* Communications of the ACM, Volume 31 Pages 532-533, 1988.

[4] OpenMP Architecture Review Board, *OpenMP Application Program Interface. http://www.openmp.org*, 3.0, May 2008.

Apéndice C

Contribuciones

C.1. Reportes Técnicos

C.1.1. Estudio de Multiplicación de Matrices

Estudio de Multiplicación de Matrices. Reporte Técnico. Realizado como parte del curso *Programación en Clusters* dictado por el Dr *Fernando Tinetti* [6].

A Case Study on High Performance Matrix Multiplication

Andrés More - amore@hal.famaf.unc.edu.ar

Abstract

This document reviews a short case study on high performance matrix multiplication. Several algorithms were implemented and contrasted against a commonly used matrix multiplication library. Performance metrics including timings and operations were gathered allowing interesting graphical comparisons of the different approaches. The results showed that carefully optimized implementations are orders of magnitude above straightforward ones.

Contents

1 Introduction **1**
- 1.1 This Work . . . 1
- 1.2 The Subject . . . 1

2 Algorithms **2**
- 2.1 Simple . . . 2
- 2.2 Blocked . . . 2
- 2.3 Transposed . . . 2
- 2.4 BLAS Library . . . 2

3 Results **3**
- 3.1 Block Size . . . 3
- 3.2 Timings . . . 3
- 3.3 Performance . . . 5

4 Parallelism **5**
- 4.1 Multi-threading . . . 5
- 4.2 Message Passing Interface . . . 5

5 Conclusions **6**

A The `mm` tool **7**
- A.1 Source Code . . . 7
- A.2 Usage . . . 7

1 Introduction

This section introduces this work and the topic under study.

1.1 This Work

This work was required to complete *Cluster Programming*, a course offered as part of the *Specialization on High Performance Computing and Grid Technology* offered at *Universidad Nacional de La Plata* (UNLP)[1].

During the generation of this short report, a tool was engineered from scratch to showcase several algorithms and to extract performance metrics.

Besides the course class-notes[2], other relevant material was consulted. This included Fernando Tinetti's work [1], a discussion on architecture-aware programming [2] and also the start point of performance evaluation on clustered systems [3].

1.2 The Subject

Matrix multiplication routines are widely used in the computational sciences in general, mostly for solving linear algebra equations. Therefore, is heavily applied on scientific modeling in particular.

Timing performance is the main roadblock preventing such models to became more complex and rich enough to really match their real-world counterparts.

More than in other areas, architecture-optimized code is needed to by-pass these time barriers.

[1] `http://www.info.unlp.edu.ar`
[2] `http://ftinetti.googlepages.com/postgrado2008`

2 Algorithms

This section surveys several intuitive matrix multiplication algorithms. For each approach, code samples showing their implementation are included as example for the reader. Square matrices will be assumed to simplify the discussion.

2.1 Simple

The formal definition of matrix multiplication is shown on equation 1. Note that it only implies the final value of each element of the result matrix; nothing is stated about how values are actually calculated.

$$(AB)_{ij} = \sum_{k=1}^{k=n} a_{i1}b_{1j} = a_{i1}b_{1j} + \ldots + a_{in}b_{nj} \quad (1)$$

The previous definition can be translated into the following C implementation.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[i * n + j] +=
        a[i * n + k] * b[k * n + j];
```

2.2 Blocked

An enhancing approach of the previous algorithm will be to apply the divide-and-conquer principle; it is expected that performing smaller matrix multiplications will optimize the usage of the memory cache hierarchy.

The implementation of the blocking approach is shown below.

```
for (i = 0; i < n; i += bs)
  for (j = 0; j < n; j += bs)
    for (k = 0; k < n; k += bs)
      block(&a[i * n + k],
              &b[k * n + j],
                &c[i * n + j], n, bs);
```

Where the definition of `block` implements each block multiplication, block size should be used as a boundary limit indicator.

```
for (i = 0; i < bs; i++)
  for (j = 0; j < bs; j++)
    for (k = 0; k < bs; k++)
      c[i * n + j] +=
        a[i * n + k] * b[k * n + j];
```

Note that the code above assumes that matrix size should be a multiple of block size.

2.3 Transposed

Another interesting approach [2] is to transpose the second matrix. In this way, the cache misses when iterating over the columns of the second matrix will be avoided.

On this case, each element is defined as shown in equation 2.

$$(AB)_{ij} = \sum_{k=1}^{k=n} a_{ik}b^T_{jk} = a_{i1}b^T_{j1} + \ldots + ... + a_{in}b^T_{jn} \quad (2)$$

The implementation on this case is very similar to our first approach; an additional step to rotate the matrix is added before the actual algorithm.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    tmp[i * n + j] = b[j * n + i];

for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[i * n + j] +=
        a[i * n + k] * tmp[j * n + k];
```

This technique will add overhead due to the need of the matrix rotation; it will also temporarily use more memory. Despite these drawbacks, performance gains are expected.

2.4 BLAS Library

The Basic Linear Algebra Subprograms[3] (BLAS) routines provide standard building blocks for performing basic vector and matrix operations [4]. Among its multiple routines, Level 3 BLAS routines perform general matrix-matrix operations as shown on equation 3. Where A, B, C are matrices and α, β vectors.

[3] `http://www.netlib.org/blas`

$$C \leftarrow \alpha AB + \beta C \tag{3}$$

The Automatically Tuned Linear Algebra Software (ATLAS) is a state-of-the-art, open source implementation of BLAS[4]. The library support single and multi-thread execution modes, feature that make advantage of parallelism on systems with multiple processing units.

The `dgemm` BLAS routine performs general matrix-matrix multiplication on double floating-point arithmetic, and the `cblas_dgemm` function call is a C language wrapper around the Fortran implementation. The actual library call is shown below.

```
cblas_dgemm(CblasRowMajor,
              CblasNoTrans,
                CblasNoTrans,
            n, n, n,
            1.0, a, n,
              b, n,
                0.0, c, n);
```

The first three arguments define how the arrays are stored on memory and if any matrix operand needs to be transposed. At last, the vectors and matrices to be used are provided, together with their size.

3 Results

Multiple test cases were launched to exercise the algorithms, a summary of its analysis is presented on this section.

Results were gathered on a system featuring an Intel Core Duo T2400 @ 1.83GHz CPU and also 2GB of DDR2 RAM memory; the software layer included a *Gentoo* system using the 2.6.24 *Linux* kernel and the *GCC* C compiler version 4.1.2.

The output of sample executions for each approach were logged to analyze scaling performance and timing. To calculate the work done, the obtained floating point operations per seconds (FLOPs) were computed. In addition, to get the run-time in seconds the `gettimeofday` function was used as shown below.

[4] `http://netlib.org/atlas`

```
double time;
struct timeval tv;

gettimeofday(&tv,NULL);
time = tv.tv_sec + tv.tv_usec/1000000.0;
```

3.1 Block Size

Choosing the best block size is critical on the blocked approach, this value depends on the presence and quantity of available cache memory.

Figure 1 shows experimental results while changing the block size. Best value matched the number of elements that fit inside each cache line.

To avoid binary search of the best parameter, cache line size may be used to dynamically adjust that setting.

The `sysconf(2)` POSIX interface allows to know this value at run-time; using the following code snippet is enough to find out the best block size on each system without recompilation or passing extra parameters.

```
size = sizeof(double);
cls = sysconf(_SC_LEVEL1_DCACHE_LINESIZE);
bs = cls / size;
```

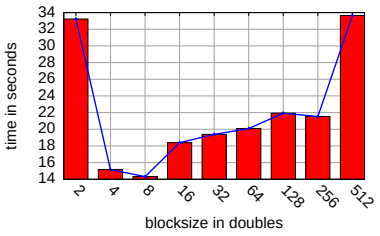

Figure 1: Block Size

3.2 Timings

According to the matrix multiplication definition, complexity will be $O(n^3)$. A better algorithm exists [5]; although it is too complex and only useful on very large sizes, not even suitable for current hardware architectures.

To exercise the implementation, all algorithms were executed for the following matrix sizes: $1024, 2048, 3072, 4096, 5120, 6144, 7168, 8192$.

The figures 3, 4, 5 and 6 show the timings of the simple, block, transp and BLAS algorithms, respectively.

Taking the 8192×8192 matrix case as example, the performance gains based on the simple definition implementation is shown on table 3.2.

method	**seconds**	**gain**
simple	25000	$\times$ 1
blocked	8000	$\times$ 3
transposed	2000	$\times$ 12
blas	500	$\times$ 50

Figure 2: 8192×8192 timings

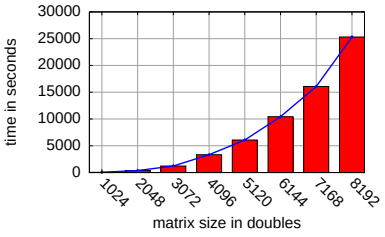

Figure 3: Simple method timings

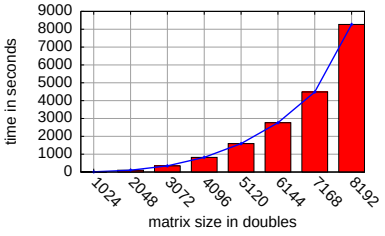

Figure 4: Block method timings

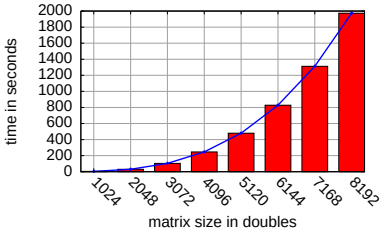

Figure 5: Transposed method timings

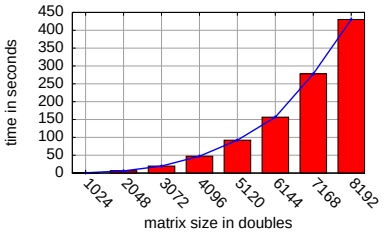

Figure 6: BLAS method timings

All results are summarized on figure 7.

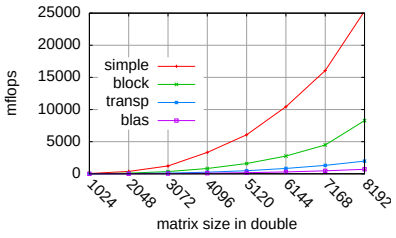

Figure 7: Timings

The exponential grow pattern is clearly depicted on each figure; imposed by the algorithms complexity. However, worst versus best implementations have differences of over 50 times.

3.3 Performance

Besides time, another interesting metric is the actual work performed per time unit.

Matrix multiplication operations can be easily estimated using formula 4, were $n \times n$ is the size of the matrix and t the time used to process it.

$$flops(method) = (n^3 - 2n^2)/t \qquad (4)$$

The table below highlights overall average gains.

method	mflops	gain
simple	50	× 1
blocked	150	× 3
transposed	500	× 10
blas	1500	× 30

Figure 8: Average mflops

Figure 3.3 compares the performance of the implemented approaches. As expected all approaches showed an steady performance, and faster methods reached higher work ratio.

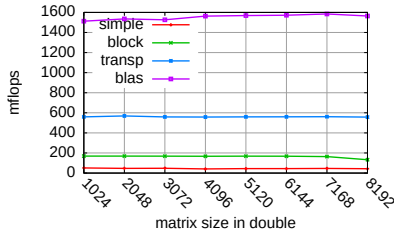

Figure 9: FLOPs Comparison

The results confirm that the system bottleneck is the memory access pattern, not the performance of the available floating point units.

4 Parallelism

Previous approaches used serial algorithms, this section details parallel solutions to the matrix multiplication problem.

4.1 Multi-threading

The ATLAS library can take advantage of multiple processing units, the multiplication is automatically spread among all available cores in the system.

Figure 10 shows a comparison between single and multi threaded BLAS executions.

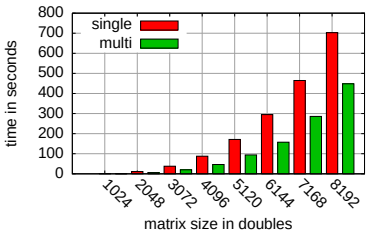

Figure 10: Single vs Multi threaded BLAS

These results shows that the ATLAS implementation has good scaling properties. Taking in account that the system has two processing units. the ideal gain is almost achieved.

4.2 Message Passing Interface

The Message Passing Interface [5] (MPI) standard defines point-to-point and collective communication among a set of processes distributed on a parallel system.

Similar to `socket(7)` programming using the `fork(2)` system call, the same binary is executed multiple times, and the programmer is in charge of the data communication protocol to be followed.

On our case, besides the actual computing of the matrix product, the code must handle the data distribution and gathering of results.

[5] http://www.mpi-forum.org

The data distribution is as follows: each worker process knows which A rows to receive (and compute) given its own task id, after computation results are sent back to the master which is in charge of the accumulation of the results into the result matrix.

