Instituto de Computación - Facultad de Ingeniería
Universidad de la República

Proyecto de grado

Adaptación del modelo WRF a una arquitectura multi/many cores (GPUs)

Montevideo, Mayo de 2013

Presentado por: Marcel Burdiat, José Ignacio Hagopian, Juan Pablo Silva
Tutores: Pablo Ezzati, Martín Pedemonte
Cliente Responsable: Ernesto Dufrechou

Resumen

Uruguay está incluyendo gradualmente energía eólica en su matriz energética. En este contexto se desarrolló en el Instituto de Mecánica de los Fluidos e Ingeniería Ambiental (IMFIA) una herramienta para predecir la energía eólica inyectada a la red en el Complejo de Parques Eólicos "Ing. Emanuele Cambilargiu" con un horizonte de 48 horas. Esta herramienta está basada en el modelo de simulación de magnitudes climáticas Weather Research and Forecasting (WRF). Una de las principales limitantes que presenta la herramienta es su elevado costo computacional, ya que la ejecución del WRF requiere un gran tiempo de cómputo.

En este trabajo se aborda la utilización de técnicas de computación de propósito general en unidades de procesamiento gráfico para la aceleración de rutinas del WRF. Con este objetivo, se realiza un relevamiento del estado del arte del uso de unidades de procesamiento gráfico para la reducción del tiempo de ejecución del mismo, así como un análisis de las rutinas que requieren mayor tiempo de ejecución en el contexto de su utilización para la predicción de viento en Uruguay. A partir de ello, se identificaron rutinas que fueron portadas exitosamente a GPU.

Los resultados obtenidos, sobre casos de prueba reales, denotan speedups de hasta 10× para los módulos abordados que redundan en aceleraciones cercanas al 50 % para configuraciones secuenciales del WRF, y de hasta 30 % para configuraciones multihilo.

Palabras clave: GPU, Paralelismo, WRF, CUDA, Energía eólica

Índice general

Capítulo 1

Introducción

En el Uruguay, investigadores del Instituto de Mecánica de Fluidos e Ingeniería Ambiental (IMFIA) han desarrollado una herramienta para la predicción de la energía generada en los Parques Eólicos. La herramienta está basada fuertemente en el WRF (del inglés Weather Research and Forecasting), uno de los modelos numéricos para la simulación y predicción climática más utilizado a nivel mundial. Actualmente, se está perfeccionando la capacidad de simulación de la herramienta utilizando diferentes técnicas, entre otras técnicas se destacan la utilización de grillas de mayor precisión, adquisición y asimilación automática de datos y el modelado del efecto de las nubes. Este tipo de herramientas aunado a los nuevos requisitos numéricos necesitan una importante capacidad de cómputo, siendo esto una limitante importante para su utilización y perfeccionamiento.

Una opción de computación de alto desempeño (HPC) de bajo costo económico es la utilización de procesadores secundarios. En particular, en los últimos años las tarjetas gráficas (co-procesador gráfico, GPU) han experimentado una evolución vertiginosa. La evolución no ha sido únicamente cuantitativa (capacidad de cómputo), sino que en gran medida ha sido cualitativa (capacidad de resolver diferentes problemáticas). Este hecho, ha motivado que diversas empresas/laboratorios busquen la utilización de las GPUs para atacar problemas de propósito general, particularmente en áreas de intenso requerimiento de cómputo (computación gráfica, computación científica, etc.). Se conoce dicha área como "Computación de Propósito General en Unidades de Procesamiento Gráfico" (GPGPU, por su sigla en inglés "General-Purpose Computing on Graphics Processing Units").

Basado en lo anterior, los objetivos del proyecto consisten en el análisis del uso de técnicas de GPGPU para acelerar el tiempo de cómputo de la herramienta de simulación de energía eólica, con especial hincapié en el WRF. En particular, avanzar en el estudio de la utilización de las GPUs para la resolución de problemas de propósito general, presentar diseños e implementaciones de soluciones que permitan acelerar el WRF utilizando GPUs, y realizar una evaluación final de la solución implementada.

El documento se estructura del modo que se describe a continuación.

En el Capítulo 2 se presenta el contexto en el que se enmarca este proyecto; allí se realiza una presentación de la herramienta desarrollada por el IMFIA junto con la motivación del problema que se intenta resolver.