The master process implementation is summarized below.

```
for (i = 1; i < size; i++) {
  MPI_Send(&a[offset * n + 0],rows * n,
           MPI_DOUBLE, i, 1, 0);
  MPI_Send(&b[0], n * n,
           MPI_DOUBLE, i, 1, 0);
  offset += rows;
}

for (i = 1; i < size; i++) {
  offset = rows * (i - 1);
  MPI_Recv(&c[offset * n + 0], rows * n,
           MPI_DOUBLE, i, 2, 0, &status)
}
```

The worker process implementation is summarized below.

```
MPI_Recv (&a[0], rows * n,
          MPI_DOUBLE, 0, 1, 0, &status);
MPI_Recv (&b[0], n * n,
          MPI_DOUBLE, 0, 1, 0, &status);

for (k = 0; k < n; k++)
  for (i = 0; i < rows; i++)
    for (j = 0; j < n; j++)
      c[i * n + k] += a[i * n + j]
                    * b[j * n + k];

MPI_Send (&c[0], rows * n,
            MPI_DOUBLE, 0, 2, 0);
```

Although more intended to other types of systems [3], a working implementation using MPI and running on the system under test was achieved. Figure 11 contains the timings gathered on such system.

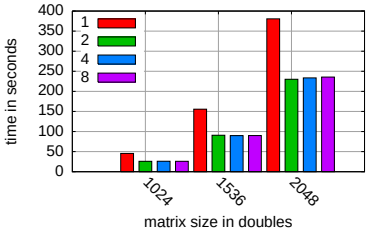

Figure 11: MPI timings according to quantity

The figure shows that using two processes nearly take half of the time. As expected, performance gain is zero when process number is greater than the quantity of processing units.

5 Conclusions

After this work, the following quick conclusions are stated.

- High performance libraries are really well optimized, their outstanding performance against mere mortals' implementations are impressive.
- High performance applications and libraries must be cache-aware, performance is highly dependant on this. To avoid recompilation, run-time information may be used for dynamic optimization.
- Theoretical approaches needs to be reviewed before the actual implementation, as the underlying architecture can alter common assumptions.
- Parallel programming is only hard, not impossible. The implementation of a distributed matrix multiplication algorithm only required the use of basic MPI communication directives and some troubleshooting.
- Memory access delays are the current bottleneck, finding a way to linearly access memory will allow easier prefetching of values and a considerable speed up.

A The `mm` tool

A command line tool was implemented from scratch to gather the timings and performance metrics of the different methods of matrix multiplication[6]. Serial and parallel programming approaches were included in order to analyze scaling and speed up.

The tool was made available under the GPLv2 license. Hopefully, its reuse and source code review will help to understand common issues on high performance computing.

A.1 Source Code

The source code is divided into files as shown in the table below.

file	contents
`src/main.c`	argument processing
`src/mm.c`	matrix multiplication
`src/main.c`	argument processing
`src/utils.h`	reusable definitions
`src/mm.h`	interface to algorithms
`doc/mm.tex`	document source
`doc/mm.gp`	`gnuplot` script

Figure 12: source file contents

Also, the autotools set (`autoconf`, `automake` and `libtool`) was applied for enabling a more portable build system package. The usual `configure; make; make install` is enough for deploying the tool on any supported system.

A.2 Usage

The tool support several methods for matrix multiplication, square matrices are randomly initialized and then multiplied using the requested method.

The arguments accepted include matrix size and the method to apply, optionally a check using best known implementation can be used, and also an specific block size can be requested.

Each execution output reports used time and performance achieved. The results are formatted for being processed as comma-separated value (CSV) registers. The included information follows the following syntax: `method,size,time,mflops,block,processes`.

The `argp` interface, part of the GNU C routines for argument parsing, was used for parameter handling. The tool understands short and long options, for a full list of supported options and parameters the `--usage` switch is available.

```
$ mm --usage
Usage: mm [-cv?V] [-b BLOCK] [--block=BLOCK]
          [--check] [--verbose] [--help]
          [--usage] [--version] SIZE METHOD
```

For instance, launching an execution of the BLAS implementation for two 8192×8192 matrices only requires the following command line. The `--check` option can be used to double check results during the troubleshooting of new algorithms.

```
$ mm 8192 cblas --check
cblas,8192,702.911549,1564.129257,8,1
```

References

[1] F. Tinetti. *Parallel Computing in Local Area Networks*. PhD thesis, Universidad Autonoma de Barcelona, 2004.

[2] Ulrich Deeper. What every programmer should know about memory. Technical report, Red Hat, Inc., November 2007.

[3] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, 1995.

[4] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, September 1979.

[5] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 1990.

[6] `http://code.google.com/p/mm-matrixmultiplicationtool`

C.1.2. Comparación de Implementaciones de una Operación BLAS

Comparación de Implementaciones de una Operación BLAS. Reporte técnico realizado como parte del curso *Programación GPU de Propósito General* dictado por la Dra *Margarita Amor*.

Implementación de BLAS SSPR en GPGPU
Programación GPGPU - UNLP/UDC

Andrés More
more.andres@gmail.com

Resumen

El trabajo consiste en implementar diferentes versiones de la función SSPR de la librería BLAS. El objetivo principal es contrastar el uso de CPUs contra GPUs, tanto en rendimiento como complejidad de implementación.

1. SSPR

Esta función realiza una operación *rank-1 update* en una matriz simétrica con números de punto flotante de precisión simple, una representación gráfica puede verse en la Figura 1.

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \dots & a_{nn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} & \dots & a_{1n} \\ . & & . \\ . & & . \\ . & & . \\ a_{n1} & \dots & a_{nn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix} \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix}$$

Figura 1: Cálculo de SSPR

Esta operación pertenece al nivel 2 de BLAS (Basic Linear Algebra Subprograms), ya que opera sobre una matriz y un vector. Aunque la matriz se define como un vector que contiene la parte superior (o inferior) triangular empacada secuencialmente por columnas.

En particular el elemento A_{ij} en el caso superior se encuentra dentro un vector AP según la ecuación 1. El vector AP tiene entonces un tamaño de $((n \times (n+1))/2)$.

$$AP(i + (j(j-1)/2)) = A_{ij} (\forall j >= i) \tag{1}$$

2. Algoritmos

En concreto se codificaron 4 versiones de la función SSPR, solo se implementó soporte para matrices superiores con desplazaje 1.

2.1. Versión secuencial en CPU

Basado fuertemente en el BLAS original `http://www.netlib.org/blas/sspr.f` y también en el implementado por la librería científica GNU (GSL) en `gsl-X/cblas/source_spr.h`.

```
for (i = 0; i < n; i++) {
   const float tmp = alpha * x[i];

   for (j = 0; j <= i; j++)
       ap[((i*(i+1))/ 2 + j)] += x[j] * tmp;
}
```

Se ve claramente que la cantidad de cómputo por cantidad de datos a transferir no es significante y por lo tanto una implementación GPU no va ser mucho más eficiente que en una CPU.

2.2. Versión en GPU con llamada a cuBLAS

Para utilizar cuBLAS basta una simple llamada como la siguiente.

```
status = cublasSspr(handle, mode, n, &alpha, cx, incx, cap);
    if (status != CUBLAS_STATUS_SUCCESS)
        err("cublasSspr: %d (%s)", status, cublas2str[status]);
```

Este código altamente optimizado puede encontrarse en el paquete *NVIDIA CUBLAS* 1.1 en los archivos `sspr.cu` y `sspr.h`. Para acceder al código hay que inscribirse como desarrollador oficial.

El kernel primero carga en memoria compartida $X[i]$ e $X[j]$, luego procesa varios elementos de la matriz por hilo reusando los mismos. Otro truco que utiliza es realizar operaciones de desplazaje de bits en lugar de división por 2 para calcular ubicaciones dentro de la matriz compactada.

Para utilizar las funciones de soporte recomendadas además de las específicas de CUDA hay que utilizar `cublasCreate()`, y muchas otras como `cublasSetVector()`, `cublasGetVector()`, `cublasDestroy()`; incluyendo manejo de errores con tipos de datos opacos como `cudaError_t`, `cublasHandle_t` y `cublasStatus_t`, `cublasFillMode_t`.

2.3. Versión directa en GPU

La versión directa en GPU utiliza un hilo por cada elemento del vector X computando en paralelo la matriz resultado.

```
__global__ void sspr_naive_kernel(int uplo, int n, float alpha,
                                  const float *x, int incx, float *ap)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        const float tmp = alpha * x[i];

        int j = 0;
        for (j = 0; j <= i; j++)
            ap[((i*(i+1))/ 2 + j)] += x[j] * tmp;
        }
}
```

Para ejecutar el kernel se utiliza una llamada similar al caso anterior.

```
sspr_naive_kernel<<< (n / capabilities.maxThreadsPerBlock),
                (capabilities.maxThreadsPerBlock) >>>
                (uplo, n, alpha, cx, incx, cap);
```

2.4. Versión en GPU con memoria compartida

La primer versión optimizada usa memoria compartida para disminuir el tiempo de acceso a parte del vector X. Todos los hilos de un mismo bloque cargar en memoria compartida un elemento. Luego al utilizar los elementos de X se comprueba si están en memoria compartida o global.

```
__global__ void sspr_optimized_kernel(int uplo, int n, float alpha,
                                      const float *x, int incx, float *ap)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        int tid = threadIdx.x;

        extern __shared__ float cache[];
        float *xi = (float *) cache;
        xi[tid] = x[i];

        __syncthreads();

        const float tmp = alpha * x[i];

        int j = 0;
        for (j = 0; j <= i; j++) {
            if (blockIdx.x * blockDim.x < j \&\&
                blockIdx.x * blockDim.x + 1 > j)
```

```
                ap[((i*(i+1))/ 2 + j)] += xi[j] * tmp;
            else
                ap[((i*(i+1))/ 2 + j)] += x[j] * tmp;
        }
    }
}
```

3. Resultados

El sistema utilizado es una *HP Mobile Workstation EliteBook 8530w* contando con un CPU *Intel T9600* y una tarjeta gráfica *Quadro FX 770m*.

```
cpu family      : 6
model           : 23
model name      : Intel(R) Core(TM)2 Duo CPU T9600 @ 2.80GHz
stepping        : 10
cpu MHz         : 2793
cache size      : 6144 KB
fpu             : yes
cpuid level     : 13
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
                  mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2
                  ss ht tm pbe pni dtes64 monitor ds_cpl vmx smx est tm2
                  ssse3 cx16 xtpr pdcm sse4_1 xsave osxsave lahf_lm
TLB size        : 0 4K pages
```

Se utilizó *CUDA Toolkit 4.2*, publicado en Febrero de 2012. Como dispositivo se utilizó una version mobile de una *Quadro FX 770M*, con una cantidad promedio de 462 MB de memoria global disponible para cálculo.

```
capabilities.name = Quadro FX 770M
capabilities.totalGlobalMem = 512.00 MB
capabilities.sharedMemPerBlock = 16.00 KB
capabilities.regsPerBlock = 8192
capabilities.warpSize = 32
capabilities.memPitch = 2097152.00 KB
capabilities.maxThreadsPerBlock = 512
capabilities.maxThreadsDim = 512 512 64
capabilities.maxGridSize = 65535 65535 1
capabilities.totalConstMem = 64.00 KB
capabilities.major = 1
capabilities.minor = 1
capabilities.clockRate = 1220.70 MHz
capabilities.textureAlignment = 256
capabilities.deviceOverlap = 1
capabilities.multiProcessorCount = 4
cudaMemGetInfo.free = 462 MB
```

En la figura 2 se grafican los tiempos obtenidos en segundos con diferentes tamaños de matriz. Los tiempos son el promedio de 32 ejecuciones de cada

método. Para validar las implementaciones se redujo la matriz a una suma global.

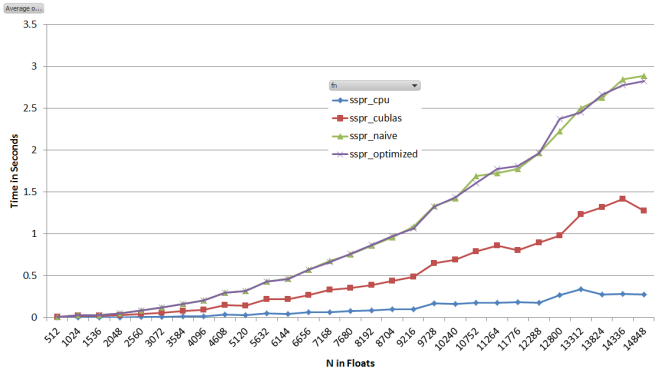

Figura 2: Comparación del Tiempo de Ejecución

Se denota claramente que la optimización realizada no impacta positivamente, por lo que la estrategia que aplica cuBLAS es superior. Sin embargo la versión serial en CPU es aún más eficiente.

4. Análisis de Rendimiento

Se realizo un análisis de rendimiento como para justificar las optimizaciones y entender el rendimiento de cada método.