Luego, en el Capítulo 3, se introduce la utilización de GPUs para la resolución de problemas de procesamiento de propósito general enmarcada en la tecnología CUDA utilizada en este trabajo. También se realiza un resumen del estado del arte de la utilización de GPUs para la aceleración del WRF.

La etapa de análisis es presentada en el Capítulo 4, en el cual se detallan las plataformas y escenarios utilizados, junto con los resultados de la etapa de profiling del WRF. Los resultados conseguidos en esta etapa sientan las bases para el diseño de las soluciones implementadas.

En el Capítulo 5 se presentan las propuestas realizadas para acelerar distintos módulos de la aplicación junto con la integración de trabajos existentes en el área, así como la evaluación experimental y análisis de resultados obtenidos. Las propuestas se presentan de forma versionada, mostrando para cada versión la descripción de su diseño, junto con el speedup alcanzado. También se presenta el análisis de diversos módulos que no fueron portados a GPU, acompañado de la justificación correspondiente. Se culmina el capítulo con una evaluación final de resultados integrando toda las implementaciones.

Finalmente, se realiza una valoración final del trabajo realizado y se señalan posibles direcciones de trabajos futuros que prometen seguir acelerando la herramienta analizada, pero que por razones de tiempo fueron dejados fuera del alcance de este proyecto.

Capítulo 2

Pronóstico de energía eólica en Uruguay

2.1. Introducción

Si bien hace tiempo que se utiliza energía eólica en Uruguay para uso doméstico, la generación de energía eléctrica a partir de parques eólicos a gran escala en Uruguay es reciente. Según el sitio web gubernamental de energía eólica [44], en el 2006 se instalaron establecimientos eólicos a cargo de la empresa "Agroland", en el departamento de Rocha, llegando a potencias de 13 MW, siendo "Nuevo Manantial" su parque más importante. Dicha empresa fue una de las precursoras en la generación de energía eólica. A partir del 2007 se puso en funcionamiento el Programa de Energía Eólica en busca de diversificar la matriz energética del país y ampliar el campo de las energías renovables utilizadas, para así incorporar esta fuente de energía a la red nacional incentivando la instalación de parque eólicos en distintas zonas del país. Este programa es una iniciativa conjunta del Gobierno Nacional con el Programa de las Naciones Unidas para el Desarrollo, ejecutado por el Ministerio de Industria, Energía y Minería, y financiado por el Fondo Global Para el Medio Ambiente. El Programa busca crear condiciones favorables e incentivar el proceso de inserción de la energía eólica en el país desde un abordaje multidisciplinario, de modo de alcanzar el objetivo de contribuir a la mitigación de emisión de gases de efecto invernadero. Sus áreas de trabajo abarcan aspectos de regulación y procedimientos, información y evaluación del recurso eólico, aspectos medioambientales, tecnológicos y financieros entre otros. Además, busca crear las capacidades técnicas en el país tanto a nivel de instituciones públicas como de desarrollos privados como potenciales proveedores de la industria eólica. Plantea una interacción entre los principales actores a nivel nacional: el Poder Ejecutivo, las Intendencias Municipales, Ministerios, UTE, la Universidad de la República, industriales y emprendedores privados, entre otros.

En el 2008, gracias al Programa de Energía Eólica, se instaló el primer parque del Complejo de Parques Eólicos "Ing. Emanuele Cambilargiu", propiedad de UTE, y ubicado en la "Sierra de los Caracoles", cerca de San Carlos en el departamento de Maldonado. Este complejo está compuesto por dos parques, cada uno con una potencia instalada de 10 MW, provistos por 5 máquinas Vestas V80 de 2 MW. Cuenta con última tecnología y es considerado el complejo eólico más importante del Uruguay en la actualidad. Desde mayo del 2011 se encuentra instalado en el departamento de San José el parque eólico "Magdalena" de la firma "Kentilux". El mismo cuenta con una potencia instalada de 10 MW, sumando un total de 43 MW en el país. Actualmente se encuentran varios proyectos en desarrollo. Se destacan entre ellos el aumento del parque "Magdalena" para contar con

20 MW instalados para el año 2013, tres proyectos por un total de 150 MW, a cargo de las empresas “Palmatir”, “Jistok” y “Fingano”, con 50 MW cada una estimándose que se contará con estos parques para el 2014 y otros proyectos por un total de 192 MW estimados también para el 2014. En la Figura 2.1 se puede ver el mapa eólico del Uruguay conteniendo la ubicación y capacidad de los parques eólicos propuestos y en operación más importantes.

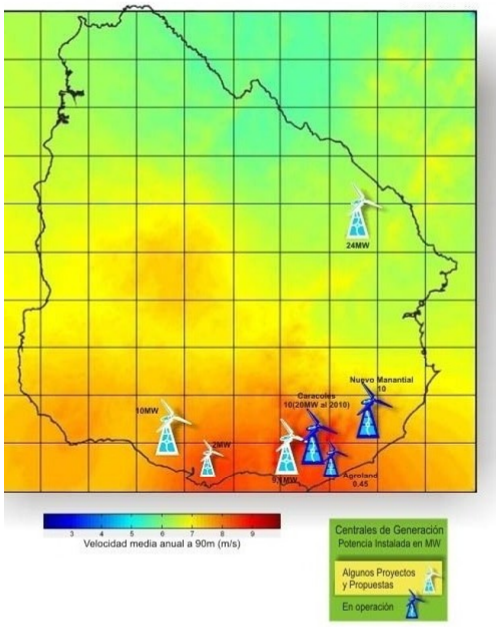

Figura 2.1: Velocidad media del viento y capacidad instalada en centrales eólicas, extraído de [11].

Con estos pronósticos se espera que para antes de 3 años la energía generada a partir de parques eólicos sea mayor al 10 % de la energía total suministrada por el país. Además las perspectivas para la generación eólica son muy auspiciosas, teniendo en cuenta diversos aspectos como lo son, el potencial eólico nacional llegando a vientos promedios de más de 9 m/s en regiones de Rocha y Maldonado, las ventajas impositivas que el país ha asignado a los emprendimientos de generación de energía renovables, el crecimiento de la demanda de energía eléctrica y la necesidad de diversificación de fuentes de energía del país [44].

De acuerdo a la importancia que tiene y tendrá el recurso eólico en nuestro país, es de suma importancia disponer de herramientas para la predicción de la potencia suministrada a la red.

2.1.1. Herramienta desarrollada por el IMFIA

La energía eólica tiene características que la diferencian respecto a otros tipos de fuentes de energía, como la hidráulica o la térmica. La energía hidráulica permite represar caudales de agua para así poder guardarla como energía potencial para luego tomar la decisión según sea conveniente de transformar dicho potencial en energía cinética para que pase por las turbinas y así transformarse en energía mecánica que permite la generación de energía eléctrica. En el caso de las centrales térmicas, la fuente de energía se encuentra

en los combustibles, de los cuales se puede decidir cuando utilizarlos en función de la demanda. En el caso de la energía eólica no existe la posibilidad de represar la energía origen, en este caso energía cinética del flujo de aire, para la generación de energía eléctrica. La energía cinética del flujo de aire se transforma en energía mecánica de giro que se utiliza para la generación de energía eléctrica que debe, sí o sí, ser inyectada en forma directa a la red.

La naturaleza del origen de la energía utilizada para la generación hace que la energía eólica tenga mayores fluctuaciones en cuanto a la energía generada respecto a otras fuentes (ver Figura 2.2), al punto que en cierto momento se puede estar generando potencia plena, y en un lapso de pocas horas pasar a generar un 50 % de dicha energía o incluso un 0 %. Esta variabilidad hace necesaria tener una herramienta que genere información sobre la generación futura para así obtener datos que permitan tomar decisiones en la gestión del sistema eléctrico, como por ejemplo activar o no centrales térmicas e hidráulicas, de forma tal de atender la demanda en forma satisfactoria.

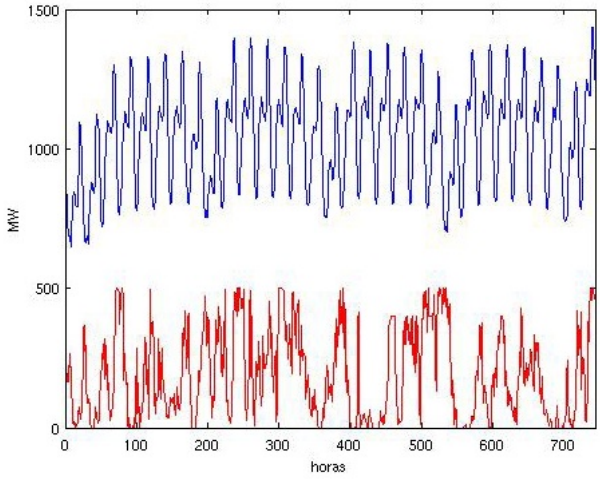

Figura 2.2: En azul demanda horaria de energía, en rojo simulación de generación de 500MW de energía eólica, extraído de [1].