4.1. Uso de Registros

Primero se ejecutó el compilador *nvcc* con información extra como para controlar la cantidad de registros. La cantidad de registros por thread es mínima.

```
ptxas info: Compiling entry function 'sspr_naive_kernel' for 'sm_13'
ptxas info: Used 7 registers, 48+16 bytes smem, 4 bytes cmem[1]
ptxas info: Compiling entry function 'sspr_optimized_kernel' for 'sm_13'
ptxas info: Used 10 registers, 48+16 bytes smem, 4 bytes cmem[1]
```

Se encontró el problema que esta información no se muestra cuando se especifica `-arch=compute_11` pero sí con `-arch=sm_13`.

4.2. NVIDIA Visual Profiler

En la figura 3 se muestra una captura de la herramienta *nvvp* sobre el código desarrollado.

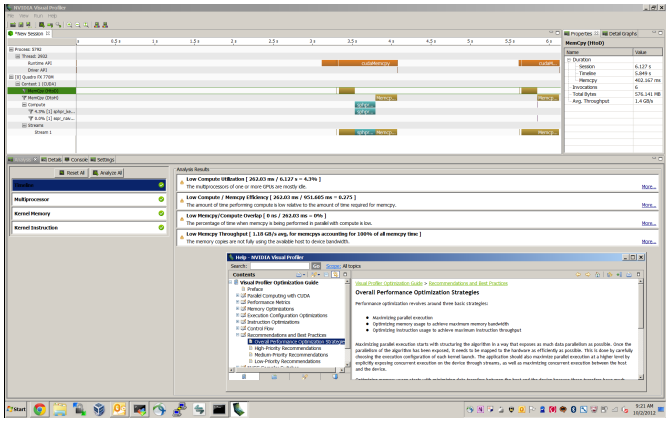

Figura 3: Análisis de Rendimiento

La herramienta identifica que:

- no hay solapamiento de transferencia de datos y cómputo
- no se utiliza todo el ancho de banda durante la transferencia
- no se utiliza completamente la capacidad de cómputo

Estos problemas se deben mayormente a la baja cantidad de memoria disponible en el dispositivo, y al hecho de que debido a la representación compactada de matrices no se realiza suficiente cantidad de cómputo por byte transmitido como para justificar el uso de GPU como acelerador.

5. Aceleraciones

Para calcular las aceleraciones logradas se eligió un tamaño de entrada de casi la totalidad de la memoria libre de la GPU. Las aceleraciones más interesantes son mostradas en la tabla 1.

```
INPUT
-----
SSPR_N = 14848 floats (packed 110238976 floats)
```

```
SSPR_ALPHA = 3.141593
memory = 420 MB
cudaMemGetInfo.free = 462 MB
```

Cuadro 1: Aceleración para N=14848

Método 1	Método 2	Aceleración
cublas (1.4995 seg)	cpu (0.389625 seg)	3.85x
naive (3.090625 seg)	cpu (0.389625 seg)	7.93x
optimized (2.97325 seg)	cpu (0.389625 seg)	7.63x
naive (3.090625 seg)	cublas (1.4995 seg)	2.06x
optimized (2.97325 seg)	cublas (1.4995 seg)	1.98x
optimized (2.97325 seg)	naive (3.090625 seg)	0.95x

6. Conclusión

6.1. Rendimiento

Un estudio relacionado fue realizado por Microsoft Research, comparando el rendimiento de BLAS nivel 2 en FPGA, CPU y GPU. La figura 4 muestra los resultados que convalidan los obtenidos durante este trabajo. Es decir, la cantidad de cómputo no es significante como para justificar la transferencia de datos.

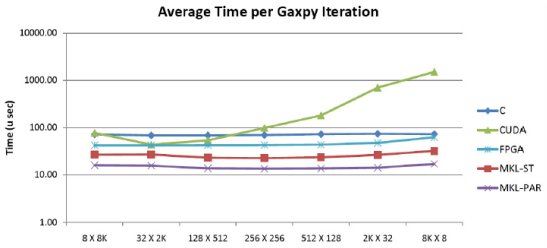

Figura 4: Microsoft Research: BLAS Comparison on FPGA, CPU and GPU

6.2. cuBLAS

El encuentro con la librería cuBLAS no fue muy grato. No hay funciones que se encarguen del ciclo completo de inicialización, transferencia de datos, cálculo y transferencia de resultados. Se necesitan aproximadamente 60 líneas de código para poder ejecutar una llamada BLAS.

cuBLAS no incluye una función para traducir códigos de error como `strerr()` o `cudaGetErrorString()`. La documentación es confusa, por ejemplo dice que no hay

que usar `cudaAlloc()` porque esta deprecada, pero la referencia a `cublas_sspr()` dice que hay que usarla.

Para este trabajo hubo que realizar una migración del código de ejemplo *Linux* a *Windows*, durante el proceso se identificó y resolvió el problema de que `gettimeofday()` no está disponible.

C.2. Artículos en Conferencias

C.2.1. *Optimizing Latency in Beowulf Clusters*

Artículo *Optimizing Latency in Beowulf Clusters*. HPC Latam 2012 [7].

Optimizing Latency in Beowulf Clusters

Rafael Garabato[1], Andrés More[1 2], and Victor Rosales[1]

[1] Argentina Software Design Center (ASDC - Intel Córdoba)
[2] Instituto Universitario Aeronáutico (IUA)

Abstract. This paper discusses how to decrease and stabilize network latency in a Beowulf system. Having low latency is particularly important to reduce execution time of High Performance Computing applications. Optimization opportunities are identified and analyzed over the different system components that are integrated in compute nodes, including device drivers, operating system services and kernel parameters.

This work contributes with a systematic approach to optimize communication latency, provided with a detailed checklist and procedure. Performance impacts are shown through the figures of benchmarks and mpiBLAST as a real-world application. We found that after applying different techniques the default Gigabit Ethernet latency can be reduced from about 50 μs into nearly 20 μs.

Keywords: Beowulf, Cluster, Ethernet, Latency

1 Introduction

1.1 Beowulf Clusters

Instead of purchasing an expensive and high-end symmetric multiprocessing (SMP) system, most scientists today choose to interconnect multiple regular-size commodity systems as a means to scale computing performance and gain the ability to resolve bigger problems without requiring heavy investments [14] [13] [16].

The key driving factor is cost, hence out-of-the-box hardware components are used together with open source software to build those systems. In the specific case of academia, open source software provides the possibility to make software stack modifications, therefore enabling innovation and broadening their adoption.

Clusters are nearly ubiquitous at the Top500 ranking listing most powerful computer systems worldwide, clustered systems represent more than 80% of the list (Figure 1).

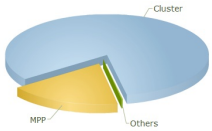

Fig. 1. Top500 System Share by Architecture (as of June 2012)

As the cheapest network fabrics are the ones being distributed on-board by system manufacturers, Ethernet is the preferred communication network in Beowulf clusters. At the moment Gigabit Ethernet is included integrated on most hardware.

1.2 Latency

Latency itself can be measured at different levels, in particular communication latency is a performance metric representing the time it takes for information to flow from one compute node into another. It then becomes not only important to understand how to measure the latency of the cluster but also to understand how this latency affects the performance of High Performance applications [12].

In the case of latency-sensitive applications, messaging needs to be highly optimized and even be executed over special-purpose hardware. For instance latency directly affects the synchronization speed of concurrent jobs in distributed applications, impacting their total execution time.

1.3 Related Work

There are extensive work on how to reduce communication latency [10] [6]. However, this work contributes not with a single component but with a system wide point of view.

The top supercomputers in the world report latencies that commodity systems cannot achieve (Table 1). They utilize specially built network hardware, where the cost factor is increased to get lower latency.

Table 1. Communication Latency at the HPCC ranking

System	**Latency**	**Description**
HP BL280cG65	0.49 μsec	Best Latency
Fujitsu K Computer	6.69 μsec	Top System

High performance network technology (like InfiniBand [2]) is used in cases were state-of-the-art Ethernet cannot meet the required latency (see reference

values in Table 2). Some proprietary network fabrics are built together with supercomputers when they are designed from scratch.

Table 2. System Level Ethernet Latency

Latency	Technology
30-125 μsec	1Gb Ethernet
5-30 μsec	10Gb Ethernet

1.4 Problem Statement

The time it takes to transmit on a network can be calculated as the required time a message information is assembled and dissembled plus the time needed to transmit message payload. Equation 1 shows the relation between these startup plus throughput components for the transmission of n bytes.

$$t(n) = \alpha + \beta \times n \tag{1}$$

In the hypothetical case where *zero bytes* are transmitted, we can get the minimum possible latency on the system (Equation 2). The value of α is also known as the theoretical or zero-bytes latency.

$$t(0) = \alpha \tag{2}$$

It is worth noticing that α is not the only player in the equation, $1/\beta$ is called network bandwidth, the maximum transfer rate that can be achieved. β is the component that affects the overall time as a function of the package size.

2 Benchmarking Latency

There are different benchmarks used to measure communication latency.

2.1 Intel MPI Benchmarks

The Intel MPI Benchmarks (IMB) are a set of timing utilities targeting most important Message Passing Interface (MPI) [7] functions. The suite covers the different versions of the MPI standard, and the most used utility is Ping Pong.

IMB Ping Pong performs a single message transfer exercise between two active MPI processes (Figure 2). The action can be run multiple times using varying message lengths, timings are averaged to avoid measurement errors.

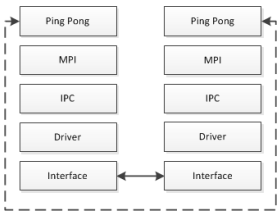

Fig. 2. IMB Ping Pong Communication

Using only MPI basic routines, a package is sent (`MPI_SEND`) from a host system and received (`MPI_RECV`) on a remote one (Figure 3) and the time is reported as half the time in μs for an X long bytes (`MPI_BYTE`) package to complete a round trip.

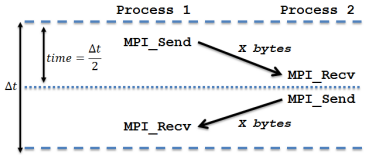

Fig. 3. IMB Ping Pong Benchmark

As described by the time formula at Equation 1, different measures of transmission time are obtained depending on the package size. To get the minimum latency an empty package is used.

2.2 Other Benchmarks

There are other relevant HPC benchmarks that are usually used to exercise clusters: HPL and HPCC. These exercise the system from an application level, integrating all components performance for a common goal.

It is worth mentioning that there are other methods that work at a lower level of abstraction, for instance using Netperf [9] or by following RFC 2544 [3] techniques. However these last two measure latency at network protocol and device level respectively.

High Performance Linpack High Performance Linpack is a portable benchmark for distributed-memory systems doing pure matrix multiplication [1]. It

provides a testing and timing tool to quantify cluster performance. It requires MPI and BLAS supporting libraries.

High Performance Computing Challenge Benchmarks The HPC Challenge benchmark suite [5] packages 7 benchmarks:

HPL: measures floating point by computing a system of linear equations.
DGEMM: measures the floating point rate of execution of double precision real matrix-matrix multiplication.
STREAM: measures sustainable memory bandwidth.
PTRANS: computes a distributed parallel matrix transpose
RandomAccess: measures random updates of shared distributed memory
FFT: double precision complex one-dimensional discrete Fourier transform.
b_eff: measures both communication latency and bandwidth

HPL, DGEMM, STREAM, FFT run in parallel in all nodes, so they can be used to check if cluster nodes are performing similarly. PTRANS, RandomAccess and b_eff exercise the system cluster wide. It is expected that latency optimizations impact their results differently.

3 Methods

Given a simplified system view of a cluster, there are multiple compute nodes that together run the application. An application uses software such as libraries that interface with the operating system to reach hardware resources through device drivers. This work analyzes the following components:

Ethernet Drivers: interrupt moderation capabilities
System Services: interrupt balancing and packet-based firewall
Kernel Settings: low latency extensions on network protocols

Further work to optimize performance is always possible; only the most relevant optimizations were considered according to gathered experience over more than 5 years on the engineering of volume HPC solutions.

3.1 Drivers

As any other piece of software, device drivers implement algorithms which, depending on different factors, may introduce latency. Drivers may even expose hardware functionalities or configurations that could change the device latency to better support the Beowulf usage scenario.

Interrupt Moderation Interrupt moderation is a technique to reduce CPU interrupts by caching them and servicing multiple ones at once [4]. Although it make sense for general purpose systems, this introduces extra latency, so Ethernet drivers should not moderate interruptions when running in HPC clusters.

To turn off Interrupt Moderation on Intel network drivers add the following line on each node of the cluster and reload the network driver kernel module. Refer to documentation [8] for more details.

```
# echo "options e1000e InterruptThrottleRate=0" > /etc/modprobe.conf
# modprobe -r e1000e && modprobe e1000e
```

For maintenance reasons some Linux distributions do not include the configuration capability detailed above. In those cases, the following command can be used to get the same results.

```
# ethtool eth0 rx-usecs
```

There is no portable approach to query kernel modules configurations in all Linux kernel versions, so configuration files should be used as a reference.

3.2 Services

Interrupt Balancing Some system services may directly affect network latency. For instance *irqbalance* job is to distribute interrupt requests (IRQs) among processors (and even between each processor cores) on a *Symmetric Multi-Processing* (SMP) system. Migrating IRQs to be served from one CPU to another is a time consuming task that although balance the load it may affect overall latency.