Dada ésta necesidad se desarrolló en el IMFIA (Instituto de Mecánica de los Fluidos e Ingeniería Ambiental) una herramienta para predecir la producción de energía eólica alcanzada en el complejo de parques eólicos "Emanuele Cambilargiu". La misma se realizó en el marco del proyecto "*Desarrollo de herramientas de Predicción, de corta y muy corta duración (2 a 48 horas) de la Generación de Energía Eléctrica de origen eólico*" presentado por la Facultad de Ingeniería al Fondo Sectorial de Energía dentro de la ANII (Agencia Nacional de Investigación e Innovación) [13] para contar con los fondos necesarios para el desarrollo. Se puede acceder a los resultados obtenidos con dicha herramienta en el sitio web "Pronósticos Numéricos Operativos en Uruguay" del IMFIA [2]. Este proceso de predicción se ejecuta diariamente y genera predicciones a 48 horas que luego son transferidas a UTE gracias a un convenio firmado entre el ente y la Facultad de Ingeniería.

El proceso se compone de una serie de pasos. En primer lugar se obtienen datos meteorológicos globales gracias a la OMM [42], a partir de estos se generan pronósticos meteorológicos mundiales públicos generados por la NOAA/National Weather Service National Centers for Environmental Prediction [33] a partir del modelo de circulación

general de la atmósfera GFS (Global Forecast System). Estos pronósticos son generados cuatro veces al día, a las 00:00 GMT, 06:00 GMT, 12:00 GMT y 18:00 GMT y tienen una resolución de grilla de $1^o \times 1^o$ (aproximadamente 100km $\times$ 100km).

Luego de transferir cada uno de estos pronósticos del sitio web NOAA al servidor de cálculo (a partir de un script de transferencia de datos GRIB2 [48]), los mismos sirven de entrada al WRF (Weather Research and Forecasting Model) ejecutado localmente. Dicho modelo toma estos datos junto con otros parámetros y genera un pronóstico de baja resolución para el país y uno de alta resolución en la región de la "Sierra de los Caracoles". Este proceso conlleva un tiempo de ejecución de aproximadamente 2 horas en un servidor con 8 núcleos, el cual es un tiempo elevado, principalmente si se desea realizar simulaciones de mayor precisión.

A partir de los pronósticos generados por el WRF se utiliza una herramienta MOS (Model Output Statistics [3]). Generar un MOS en este caso se refiere a generar pronósticos de la energía de origen eólico inyectada a la red por cada aerogenerador en base a datos estadísticos, utilizando para esto la velocidad y dirección del viento pronosticada a la altura del eje de cada aerogenerador. Es de importancia la dirección, además de la velocidad, debido a que la interferencia entre aerogeneradores y la consiguiente generación de cada equipo depende de la dirección del viento. Si bien se busca minimizar la interferencia entre aerogeneradores, por criterios de optimización económica pueden darse en mayor o menor medida fenómenos de interferencia para algunas direcciones. Al terminar de correr la herramienta MOS se publican los resultados en el sitio web de la herramienta. Estos resultados son publicados de forma automática en intervalos de 6 horas, teniendo así cuatro pronósticos diarios que pronostican el resultado para las siguientes 24 horas.

En la Tabla 2.1 se presentan los horarios de ejecución del proceso implementado.

Lugar	**Proceso**	**Hora**
OMM	Asimilación global de datos, a escala planetaria	09:00 Local 12:00 GMT
NOAA	Disponibilidad pública de pronósticos GFS, sitio web NOAA	15:00 Local 18:00 GMT
IMFIA	Script de transferencia de datos GRIB2	15:30 Local
IMFIA	Script WRF	16:00 Local
IMFIA	Scirpt MOS	18:00 Local
IMFIA	Script de carga en sitio web	18:10 Local

Tabla 2.1: Horarios de ejecución del proceso de predicción.

Como se puede observar en la Tabla 2.1, la principal etapa desde el punto de vista de tiempo de ejecución es el cómputo del WRF local.

2.2. WRF

El WRF es un modelo numérico de predicción del tiempo y un sistema de simulación atmosférica diseñada para investigación y aplicaciones operacionales [47]. El mismo es una herramienta de Software Libre que fue realizada gracias al esfuerzo de varias agencias, entre ellas, National Center for Atmospheric Research (NCAR), Mesoscale and Microscale Meteorology Division (MMM), National Oceanic and Atmospheric Administration (NOAA), National Centers for Environmental Prediction (NCEP), entre otras y junto con la colaboración de muchos científicos. El modelo puede ser utilizado en un amplio

espectro de aplicaciones, desde pequeñas regiones a simulaciones globales. Estas aplicaciones incluyen predicciones numéricas en tiempo real, simulaciones climáticas regionales, modelado del aire, procesos generados por la interacción entre la atmósfera y el océano, entre otros. Al ser mantenido por la comunidad, tiene una gran frecuencia de actualizaciones y un conjunto de parametrizaciones muy grande que permite utilizar distintos modelos de simulaciones de una gran variedad de aspectos físicos.

El WRF es un modelo euleriano, no hidrostático y compresible. Se dice que un modelo es euleriano, cuando utiliza un sistema de coordenadas fijo, con respecto a la tierra, y lagrangiano a aquellos que utilizan un sistema de referencia que se ajusta al movimiento atmosférico. El primer grupo representa las ecuaciones de movimiento y las transformaciones químicas de los contaminantes en la atmósfera utilizando este sistema fijo de coordenadas, lo que permite hacer una partición del dominio en celdas. Los modelos hidrostáticos son aquellos que suponen un equilibrio, en el eje vertical, entre presión y la fuerza gravitatoria, mientras que los no-hidrostáticos incluyen ecuaciones explícitas para el cálculo de dichos factores. El WRF es compresible dado que considera las variaciones de densidad sufrida por los diversos fluidos involucrados.

El código del WRF se comenzó a escribir en FORTRAN y C en 1998, culminándose su primer versión en diciembre del 2000. Actualmente está en su versión 3.4.1 liberada en agosto del año 2012, la cual ronda el medio millón de lineas de código distribuidas en alrededor de 200 módulos. El modelo se basa en una arquitectura modular, y puede ser configurado tanto para investigación como para aplicaciones operacionales. El código se encuentra organizado en tres capas, como muestra la Figura 2.3, la primera capa denominada "Driver", es la encargada de la lectura y primer procesamiento de datos, realizando interpolaciones y generando las estructuras auxiliares necesarias. Existe una capa intermedia que oficia de mediador entre la capa externa y la capa del modelo, teniendo como responsabilidad el avance del tiempo y los cálculos realizados por los *cores* dinámicos (ARW o NMM). Finalmente se encuentra la capa que contiene las rutinas propias del modelo WRF, escritas de manera independiente de la forma y tamaño del dominio. Para lograr esto se le impide a los módulos aquí incluidos realizar operaciones de entrada-salida, detener o abortar la ejecución o utilizar información de otros módulos.

Los dos posibles *cores* previamente mencionados, "The Advanced Research WRF" (ARW) y el "Nonhydrostatic Mesoscale Model" (NMM) son los encargados de los cálculos de advección [4], gradientes de presión [5], aplicación del efecto coriolis [6], cálculos de flotabilidad [7], y del avance del tiempo. La elección del módulo a utilizar se realiza en tiempo de compilación. El *core* ARW fue diseñado principalmente con fines de investigación, pero también es utilizado para realizar predicciones del clima. Soporta varios agregados como el WRF-Chem que fusiona la emisión, transporte y transformaciones químicas de gases junto con la meteorología. El *core* NMM se concentra puramente en la predicción climática.

Existe un módulo generado automáticamente, que no suele ser incluido en ninguna de las tres capas, y que concentra la lógica necesaria para los accesos de entrada-salida, la comunicación entre procesos, definición de tipos de datos y de operaciones sobre estas estructuras. Este módulo llamado "registry" representa aproximadamente un 40 % de la cantidad total de líneas del código.