The main objective of having such a service is to balance between power-savings and optimal performance. The task it performs is to dynamically distribute workload evenly across CPUs and their computing cores. The job is done by properly configuring the IO-ACPI chipset that maps interruptions to cores.

An ideal setup will assign all interrupts to the cores of a same CPU, also assigning storage and network interrupts to cores near the same cache domain. However this implies processing and routing the interrupts before running them, which has the consequence of adding a short delay on their processing.

Turning off the *irqbalance* service will help then to decrease network latency. In a Red Hat compatible system this can be done as follows:

```
# service irqbalance stop
# chkconfig irqbalance off
$ service irqbalance status
```

Firewall As compute nodes are generally isolated on a private network reachable only through the head node, the firewall may not even be required. The system firewall needs to review each package received before continuing with the execution. This overhead increases the latency as incoming and outgoing packet fields are inspected during communication.

Linux-based systems have a firewall in its kernel that can be controlled throughout a user-space application called *iptables*. This application runs in the system as a service, therefore the system's service mechanisms has to be used to stop it.

```
# service iptables stop
# chkconfig iptables stop
$ lsmod | grep iptables
```

3.3 Kernel Parameters

The Linux Transport Control Protocol (TCP) stack makes decisions by default that favors higher throughput as opposed to low latency. The Linux TCP stack implementation has different packet lists to handle incoming data, the PreQueue can be disabled so network packets will go directly into the Receive queue. In Red Hat compatible systems this can be done with the command:

```
# echo 1 > /proc/sys/net/ipv4/tcp_low_latency
$ sysctl -a | grep tcp_low_latency
```

There are others parameters that can be analyzed [15], but the impact they cause are too application specific to be included on a general optimization study.

4 Optimization Impact

4.1 IMB Ping Pong

Using IMB Ping Pong as workload, the following results (Figure 4) reflect how the different optimizations impact communication latency. The actual figures on average and deviation are shown below at Table 3.

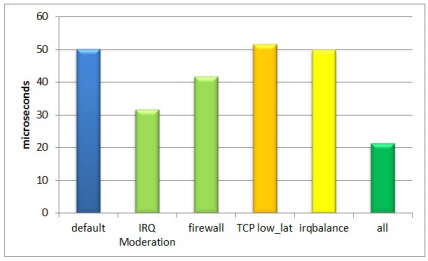

Fig. 4. Comparison of Optimizations

Table 3. IMB Ping Pong Optimization Results

Optimization	$\bar{x}$ (σ^2)	**Impact**
Default	50.03 (4.31)	N/A
IRQ Moderation	31.63 (0.83)	36.79%
Firewall	41.62 (8.90)	16.82 %
TCP LL	51.59 (8.22)	-3.11%
IRQ Balance	49.72 (9.68)	0.62 %
Combined	21.31 (2.09)	57.40 %

The principal cause of overhead in communication latency is then IRQ moderation. Another important contributor is the packet firewall service. We found that the low latency extension for TCP was actually slightly increasing the IMB Ping Pong reported latency. In the case of the IRQ balance service, the impact is only minimal.

Optimizations impact vary, and not surprisingly they are not accumulative when combining them all. At a glance, it is possible to optimize the average latency in nearly 54%, nearly halving result deviations.

4.2 High Performance Linpack

A cluster-wide HPL running over MPI reported results as shown in Table 4. The problem size was customized to `Ns:37326 NBs:168 Ps:15 Qs:16` for a quick but still representative execution with a controlled deviation.

Table 4. HPL Results

Optimization	Wall-time	Gflops
Default	00:20:46	0.02921
Optimized	00:09:03	0.07216

As we can see on the results, the actual synchronization cycle done by the algorithm heavily relies on having low latency. The linear system is partitioned in smaller problem blocks which are distributed over a grid of processes which may be on different compute nodes. The distribution of matrix pieces is done using a binary tree among compute nodes with several rolling phases between them. The required time was then reduced 56%, and the gathered performance was increased almost 2.5 times.

4.3 HPCC

Figure 5 and table 5 show HPCC results obtained with a default and optimized Beowulf cluster. As we can see on the results, the overall execution time is directly affected with a 29% reduction. The performance figures differ across packaged benchmarks as they measure system characteristics that are affected by latency in diverse ways.

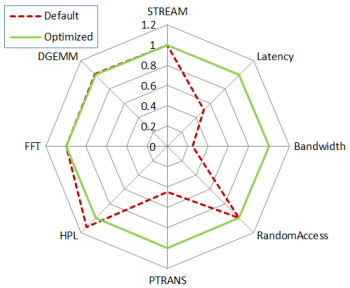

Fig. 5. HPCC Performance Results (higher is better)

Table 5. HPCC Timing Results

Optimization	Wall-time
Default	00:10:32
Optimized	00:07:27

Local benchmarks like STREAM, DGEMM and HPL are not greatly affected, as they obviously do not need communication between compute nodes. However, the actual latency, bandwidth and PTRANS benchmark are impacted as expected due they communication dependency.

4.4 mpiBLAST

In order to double check if any of the optimization have hidden side effects and the real impact on the execution of a full-fledge HPC application, a real-world code was exercised. mpiBLAST [11] is an open source tool that implements DNA-related algorithms to find regions of similarity between biological sequences.

Table 6 shows the actual averaged figures after multiple runs. Results got with a default and optimized system on a fixed workload for mpiBLAST. The required time to process the problem was reduced by 11% with the previous 42% improvement as measured by IMB Ping Pong.

Table 6. mpiBLAST Results

Optimization	**Wall-time**
Default	534.33 seconds
Optimized	475.00 seconds

This shows that the results of a synthetic benchmark like IMB Ping Pong can not be used directly to extrapolate figures, they are virtually the limit to what can be achieved by an actual application.

4.5 Testbed

The experiments done as part of this work were done over 32 nodes with the following bill of materials (Table 7).

Table 7. Compute Node Hardware and Software

Component	**Description**
Server Board	Intel(R) S5000PAL
CPU	Intel(R) Xeon(R) X5355 @ 2.66GHz
Ethernet controller	Intel(R) 80003ES2LAN (Copper) (rev 01)
RAM Memory	4 GB DDR2 FB 667 MHz
Operating System	Red Hat Enterprise 5.5 (Tikanga)
Network Driver	Intel(R) PRO/1000 1.2.20-NAPI
Ethernet Switch	Hewlett Packard HPJ4904A

5 Optimization Procedure

Figure 6 summarizes the complete optimization procedure. It is basically a sequence of steps involving checking and reconfiguring Ethernet drivers and system services if required. Enabling TCP extensions for low latency is not included due their negative consequences.

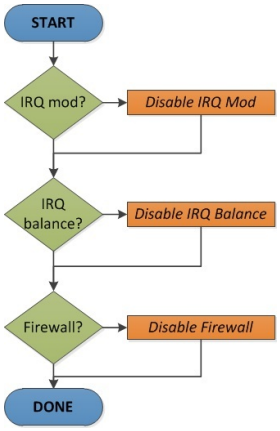

Fig. 6. Latency Optimization Procedure

5.1 Detailed Steps

The steps below include the purpose and an example of the actual command to execute as required on Red Hat compatible systems. The pdsh[3] parallel shell is used to reach compute nodes at once.

Questions (1) helps to dimension the required work to optimize driver configuration to properly support network devices. Questions (2) helps to understand what's needed to properly configure system services.

1. Interrupt Moderation on Ethernet Driver
   (a) Is the installed driver version the latest and greatest?

```
$ /sbin/modinfo -F version e1000e
1.2.20-NAPI
```

[3] `http://sourceforge.net/projects/pdsh`

(b) Is the same version installed across all compute nodes?

```
$ pdsh -N -a '/sbin/modinfo -F version e1000e' | uniq
1.2.20-NAPI
```

(c) Are interrupt moderation settings in HPC mode?

```
# pdsh -N -a 'grep "e1000e" /etc/modprobe.conf' | uniq
options e1000e InterruptThrottleRate=0
```

2. System Services

(a) Is the firewall disabled?

```
# pdsh -N -a 'service iptables status' | uniq
Firewall is stopped.
```

(b) Is the firewall disabled at startup?

```
# pdsh -N -a 'chkconfig iptables --list'
irqbalance 0:off 1:off 2:off 3:off 4:off 5:off 6:off
```

(c) Was the system rebooted after stopping firewall services?

```
$ uptime
 15:42:29 up 18:49, 4 users, load average: 0.09, 0.08, 0.09
```

(d) Is the IRQ balancing service disabled?

```
# pdsh -N -a 'service irqbalance status' | uniq
irqbalance is stopped
```

(e) Is IRQ balancing daemon disabled at startup?

```
# pdsh -N -a 'chkconfig irqbalance --list' | uniq
irqbalance 0:off 1:off 2:off 3:off 4:off 5:off 6:off
```

Once gathered all the information required to known if optimizations can be applied, the following list can be used to apply configuration changes. Between each change a complete cycle of measurement should be done. This include contrasting old and new latency average plus their deviation using at least IMB Ping Pong.

Disable IRQ Moderation

```
# pdsh -a 'echo "options e1000e InterruptThrottleRate=0" >> \
/etc/modprobe.conf'
# modprobe -r e1000e; modprobe e1000e
```

Disable IRQ Balancer

```
# pdsh -a 'service irqbalance stop'
# pdsh -a 'chkconfig irqbalance off'
```

Disable Firewall

```
# pdsh -a 'service iptables stop'
# pdsh -a 'chkconfig iptables off'
```

6 Conclusion

This work shows that by only changing default configurations the latency of a Beowulf system can be easily optimized, directly affecting the execution time of High Performance Computing applications. As a quick reference, an out-of-the-box system using Gigabit Ethernet has around 50 μs of communication latency. Using different techniques, it is possible to get as low as nearly 20 μs.

After introducing some background theory and supporting tools, this work analyzed and exercised different methods to measure latency (IMB, HPL and HPCC benchmarks). This work also contrasted those methods and provided insights on how they should be executed and their results analyzed.

We identified which specific items have higher impact over latency metrics (interrupt moderation and system services), using de-facto benchmarks and a real-world application such as mpiBLAST.

6.1 Future Work

Running a wider range of real-world computational problems will help to understand the impact in different workloads. A characterization of the impact according to the application domain, profiling information or computational kernel might be useful to offer as a reference.

There are virtually endless opportunities to continue with the research on latency optimization opportunities; among them components like BIOS, firmware, networking switches and routers. An interesting opportunity are the RX/TX parameters of Ethernet drivers that control the quantity of packet descriptors used during communication.

Another option is to implement an MPI trace analysis tool to estimate the impact of having an optimized low latency environment. At the moment there are several tools to depict communication traces (Jumpshot[4], Intel's ITAC[5]), but they do not provide a simulation of what would happen while running over a different network environment. Having this approximation can be useful to decide if it is worth to purchase specialized hardware or not.

At last, it would be interesting also to understand the impact of this work into research or development processes using clusters, not only in industry but also in academia.

Acknowledgments

The authors would like to thanks the Argentina Cluster Engineering team at the Argentina Software Design Center (ASDC Intel) for their contributions.

[4] `http://www.mcs.anl.gov/research/projects/perfvis/software/viewers`
[5] `http://software.intel.com/en-us/articles/intel-trace-analyzer`

References

1. J. Dongarra A. Petitet, R. C. Whaley and A. Cleary. A portable implementation of the high-performance linpack benchmark for distributed-memory computers. Technical report, 2008.
2. Infiniband Trade Association. Infiniband architecture specification release 1.2.1. Technical report, 2008.
3. S. Bradner and J. McQuaid. Ieee rfc2544: Benchmarking methodology for network interconnect devices, 1999.
4. Intel Corporation. Interrupt Moderation Using Intel Gigabit Ethernet Controllers Application Note . Technical report, 2007.
5. Jack J. Dongarra, I. High, and Productivity Computing Systems. Overview of the hpc challenge benchmark suite, 2006.
6. A.P. Foong, T.R. Huff, H.H. Hum, J.R. Patwardhan, and G.J. Regnier. Tcp performance re-visited. In *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*, pages 70 – 79, march 2003.
7. Message Passing Interface Forum. Mpi: A message-passing interface standard. Technical report, 2009.
8. Improving Measured Latency in Linux for Intel(R) 82575/82576 or 82598/82599 Ethernet Controllers. Interrupt moderation using intel gigabit ethernet controllers application note. Technical report, 2009.
9. R. Jones. Netperf, 2007.
10. S. Larsen, P. Sarangam, and R. Huggahalli. Architectural breakdown of end-to-end latency in a tcp/ip network. In *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*, pages 195 –202, oct. 2007.
11. Heshan Lin, Pavan Balaji, Ruth Poole, Carlos Sosa, Xiaosong Ma, and Wu-chun Feng. Massively parallel genomic sequence search on the blue gene/p architecture. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 33:1–33:11, Piscataway, NJ, USA, 2008. IEEE Press.
12. QLogic. Introduction to ethernet latency. Technical report, 2011.
13. John Salmon, Christopher Stein, and Thomas Sterling. Scaling of beowulf-class distributed systems. In *In Proceedings of SC'98*, 1998.
14. Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *In Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.
15. Assigning Interrupts to Processor Cores using an Intel(R) 82575/82576 or 82598/82599 Ethernet Controller. Interrupt moderation using intel gigabit ethernet controllers application note. Technical report, 2009.
16. Ewing Lusk William Gropp and Thomas Sterling. *Beowulf Cluster Computing with Linux, Second Edition*. The MIT Press, 2003.

C.2.2. *Lessons Learned from Contrasting BLAS Kernel Implementations*

Artículo *Lessons Learned from Contrasting BLAS Kernel Implementations* - XIII Workshop Procesamiento Distribuido y Paralelo (WPDP), 2013 [9].

Lessons learned from contrasting a BLAS kernel implementations

Andrés More[1 2]

[1] Intel Software Argentina (Argentina Software Design Center)
andres.more@intel.com
[2] Instituto Aeronáutico Córdoba
amore@iua.edu.ar

Abstract. This work reviews the experience of implementing different versions of the SSPR rank-one update operation of the BLAS library. The main objective was to contrast CPU versus GPU implementation effort and complexity of an optimized BLAS routine, not considering performance. This work contributes with a sample procedure to compare BLAS kernel implementations, how to start using GPU libraries and offloading, how to analyze their performance and the issues faced and how they were solved.

Keywords: BLAS libraries, SSPR kernel, CPU architecture, GPU architecture, Performance analysis, Performance measurement, Software Optimization.
XIII Workshop de Procesamiento Distribuido y Paralelo

1 Introduction

With the growth of the application of general-purpose GPU techniques to compute intensive problems [**?**], there will be lots of domain experts trying to implement an specific math kernel operation both in CPU and GPU, applying optimizations and finally doing a performance comparison to double check if gains are relevant due the invested effort [**?**].

It is non trivial to identify the outcome. While CPUs are designed for general purpose, GPUs are designed for specific purposes; specific problems are well suited to GPU architecture and can be solved faster than using CPUs, but they usually need to be represented in a way parallelism is explicitly exposed. Picking the right approach will contribute to faster, more detailed problem solving in domain specific problems. It is expected that the procedure of reviewing an specific kernel implementations will be done on each release of new CPU and GPU architecture by people doing problem solving simulations. From that point of view this work contributes with a similar analysis experience. Related work is included as part of the results as a validation proof.

The rest of the document is structured as follows: this section is completed with information on the operation and system being used for the comparison, plus a quick discussion on the GPU programming model. Section 2 include details on which algorithms were executed after reviewing well-known implementations. Section 3 provide details on how the performance was contrasted, including results and related work validating our results.

1.1 Single-precision Symmetric Packed Rank-one update

The Single-precision Symmetric Packed Rank-one update (SSPR) operation computes a rank-1 update of a symmetric matrix of single precision floating point numbers. SSPR performs the symmetric rank 1 operation show in Equation 1, where α is a real scalar, x is an n element vector and A is an n by n symmetric matrix, supplied in packed form.

$$A := \alpha \times x \times x^T + A \tag{1}$$

A graphical representation of the operation is shown in Figure 1.

$$\begin{bmatrix} a_{11} \dots a_{1n} \\ . \quad . \\ . \quad . \\ . \quad . \\ a_{n1} \dots a_{nn} \end{bmatrix} \leftarrow \begin{bmatrix} a_{11} \dots a_{1n} \\ . \quad . \\ . \quad . \\ . \quad . \\ a_{n1} \dots a_{nn} \end{bmatrix} + \alpha \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix} \begin{bmatrix} x_1 \dots x_n \end{bmatrix}$$

Fig. 1. Graphical representation of rank 1-update

The SSPR operation belongs to level 2 BLAS (Basic Linear Algebra Subprograms) [**?**] [**?**], as it runs over a matrix-vector combination. Matrix-vector multiplication has $O(N^2)$ complexity on the size of the matrix [**?**]. However, the matrix is required to be represented as a vector which contains only half of the symmetric matrix triangular, packed sequentially per column. Represented by Equation 2. In particular, the vector has a size of $(n \times (n+1)/2)$. This approach avoids redundancy and saves device memory for bigger input.

$$AP(i + (j \times (j-1)/2) = A_{ij}(\forall j \geq i) \tag{2}$$

Operation signature in the Fortran language is defined in Listing 1 and clarified in Table 1 below. Towards generality of the function, UPLO specifies if the packed triangular matrix is the upper or the lower part of the original data. INCX is the required increment to reference the vector elements in the provided vector reference. This is useful to iterate over a vector which is part of a matrix, avoiding extra buffer copies.

Listing 1. SSPR Fortran Signature

```
1  SUBROUTINE SSPR(UPLO, N, ALPHA, X, INCX, AP)
2          .. Scalar Arguments ..
3          REAL ALPHA
4          INTEGER INCX,N
5          CHARACTER UPLO
6          ..
7          .. Array Arguments ..
8          REAL AP(*),X(*)
```

Table 1. SSPR Arguments Description

Argument	**Description**
UPLO	Specifies whereas upper/lower triangular part of A is supplied in array
N	Specifies the order of the matrix A
ALPHA	Specifies the scalar alpha
X	Array of dimension at least $(1 + (n-1) * abs(INCX))$
INCX	Specifies the increment for the elements of X
AP	Array of dimension at least $((n * (n+1))/2)$

1.2 Testbed

The system used for experiments was an *HP Mobile Workstation EliteBook 8530w*, having integrated an *Intel CPU model T9600* [3] plus a *Quadro FX 770m GPU* [4]. Although the system is a little bit outdated and it is not state-of-the-art, the computing devices were integrated at the same time-frame and hence provide a valid point of comparison.

The relevant information from their hardware specifications are shown below. The CPU information was taken from `/proc/cpuinfo` and a custom program to access GPU attributes not yet exposed thru standard kernel interfaces.

```
cpu family: 6
model: 23
model name: Intel(R) Core(TM) 2 Duo CPU T9600 @ 2.80GHz
stepping: 10
cpu MHz: 2793
cache size: 6144 KB
fpu: yes
cpuid level: 13
flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr
pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2
ss ht tm pbe pni dtes64 monitor ds_cpl vmx smx est tm2
ssse3 cx16 xtpr pdcm sse4_1 xsave osxsave lahf_lm
TLB size: 0 4K pages
```

[3] T9600 CPU specifications
[4] Quadro FX 770m GPU specifications

The GPU card has built-in 512MB memory, meaning that in average during executions there is about 462 MB of global memory available for computation.

```
capabilities.name = Quadro FX 770M
capabilities.totalGlobalMem = 512.00 MB
capabilities.sharedMemPerBlock = 16.00 KB
capabilities.regsPerBlock = 8192
capabilities.warpSize = 32
capabilities.memPitch = 2097152.00 KB
capabilities.maxThreadsPerBlock = 512
capabilities.maxThreadsDim = 512 512 64
capabilities.maxGridSize = 65535 65535 1
capabilities.totalConstMem = 64.00 KB
capabilities.major = 1
capabilities.minor = 1
capabilities.clockRate = 1220.70 MHz
capabilities.textureAlignment = 256
capabilities.deviceOverlap = 1
capabilities.multiProcessorCount = 4
cudaMemGetInfo.free = 462 MB
```

As depicted in Figure 2, the different in single-precision (real) floating point operations is significant. It might be expected results that choose GPUs as the winner, which will be a different assumption if the operation was using double precision floating point operations.

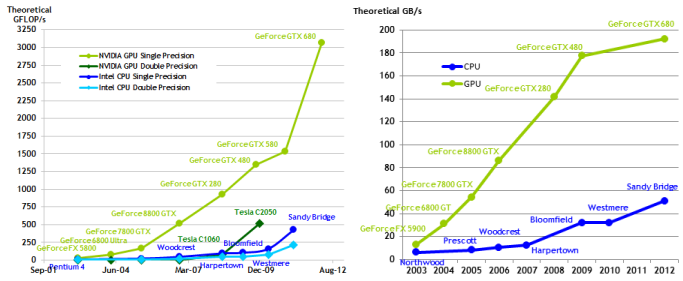

Fig. 2. FLOPs and Bandwidth Comparison between CPUs and GPUs [**?**]

1.3 CUDA Programming

The Compute Unified Device Architecture (CUDA) is a high performance computing architecture and programming model created by NVIDIA. The CUDA programming model gives access to GPUs instruction set to be used for general purpose computing, providing both low level and high level interfaces to simplify application.

The programming model assumes that the host system will execute kernels by asynchronous offloading to a GPU device. Memory allocation and transfer are also controlled by the developer. The programming model relies on a hierarchy of thread groups that can access per-group shared memory and can synchronize between them. An extension to the C language allows the inclusion of compute kernels that run in the GPU device. Hardware threads are part of an N-dimensional block with `N=1,2,3`.

All threads on a block are expected to share (faster than global) memory resources and to synchronize with each other. Blocks per-se are also organized in sets called grids, similar to blocks. Thread blocks perform independent computation. Each block memory hierarchy consists of local, global, shared, constant and texture memory; the latter having special addressing to better fit graphical rendering. All but shared memory is persistent across kernel executions.

GPU hardware and software capabilities are evolving fast, not only instructions per cycle have increased. Support for native arithmetic instructions have expanded the set of atomic operations and the provided math functions. The usual

`printf` capability has been recently made available, the runtime uses a special device buffer that it is transferred together with the device memory. It is worth noting that IEEE 754-2008 binary floating-point arithmetic has deviations from standard by default. Enabling strict support imposes a performance penalty.

2 Algorithms

As part of the work 4 different versions of SSPR functions were exercised, using well-known implementations as a reference. The following subsections contains details on how the versions were implemented that can be reused for any other BLAS kernel. In order to streamline analysis only support for upper matrices with 1 increments were incorporated.

2.1 CPU Sequential

Using both the original BLAS implementation [5] and the GNU Scientific Library version [6] an initial CPU sequential version was implemented as shown in Listing 2. It is worth to note that any BLAS kernel can be reviewed also on this two implementations. A naive implementation of the mathematical definition was not used as proper speedup computation requires best known sequential algorithm [**?**], shown in Listing 3.

Listing 2. Naive SSPR CPU Implementation

```
k=0;
for (j=0;j<n;j++)
        for(i=0;i<=j;i++) {
                ap[k] += alpha*x[i]*x[j];
                k++;
}
```

Listing 3. Optimized SSPR CPU Implementation

```
for (i = 0; i < n; i++) {
        const float tmp = alpha * x[i];

        for (j = 0; j <= i; j++)
                ap[((i * (i+1)) / 2+j )] += x[j] * tmp;
}
```

Here it can be estimated that the required computation per data quantity is not enough to justify accelerator offload time. It is expected then that a GPU version of it is not going to have huge performance increments for small data.

2.2 GPU cuBLAS

This implementation was done directly reusing CUDA source code. NVIDIA CUDA [**?**] provides its own version of BLAS routines, which are heavily optimized to their architecture. Using the *cuBLAS* [**?**] library requires one call, an example is shown in Listing 4.

Listing 4. cuBLAS SSPR GPU Implementation

```
ret = cublasSspr(handle, mode, n, &alpha, cx, incx, cap);
if (ret != CUBLAS_STATUS_SUCCESS)
        err("cublasSspr: %d (%s)", ret, cublas2str[ret]);
```

This highly optimized code can be found in the package available to registered developers, inside `sspr.cu` and `sspr.h` files. The main routine is `cublasSspr()`. This implementation first loads into device shared memory elements reused during computation, then computes several matrix elements for hardware thread. It also uses cheap left bit-shifting instructions instead of expensive division-by-2 to locate elements inside the packed matrix.

The library documentation recommends the use of on utilities like: `cublasCreate()`, `cublasSetVector()`, `cublasGetVector()`, `cublasDestroy()`. They provide easier allocation of memory on the GPU device. The library also defines opaque data-types for parameters and error handling such as: `cudaError_t`, `cublasHandle_t`, `cublasStatus_t`, `cublasFillMode_t`.

[5] BLAS SSPR implementation.
[6] GSL SSPR implementation.

2.3 GPU Naive

A naive GPU implementation is useful to have a worst-estimate to be compared against potential optimizations. A direct translation to GPU using one thread per vector element to computing the result in parallel is shown in Listing 5. Here it is assumed that the number of elements in x it is close to the number of GPU threads.

Listing 5. Naive SSPR GPU Implementation

```
__global__ void sspr_naive_kernel(int uplo, int n, float alpha, const float *x, int incx, float *ap) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < n) {
                const float tmp = alpha * x[i];
                int j = 0;
                for (j = 0; j <= i; j++)
                        ap[((i*(i+1))/ 2 + j)] += x[j] * tmp;
        }
}
```

How to execute the kernel is shown in Listing 6. CUDA will run a preprocessor transforming the code before performing actual compilation.

Listing 6. GPU Kernel execution

```
int threads = capabilities.maxThreadsPerBlock;
sspr_naive_kernel <<< (n / threads), (threads) >>> (uplo, n, alpha, cx, incx, cap);
```

2.4 GPU using shared-memory

The recommended approach to start optimizing GPU code is to use shared memory to reduce access time to data. Every thread on the same thread block loads in shared memory one element of the vector, this work is done in parallel and a barrier is used to synchronize. During computation, elements are then gathered from faster shared memory when possible, slower global memory is used otherwise. The implementation is shown in Listing 7.

Listing 7. Optimized SSPR GPU Implementation

```
__global__ void sspr_optimized_kernel(int uplo, int n, float alpha, const float *x, int incx, float *ap) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < n) {
                int tid = threadIdx.x;
                extern __shared__ float cache[];
                float *xi = (float *) cache;
                xi[tid] = x[i];
                __syncthreads();
                const float tmp = alpha * x[i];
                int j = 0;
                for (j = 0; j <= i; j++) {
                        if (blockIdx.x * blockDim.x < j &&       blockIdx.x * blockDim.x + 1 > j)
                                ap[((i*(i+1))/ 2 + j)] += xi[j] * tmp;
                        else
                                ap[((i*(i+1))/ 2 + j)] += x[j] * tmp;
                }
        }
}
```

However, this method does not take into account any other potential optimization, it is included here to show that naive optimizations are not preferred and it is more useful to build upon CUDA implementation source code.

3 Performance Analysis

This section includes details on how the performance of the different implementations were conducted. The tools and procedure can be reused for any other BLAS kernel without major modifications.

3.1 GPU Registers

The CUDA `nvcc` compiler can be configured to show extra information about the number of registers used during execution. With this option the code can be optimized so register usage metric is minimal.

```
ptxas info: Compiling entry function 'sspr_naive' for 'sm_13'
ptxas info: Used 7 registers, 48+16 bytes smem, 4 bytes cmem[1]
ptxas info: Compiling entry function 'sspr_optimized' for 'sm_13'
ptxas info: Used 10 registers, 48+16 bytes smem, 4 bytes cmem[1]
```

It was discovered that running with the `-arch=compute_11` option did not included this output, but using `-arch=sm_13` instead solved the issue. The first one uses the compute capability version to specify the hardware target, while the second uses the hardware architecture generation.

3.2 Nvidia Visual Profiler

The `nvvp` tool depicts a visual execution profile of the program, useful when trying to understand where the computation is spending execution time. On our case, even with *cuBLAS* implementation the tool identified:

- there is no overlap between data transfer and computation
- the data transfer action does not fully saturate available memory bandwidth
- the computation does not fully load processing cores capacity

The affected GPU device does not have enough internal memory to run an interesting enough problem, the required computation per transfered bytes does not justify GPU offloading.

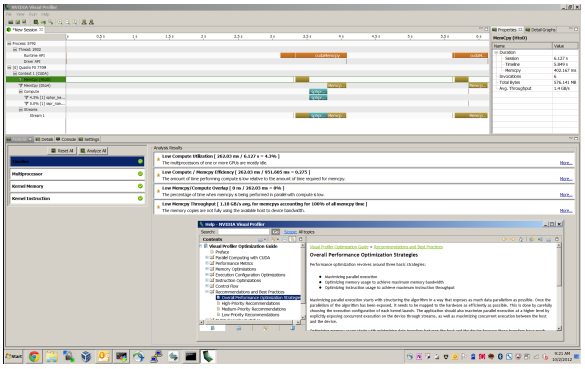

Fig. 3. GPU Profiler

3.3 Results

Figure 4 shows wall clock times in seconds with varying matrix sizes. [7] Here it can be seen that the CPU optimized version is taking the least time on all of the matrix sizes. On the other hand we double check that the GPU naive and shared-memory optimized versions are not a match against the CUDA one provided by cuBLAS. Here it can be clearly confirmed that the introduced shared memory optimization do not positively impact execution time, so *cuBLAS* strategy is hence superior.

[7] Times are the geometric mean of 32 executions in order to reduce measurement noise. To validate results the output matrix was reduced to a single figure being the common global sum of the elements.

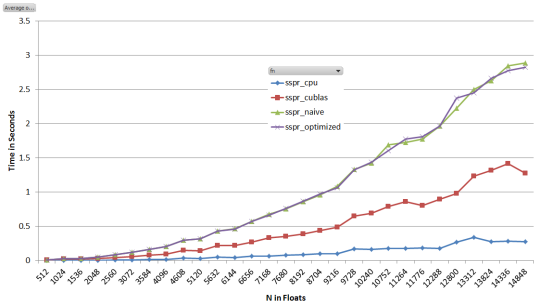

Fig. 4. Execution Time Speedup

3.4 Speedups

To measure speedups the input size was selected as big as possible to fit inside the available memory in GPU. This is always recommended to maximize computation per transfered byte. The time taken to transfer the data to the GPU is included on the measurements, the goal was to contrast the complete operation execution from start to end.

```
SSPR_N = 14848 floats (packed 110238976 floats)
SSPR_ALPHA = 3.141593
memory = 420 MB
cudaMemGetInfo.free = 462 MB
```

Most interesting speedup comparisons are shown in Table 2. The optimized CPU version has nearly 4x of cuBLAS version, and close to 8x of our naive implementations using GPU. It is interesting to note that cuBLAS optimization got 2x speedup when matched against our naive optimization with shared memory.

Table 2. Speedup comparisons

cublas (1.4995 seg)	cpu (0.389625 seg)	3.85x
naive (3.090625 seg)	cpu (0.389625 seg)	7.93x
optimized (2.97325 seg)	cpu (0.389625 seg)	7.63x
naive (3.090625 seg)	cublas (1.4995 seg)	2.06x
optimized (2.97325 seg)	cublas (1.4995 seg)	1.98x
optimized (2.97325 seg)	naive (3.090625 seg)	0.95x

3.5 Related Work

There is a related study conducted by Microsoft Research [**?**], that performed benchmarking of BLAS Level 2 routines in FPGA, CPU and GPU. Their findings in Figure 5 validates the results obtained as part of this work. An an optimized implementation in CPU is better than an optimized GPU implementation. Note that *Intel Math Kernel* library version is still far better than an optimized CPU version, as it uses advanced knowledge of the architecture of computing units inside the CPU.

It is worth to note that state-of-the-art GPU devices have increased their internal memory to cope with this limitation, up to 4GB in latest 2012 boards. If ECC is enabled to verify contents then this quantity is decreased 10%. A detailed analysis on how to better exploit GPU performance is reviewed in [**?**] using a complete application as case study.

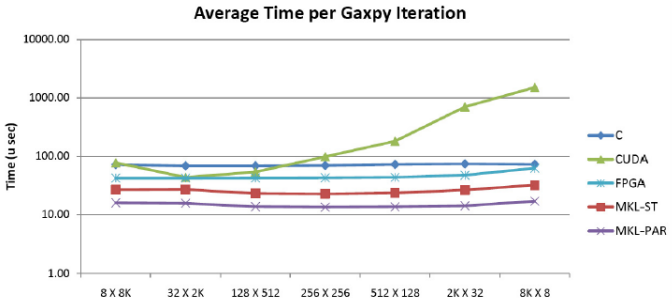

Fig. 5. Independent measurement of matrix-vector kernels (extract from [**?**])

4 Conclusions

This works provides a sample procedure to contrast BLAS kernel implementations after a experience with the SSPR operation. Source code pointers details are provided that guides through well-known implementation, also include performance analysis tools and their application example. In order to gather performance figures, it is always recommended to review the optimized code of BLAS instead of doing naive implementations. It is also worth to note that efficient GPU offload requires significant amounts of required computation per transfered byte. In this case, matrix-vector kernel computation showed that CPUs provide better results than GPUs; even when using highly optimized implementations of the kernel operation.

Regarding the experience with CUDA, the cuBLAS library documentation and interface properly support development, although they can still be improved. Some utility calls like `cublasAlloc()` are deprecated but still referenced by `cudaSspr()` and others, confusing the reader. The library does not provide a wrapper call that goes over the complete offload cycle: initialization, data transference to and from accelerator, explicit computation, etc. Using the documented primitives plus required error checking implies nearly 60 lines of code, just to offload one BLAS routine. cuBLAS also lacks an error formatter function to translate error codes to text representations (similar to `strerr()`). If it is required to support both Linux and Windows environments, the usual time keeping routines are not portable so a `gettimeofday()` stub was coded that could have been provided by the CUDA runtime instead.

Regarding further work, developing a framework to quickly benchmark compute kernels on different processing devices will be of value to domain experts researching what type of hardware to acquire. Ideally, including support for state-of-the-art BLAS implementations to provide figures from optimized algorithms. Also extending the comparison will be a good initial step, including other parallel programming techniques and technologies like OpenMP and MPI. Including results from other devices like FPGAs and co-processors would be another interesting option. There are other GPU optimizations that although require advanced knowledge (i.e. overlapping communication and computation, using texture memories) may result in better performance figures. Upcoming GPU architectures having more memory or new hardware-assisted features (i.e. zero-copy data replication) may also show different results.

C.2.3. *Hotspot: a Framework to Support Performance Optimization on Multiprocessors*

Artículo *Lessons Learned from Contrasting BLAS Kernel Implementations* - PDPTA'15: The 2015 International Conference on Parallel and Distributed Processing Techniques and Applications [10].

Hotspot: a Framework to Support Performance Optimization on Multiprocessors

Fernando G. Tinetti[1], **Andres More**[2]
[1]III-LIDI, Fac. De Informática, UNLP
Comisión de Inv. Científicas Prov. de Bs. As.
1900, La Plata, Argentina
[2]Intel Corp., Argentina Software Design Center
5000, Córdoba, Argentina

Abstract— *High Performance Computing programs are usually developed by domain specialists, and they are not always experts on performance optimization. Those experts usually have to rely on a systematic and unattended (and/or partially guided) method to support performance analysis and optimization. This paper presents a framework that simplifies the job, including several case studies as a validation step and as a proof of concept. The main objective is to make the data recollection task straight-forward, allowing to focus on experimentation and optimization. The framework gathers context information about the program and the underlying system, detailing scaling behavior, execution profile, resource utilization, and bottlenecks. In a nutshell, this work contributes with a performance report generator for OpenMP programs.*

Keywords: Computer Aided Analysis, Performance Analysis, High Performance Computing, Parallel Processing

1. Introduction

High Performance Computing (HPC) optimized code can run several orders of magnitude faster than a naïve implementation [1]. This implies that investing on program parallelization and optimization can lead to large gains in productivity for processing intensive programs, which usually require multiple runs to simulate or solve a problem.

Parallel programming is used to fully take advantage of available computing power; incrementing both complexity and the required effort when implementing, debugging and optimizing any program [2]. The optimization and parallelization tasks are usually performed ad-hoc, without knowledge of available tools and their capabilities. Lacking the use of quantitative information to direct optimizations is another (not minor) problem underlying such parallelization and optimization work. Many times, well-known algorithms are (re)implemented instead of using already heavily optimized libraries, with proven correctness and a supporting community.

This paper is focused on simplifying the tedious and error-prone task of performance analysis. The methodology implies repetitive execution of multiple executions under different input conditions, taking advantage of the expertise on different tools to instrument behavior and understand resource utilization internals. The framework runs benchmarks on the system, apply profiling tools, and graphically show results into a final detailed report with statistical data on scaling and bottlenecks.

The rest of this paper is structured as follows: in Section 2 we introduce performance analysis background and related work. Section 3 the problem we aim to solve is reviewed, and in Section 4 we cover our proposed solution and the implemented framework. Finally, in Section 5, we apply the framework to several well-known computing kernels before concluding remarks are discussed.

1.1 Related Work

Performance optimization is relevant to almost any discipline involving computing processing. An introduction can be found in [12], a general revision in [13], just to mention a few. A proposal of the direction of the state-of-the-art in [14]. Useful reus-able design patterns in [15]. Several ideas for gathering relevant performance information can be found in [16], and an effort on automatic optimization in [17]. Further details on the state-of-the-art can be found in [18].

2. Performance Analysis

Performance is characterized by a metric which must be quantifiable units of achieved work in contrast with utilized time and resources. Metrics allows the relative comparison of systems, and also understand a reconfigured system behavior.

There are several laws that provide some guidance for estimating performance gains when incrementing computing resources. The so called Amdahl Law [7] pro-vides some insight in potential speedup according to the serial and parallel parts of an program. Gustafson [8] establishes something similar, but taking into account how many times we can compute the same problem and, also, the fact that increasing processing facilities implies increasing problem size (thus, also increasing processing requirements). A *raw* representation on how parallelism ratios impact speedup when doubling processing units is shown in Fig. 1.

	0.1	0.3	0.5	0.8	0.9	0.95
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.05	1.14	1.33	1.60	1.82	1.90
4	1.08	1.23	1.60	2.29	3.08	3.48
8	1.10	1.28	1.78	2.91	4.71	5.93
16	1.10	1.31	1.88	3.37	6.40	9.14
32	1.11	1.32	1.94	3.66	7.80	12.55
64	1.11	1.33	1.97	3.82	8.77	15.42
128	1.11	1.33	1.98	3.91	9.34	17.41
256	1.11	1.33	1.99	3.95	9.66	18.62
512	1.11	1.33	2.00	3.98	9.83	19.28
1024	1.11	1.33	2.00	3.99	9.91	19.64
2048	1.11	1.33	2.00	3.99	9.96	19.82
4096	1.11	1.33	2.00	4.00	9.98	19.91
8192	1.11	1.33	2.00	4.00	9.99	19.95
16384	1.11	1.33	2.00	4.00	9.99	19.98
32768	1.11	1.33	2.00	4.00	10.00	19.99
65536	1.11	1.33	2.00	4.00	10.00	19.99

	0.1	0.25	0.5	0.75	0.9	0.95
1	1	1	1	1	1	1
2	1	1	2	2	2	2
4	1	2	3	3	4	4
8	2	3	5	6	7	8
16	3	5	9	12	15	15
32	4	9	17	24	29	30
64	7	17	33	48	58	61
128	14	33	65	96	115	122
256	27	65	129	192	231	243
512	52	129	257	384	461	486
1024	103	257	513	768	922	973
2048	206	513	1025	1536	1843	1946
4096	411	1025	2049	3072	3687	3891
8192	820	2049	4097	6144	7373	7782
16384	1639	4097	8193	12288	14746	15565
32768	3278	8193	16385	24576	29491	31130
65536	6555	16385	32769	49152	58983	62259

Fig. 1: Speedup Limits According to Amdahl (left) and Gustafson (right)

The procedure to perform optimizations involves cycles of measurement, bottleneck identification, optimization, and gain comparison and analysis. Every decision should be made using strongly meaningful data in order to focus the work on maximizing impact. In the case the measurement of a metric has deviations, it is required to take several samples and use an average to avoid transient noise. Usually, a longer execution time of a problem will help to stabilize results. If the system has dynamic configuration, then results reproduction is non-trivial. On this case, it is advisable to run together old and new versions of the program for a direct comparison.

2.1 Performance Tools

There are many available tools for performance analysis [19]. Tools work at different levels of abstraction: from hardware-based event counters, to resource monitoring inside operating system kernels, code and binary instrumentation, up to the simple utilization of runtime as a reference metric.

Benchmarks are specially-built synthetic programs, or even a predefined set of real-world programs that provide a reference figure to compare performance. The required features for a benchmark are portability, simplicity, stability and results reproduction. It is also required that execution time is reasonable, and problem size can be adjusted to keep applicability over time and the evolution of technologies. High Performance Computing Challenge (HPCC) [22] is a package that includes several HPC benchmarks and provides multiple metrics at once.

An incremental approach is proposed in this paper, applying tools in the sequence shown in Table 1, every step adding more information and detail to the analysis. The overall system capacity extracted using the HPCC benchmarks allows to set a practical limit on performance of the system being used. The timing of workload executions allows to understand their runtime deviation. The execution profile will show call flows and bottlenecks at function, source code line and assembly levels. The program scaling behavior shows speedup trends when incrementing either problem size or processing units. A system profile show impact on available system resources. At last, low-level reports on vectorization details and hardware counter status can lead to fine-tuning needs at CPU instruction level.

Table 1: Performance Tools Incremental Application.

Information	Tools
Overall System Capacity	HPCC Benchmark
Workload Execution	time, gettimeofday()
Execution Profile	gprof, perf
Program Scaling	gprof, perf
System Profile	Perf
Vectorization	Compilers
Hardware Counters	Perf

3. Optimization Problems

The approach in this paper identifies three different dimensions in which the problems have to be addressed: performance analysis, optimization methods implementation, and supporting infrastructure.

3.1 Performance Analysis

The performance analysis has multiple challenges:

- Human interaction is always a source of involuntary mistakes. Miss investing valuable time in tasks that can be automated. Usually, one person should run tests, gather results, and draw charts in a performance report to rely on during subsequent analysis. Absolute discipline is mandatory as the main time-consuming task is to execute the same program under different configurations to record its behavior.
- Required expertise in tools. Learning about the proper use of the multiple tools takes considerable time. However, those tools povide high quality information necessry to take data-driven decisions at the time of optimization efforts. The analysis requires the correct use of statistics to average results, discard noise and outliers, and understand limits on potential improvements.
- Data gathering and representation of results. The analysis requires the gathering of supporting data about both the program and the system being analyzed. Idetifying metrics, how to get them and even how to represent them is non-trivial and requires time and expertise.
- Early optimization. An optimization made in the wrong place implies wasted effort and potentially little impact on the overall execution runtime of the program.
- Naïve implementation of algorithms. Direct implementation of an algorithm may guarantee correctness but not efficiency without first understanding underlying low-level details of the computer architecture. Reuse of portable math libraries guarantee both correctness and

performance with only a minimum effort on learning how to apply them.

And many of them (maybe all) are interraleted, so they cannot be treated indepently of each other.

3.2 Optimization Methods Implementation

The usual optimization methods target different aspects of a given program:

- Code. The source code is analyzed to improve jump prediction by pre-fetching units, in-lining frequently used routines to avoid unnecessary returns, align code and unroll loop to make program steps easy to map into vectorised instructions, among many other aspects.
- Execution. The generated assembly level code is inspected to verify the usage of lightweight and vectorised instructions, trying also to reduce the number of used registers.
- Memory. Program structures are analyzed to minimize cache failures, align inner components and improve the locality of data to enhance the memory hierarchy performance.
- Prefetching. Use of pre-fetching instructions are analyzed to help prediction unit to be more effective.
- Floating point arithmetic. Given a problem it can be considered to relax representation assumptions following standards, in order to exchange accumulated error and final precision by processing speed.

And, again (as in the case of the performance analysis explained above), many of them (maybe all) are interraleted, so they cannot be treated indepently of each other.

3.3 Supporting Infrastructure

Clearly, performance analysis needs (semi)automated support, the following are some requirements that any framework should include:

- Reusability. The framework should be applicable to a wide range of programs, without requiring its modification. Its deployment should only depend on the same tools that any user may need to gather performance-related information manually.
- Configuration. The framework should be configurable and parameters should include how to compile and execute the program, the input range to consider, and the number of repetitions to check for stability, among other details.
- Portability. The framework should be implemented in a portable (and maybe interpreted) language, to allow easy review and the expansion with new tools, new data gathering experiments or the modification on how things are being done.
- Extensibility. The framework should be designed for extension from scratch, incorporating new tools, charts o sections on a final report should be an almost trivial task given the user already knows how to gather the metric manually.
- Simplicity. The framework should reuse the same tools available at system level to a regular user. It should generate log files of all the issued commands and their output so any user can check them if required. It should be possible to use the framework to run overnight and allow incremental optimization.

4. Proposed Solution

We propose an approach that combines a generic procedure on top of an automated framework that takes care of running the program multiple times, applying performance analysis tools, identifying bottlenecks, gathering metrics and representing results graphically into a performance report. This allow the user to solely focus on optimization of the identified bottlenecks, freeing him of performance-related data recollection work.

4.1 Method

The performance analysis procedure needs to be done incrementally to guarantee progress no matter the allowed time for the task. We propose then a process which starts analyzing system computing power. The computing power is taken into account to follow a sequence of improvement cycles of execution of the program for gathering performance metrics and revealing bottlenecks. Thus, optimization efforts can be focused on those identified bottlenecks, in order to maximize overall impact. The following steps reflects how the process should be done:

1) Run the HPCC benchmark to get insights on overall system performance.
2) Run the program multiple times using the same workload to check deviations.
3) Establish a baseline using geometric mean to avoid measurement noise.
4) Run the program scaling problem size to dimension its behavior.
5) Run the program scaling computing resources to dimension its behavior.
6) Extract execution profile of the program
7) Extract system profile while running the program.

4.2 Framework

A framework named hotspot (https://github.com/more-andres/hotspot) was built, implementing the previous procedure. The automation behaves exactly like a regular user running commands and checking their output. It executes tools like gcc, make, prof, gprof, pidstat (among others) to gather relevant information and finally using the Latex typesetting environment to compile a human-friendly report including the data and associated charts.

Currently, only GNU/Linux systems with kernels above 2.6.32 and OpenMP threaded programs are supported out-of-the-box. Old kernel versions do not properly support the tools used to identify bottlenecks at assembly level. OpenMP threaded programs easily let to change the number of processing units in order to understand scaling behavior.

At a lower abstraction level, the framework is designed with two simple object class hierarchies as shown in Fig. 2. The first one keeps the main engine which runs the second one as independent sections that gather information and report back metrics. The latter one can be extended to add more sections to the final report. The implementation

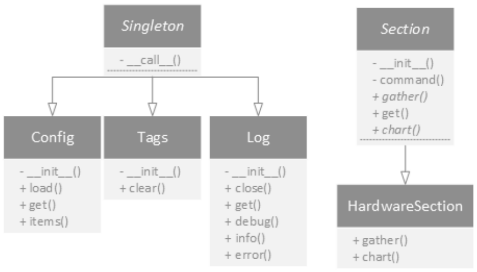

Fig. 2: Speedup Limits According to Amdahl (left) and Gustafson (right)

defines multiple objects as part of these class hierarchies. The components of the framework engine are:

- Singleton: extended by other object to guarantee unique instance.
- Tags: storage of keywords to be replaced at the report template.
- Log: manager used to structure output messages.
- Config: manager used to read and share configuration attributes.

And the components that hold the responsibility and knowledge of running experiments and gathering relevant metrics are the following:

- Section: extended by sections to be included in the report.
- HardwareSection: hardware-related information such as used CPU and Memory.
- ProgramSection: program-related information such as used problem input range.
- SoftwareSection: program-related information such as used compiler and libraries.
- SanitySection: basic check on that the workload can run without issues.
- BenchmarkSection: runs HPCC and gather relevant metrics.
- WorkloadSection: reports on program footprint and its stability.
- ScalingSection: reports on problem size scalability.
- ThreadsSection: reports on computing scalability.
- OptimizationSection: reports on program behavior under compiler optimizations.
- ProfileSection: reports on program execution profile, call flow and bottlenecks.
- ResourcesSection: reports on the use of system resources while the program runs.
- AnnotatedSection: reports annotated bottlenecks mapping code to assembly.
- VectorizationSection: reports loops being vectorised or not and the impediments
- CountersSection: reports hardware counter status.
- ConfigSection: reports used configuration for the overall framework execution.

The framework uses a per-program hidden directory to keep record on executions, classified per timestamp and caching the different results to avoid long waiting times.

A configuration file is used to define the framework parameters to handle any program. It is important to note that the framework needs to know how to run, compile and instrument each program as part of the configuration. The framwork will rely on variables that hold problem input size, numbers of OpenMP threads to use and even compiler flags to instrument binaries.

4.3 Operation and Report

At high level the design mimics how a user manually interacts with the system. The framework depends on tools available on the system, the program, and its matching configuration plus the LaTeX typesetting system.

After reading the configuration, the workload is executed multiple times and its wall time is checked for stability by charting a histogram using as a baseline a normally distributed curve. The geometric mean is extract to be used as a reference in future executions. As second step, the program is executed over the full range of input size and available computing resources, with this scaling charts are generated, plotting ideal scaling as a comparison approach as well. Using this scaling information the potential speedups are computed according to Amdahl and Gustafson laws. Then using expanded debugging information bottlenecks are identified on logical, source and assembly program levels. One last execution is done while at the same time resource usage is recorded to understand system bottlenecks. After all these executions the gathered information is used to generate a detailed PDF report to support the performance optimization task.

The general considerations followed when building the report include:

- Format similar to a scientific paper.
- Hyperlinks to full logs of tools output.
- Brief explanation of each section and chart objective.
- Inclusion of ideal trends and behavior in charts.

- References to base bibliography.

Several pieces of information are given, such as: a) Abstract: a brief summary introducing the framework and the location of the supporting output for detailed review, b) Content: reduced table of content with direct links to the information, c) Program: details of the program under analysis, timestamp of the analysis and input parameters used during tests, d) System computing poer: beyond specifics about hardware and software on the system, the framework includes the metric reported by an HPCC execution, e) Workload: working set and in-memory structures, histogram of execution to understand its deviation and the geometric mean used as a baseline. Also a chart checking how the compiler optimization levels improve time.

5. Case Study - Examples

This section will showcase the hints that the framework can lead to, but will not include solutions to those as they are out-of-scope. The framework was used to analyze three well-known compute kernels: a) Mtrix Multiplication: a naïve implementation of dense matrix multiplication., b) 2D Heat Distribution: a naïve implementation of iterative heat distribution, and c) Mandelbrot Set: a naïve implementation of a recursive fractal algorithm.

The HPCC benchmarks executed and its multiple reference metrics are included to be used as top reference of system capabilities, shown in Table 2.

Table 2: Performance Metrics Reported by HPCC

Benchmark	Value	Unit
hpl	0.00385977 TFlops	tflops
dgemm	1.03413 GFlops	mflops
ptrans	0.997656 GBs	MB/s
random	0.0274034 GUPs	MB/s
stream	5.19608 MBs	MB/s
fft	1.2658 GFlops	MB/s

5.1 Examples

Fig. 3 shows the initial performance and metrics report provided by our tool. As a first step towards optimization,

Execution time:
(a) problem size range: `1024 - 2048`
(b) geomean: `4.61694` seconds
(c) average: `4.61757` seconds
(d) stddev: `0.07678`
(e) min: `4.52168` seconds
(f) max: `4.73660` seconds
(g) repetitions: `8` times

Fig. 3: Initial Matrix Multiplication Report

the different compiler optimization options are also reported, shown in Fig. 4. We have seen almost the same *pattern*

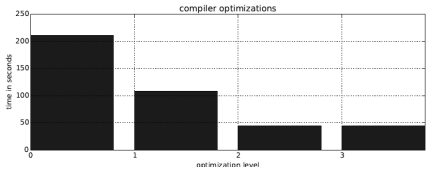

Fig. 4: Compiler Optimizations on Matrix Multiplication

for the different compiler optimization levels on several programs, e.g. a huge improvement for -O1 and -O2, and relatively small improvement for -O3. Fig. 5 shows the information about serial and parallel fractions as well as the limits of parallelization according to de the so called Amdhal and Gustafson laws.

1. Parallel Fraction: `0.52260`.
   Portion of the program doing parallel work.
2. Serial: `0.47740`.
   Portion of the program doing serial work.

Optimization limits can be estimated using scaling laws.

1. Amdhal Law for 1024 procs: `2.09466 times`.
   Optimizations are limited up to this point when scaling problem size. [2]
2. Gustafson Law for 1024 procs: `535.61564 times`.
   Optimizations are limited up to this point when not scaling problem size. [3]

Fig. 5: Matrix Multiplication Parallelization Reports

The tool on the naïve implementation of heat distribution in 2 dimensions identifies:

- Workload is stable, although deviation is greater than in the previous case.
- Compiler optimizations have little impact.
- Execution time does not grow monotonically when scaling input size; it neither decreases when adding more computing units.
- Parallelism only reaches 65%, therefore speedup has a limit on 2.9x.
- There are two main bottlenecks with 30% and 15% of overall execution time.
- CPU utilization is not constant, and loops are not vectorised.

On the third example, the naïve implementation of Mandelbrot sets [23], the tool reports:

- Structures used to represent complex numbers are aligned, without holes that may consume cache memory.
- Workload is stable.
- 50% of the time is spent on the same line of code.
- There are already optimized cycles using the movss and adds vectorised in-structions.

6. Conclusions and Further Work

The performance optimization process is not trivial and requires disciplined analysis of used resources and gathering of metrics characterizing program behavior. This works reviews the development of a supporting framework that streamlines the process by running in unattended mode and generating a report to direct optimization efforts. The generated report combines multiple tools to identify bottlenecks at function, source, and assembly level (e.g. vectorization). It also records system configuration and resource usage. All of this is offered as an unattended automated task before undergoing the program optimization analysis. We hence propose a systematic procedure supported with an automated framework that is suitable for both newcomers and experts. It also allows the exchange of standardized performance reports between research and development groups.

Extension possibilities on this framework are straightforward considering new sections can be added to the report including the application of new tools, charts or contextual information. The application of this framework to a well-known open source program will size its usefulness and provide feedback on both the procedure and the generated report. At last, it may be interesting to move from a static report format to a dynamic one offering pivot tables and charts that can be reconfigured easily. This may be better achieved using HTML5 technologies over a browser for instance.

References

[1] A. More. A Case Study on High Performance Matrix Multiplication. Technical Report, Universidad Nacional de La Plata, 2008. Available at http://mm-matrixmultiplicationtool.googlecode.com/files/mm.pdf

[2] P. E. McKenney. Is Parallel Programming Hard, And, If So, What Can You Do About It? kernel.org, 2010.

[3] Intel Corporation. Intel Math Kernel Library. Reference Manual. Intel Corporation, 2009.

[4] R. Garabato, V. Rosales, A. More. Optimizing Latency in Beo-wulf Clusters. CLEI Electron. J., 15(3), 2012.

[5] J. Jeffers, J. Reinders. Intel Xeon Phi Coprocessor High Performance Programming. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[6] A. More. Lessons Learned from Contrasting BLAS Kernel Implementations. In XVIII Congreso Argentino de Ciencias de la Computación, 2013.

[7] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the April 18-20, 1967, spring joint computer conference, AFIPS '67 (Spring), pages 483-485, New York, NY, USA, 1967. ACM.

[8] J. L. Gustafson. Reevaluating Amdahl's Law. Communications of the ACM, 31:532-533, 1988.

[9] A. H. Karp, H. P. Flatt. Measuring parallel processor performance. Commun. ACM, 33(5):539-543, May 1990.

[10] P. J. Fleming, J. J. Wallace. How not to lie with statistics: the correct way to summarize benchmark results. Commun. ACM, 29(3):218-221, March 1986.

[11] K. Atkinson. An Introduction to Numerical Analysis. Wiley, 2 edition, 1989.

[12] C. U. Smith. Introduction to software performance engineering: origins and outstanding problems. In Proceedings of the 7th international conference on Formal methods for performance evaluation, SFM'07, pages 395-428, Berlin, Heidelberg, 2007. Springer-Verlag.

[13] J. C. Browne. A critical overview of computer performance evaluation. In Proceedings of the 2nd international conference on Software engineering, ICSE '76, pages 138-145, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[14] M. Woodside, G. Franks, D. C. Petriu. The future of software performance engineering. In 2007 Future of Software Engineering, FOSE '07, pages 171-187, Washington, DC, USA, 2007. IEEE Computer Society.

[15] T. Mattson, B. Sanders, B. Massingill. Patterns for parallel programming. Addison-Wesley Professional, First edition, 2004.

[16] K. A. Huck, O. Hernandez, V. Bui, S. Chandrasekaran, B. Chapman, A. D. Malony, L. Curfman McInnes, B. Norris. Capturing performance knowledge for automated analysis. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing, SC '08, pages 49:1-49:10, Piscataway, NJ, USA, 2008. IEEE Press.

[17] T. Margalef, J. Jorba, O. Morajko, A. Morajko, E. Luque. Performance analysis and grid computing. In V. Getov, M. Gerndt, A. Hoisie, A. Malony, B. Miller, editors, Performance analysis and grid computing, chapter Different approaches to automatic performance analysis of distributed applications, pages 3-19. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[18] F. Wolf B. Mohr. Automatic performance analysis of hybrid mpi/openmp applications. J. Syst. Archit., 49(10-11):421-439, November 2003.

[19] B. Gregg. Linux Performance Analysis and Tools. Technical report, Joyent, February 2013.

[20] J. D. McCalpin. Memory bandwidth and machine balance in current high perfor-mance computers. IEEE Technical Committee on Computer Architecture (TCCA) Newsletter, Dec 1995.

[21] U. Drepper. What Every Programmer Should Know About Memory. Tech-nical report, Red Hat, November 2007.

[22] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. Mccalpin, D. Bailey, D. Takahashi. Introduction to the HPC Challenge Benchmark Suite. Technical report, 2005.

[23] B. B. Mandelbrot, D. E. Passoja, editors. Fractal aspects of materials: metal and catalyst surfaces, powders and aggregates: extended abstracts, volume E-4 of Materials Research Society extended abstracts, Pittsburgh, PA, USA, 1984. Materials Research Society.

C.3. Libros

C.3.1. Contribuciones en el Libro *Programming Intel Xeon Phi*

Agradecimiento incluído de los autores por el borrador de los capítulos sobre *Intel Cluster Checker* e *Intel Cluster Ready*.

C.3.2. Reseña del Libro *Intel Xeon Phi Coprocessor High Performance Programming*

Reseña del Libro *Intel Xeon Phi Coprocessor High Performance Programming* - JCS&T Vol 13 N 2 Octubre 2013 [1].

[1] JCS&T Vol. 13 No. 2

Book Review:

Intel Xeon Phi Coprocessor High Performance Programming
James Jeffers, James Reinders
Morgan Kaufmann, 2013
ISBN-13: 978-0124104143

James Reinders (Chief Evangelist of Intel® Software at Intel) and Jim Jeffers (Principal Engineer at Intel) have gathered in this book the working knowledge of the people involved to prototype and productize the Intel® Xeon Phi coprocessor. This include the experience of Intel's Product, Field, Application and Technical Consulting Engineers, as well as key partners involved on product pre-release work.

The book offers a hands-on and practical guide, presenting best-known-methods for High Performance Computing programming with the Intel Xeon Phi coprocessor. The book's contents can be split in four main sections: the usual introduction, an interesting optimization guide, plus product architecture details and, before concluding, a review of available tools and libraries.

The initial section introduces Intel Xeon Phi through analogies to a racing car. *Chapter 1 – Introduction* details when, why and how to use Intel Xeon Phi coprocessors. This chapter starts applying an analogy of how sports cars are designed and how they behave in a variety of situations. One key point of this introduction is that learning how to optimize code to expose parallelism will be useful to any other processor, not only Intel Xeon Phi. *Chapter 2 - High Performance Closed Track Test Drive* takes the sports cares analogy to a test with a sample hello world code with the usual SAXPY computation. Through real command line examples takes the reader from the C compiler invocation, passing for the checking of the vectorization results, up to the actual execution. Surprisingly, all of this happening in a Linux image running inside the coprocessor card. Discussion continues showing how to use multithreading computation using OpenMP extensions. How to offload work directly from a host-system is also shown, double checking that gathered performance is about the same. *Chapter 3 - A Friendly Country Road Race* moves the sports car out of the synthetic and controlled environment to solve a 9-point stencil algorithm. On this case even running over multiple coprocessors using MPI capabilities. Then low-level tuning start with access to memory taking into account memory alignment, streaming stores and bigger memory pages. *Chapter 4 - Driving Around Town* uses a short but still representative example on how to vectorize a loop by exposing parallelism properly, focusing on data locality and data tiling to favor cache reuse.

The second section goes into detail on parallelization techniques. A good aspect to take into account is these optimization techniques might end up favoring any multicore system. *Chapter 5 - Lots of Data (Vectors)* gives helpful vectorization insights, proposing a simple systematic procedure that can be iterated over the code. For instance, manual loop unrolling is disregarded here as it is more future-proof to guide the compiler instead of manual re-tuning when architecture evolves. *Chapter 6 - Lots of Tasks (not Threads)* takes the parallelism abstraction up level, up to computation threads. Here it is pointed out that thread creation should happen at the microprocessor, thread launch nesting is discussed and shown with language extensions like OpenMP and FORTRAN arrays, plus abstraction libraries like Intel® Threading Building Blocks, Cilk and the well-known Intel® Math Kernel library.

The third section shows product specific details, showing how to offload work and exciting internals on the available hardware and computation units. *Chapter 7 – Offload* discusses available offload models, the code can be executed natively, offloaded manually or even automatically. This enables asynchronous computation when offload is properly scheduled. *Chapter 8 - Coprocessor Architecture* touches bare metal architecture, providing details about cache organization, vector processing units, direct memory access and available power management interfaces. *Chapter 9 - Coprocessor System Software* details low level libraries used to exchange information between the host system and the coprocessor. These libraries abstract PCI mapped memory as a regular networking device, allowing usual IPC mechanisms to work against the coprocessor. Memory allocation at operating system kernel level is discussed as it provides yet another optimization possibility.

Chapter 10 - Linux on the Coprocessor provides Linux specifics of the image running inside the processor, this fact enables easy extension of the available software, as Open Source tools and libraries might be compiled natively and included inside the Linux image.

A fourth section explain how to use the multiple libraries and tools made available by Intel to increment productivity. Not only for compiling code, but also graphical tools to review profiling information of code running on the coprocessor. This tools were built specifically for the product, with negligible overhead thanks to the use of special purpose hardware counters. *Chapter 11 - Math Library* and *Chapter 12 – MPI* shows how the well-known software now supports Intel Xeon Phi, offering specific optimizations and abstractions to de-facto interfaces like BLAS and FFT. *Chapter 13 - Profiling and Timing* makes a useful summary of profiling information such as expected CPI, monitored events and efficiency metrics to look at while reviewing performance.

In a nutshell, this book covers parallel programming with Intel Xeon Phi. The book is reader friendly, both for the novice and expert, as it includes tons of analogies and examples with real-world code. Luckily, the end of the book specifies that a Volume 2 is coming with even more low level details to exploit every available computing capability left. Lots of pointers to relevant publications are included on each chapter during the discussion of presented ideas, it is up to the reader to follow them and get in-deep details.

Disclaimer: the author of this review currently works for Intel Corp. as a Senior Software Engineer. He personally collaborated with draft content of the book sections about Intel Cluster Ready and Intel Cluster Checker, projects on which contributed for more than 5 years.

Andres More
Intel Corporation (andres.more@intel.com)
Instituto Universitario Aeronáutico (amore@iua.edu.ar)