Para su ejecución el WRF particiona el dominio en piezas rectangulares, llamadas *patches*, que pueden ser asignadas a distintos procesadores si se busca paralelizar el trabajo entre varias CPUs. Estos *patches* a su vez se dividen en *tiles*, sobre los que pueden

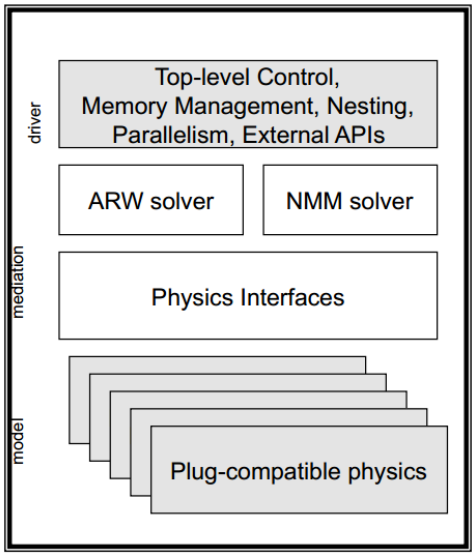

Figura 2.3: Estructuración en capas del código WRF, extraído de [22].

trabajar distintos hilos de ejecución, típicamente en plataformas paralelas con memoria compartida. Normalmente los *patches* necesitan datos de otros *patches* vecinos, en los *tiles* frontera, para estos casos es necesario propagar los cambios entre distintos *patches*. La capa intermedia es la encargada de iterar sobre el dominio, invocando métodos de la capa del modelo que sólo conocen la información de un *tile*.

Los datos que ingresan al WRF, provienen del "WRF Preprocessing System" (WPS), que se encarga de definir una grilla comprendida por la ubicación en el globo, números de puntos en la grilla, "nest locations" y distancias entre puntos. Junto a éstos se adjuntan un conjunto de valores estáticos, entre ellos, albedo (porcentaje de radiación reflejada en una superficie respecto a la radiación que incide sobre ella), elevación del terreno y textura del suelo, además de un conjunto de valores dependientes del tiempo, como son temperatura de las distintas capas del suelo, así como de la superficie del mar, humedad y presión. La salida del WPS provee una imagen tridimensional completa de la atmósfera en una región, conteniendo un eje vertical con los distintos niveles de la misma y dos ejes horizontales a nivel del suelo.

2.3. Utilización del WRF en la herramienta desarrollada

La versión del WRF utilizada por el IMFIA es el WRF-ARW [1], el mismo permite realizar diferentes configuraciones en las parametrizaciones de los distintos módulos del modelo.

En la Figura 2.4 se presentan los módulos del WRF, detallando los bloques de preprocesamiento del mismo, a donde ingresarán los datos iniciales. El bloque STATIC DATA corresponde a información de la base geográfica, como la topografía, rugosidad del terreno y otros. El bloque GRIB DATA contiene la información con condiciones iniciales y de frontera necesaria obtenida a partir de los pronósticos GFS. Los ejecutables GEOGRID UNGRIB y METGRIB corresponden a un pre-procesamiento de la información, la cual se ajusta a la grilla definida en la corrida específica. Por último, el ejecutable REAL acondiciona el formato de los datos para ingresar a los módulos de simulación del WRF.

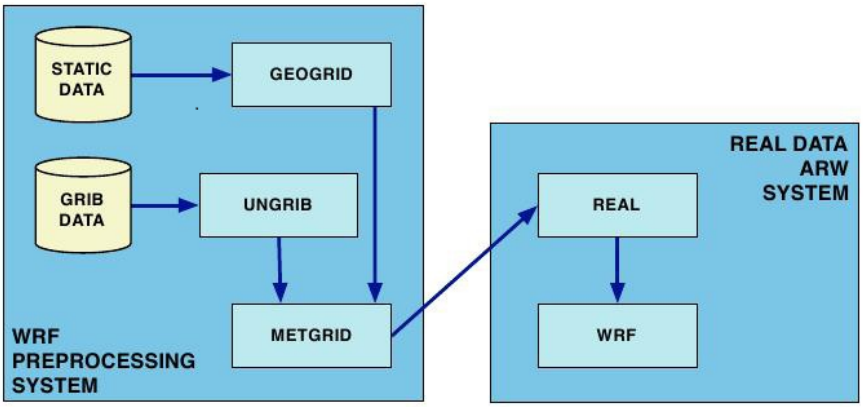

Figura 2.4: Representación gráfica de los módulos del WRF, extraído de [1].

La simulación se realiza generando cuatro niveles de grilla aumentando la resolución en cada paso a través del proceso de anidación, donde los niveles de grilla de mayor resolución asimilan la información que genera el modelo en los niveles de grilla de menor resolución, ajustando además en cada paso las parametrizaciones asociadas. Las grillas generadas tienen resoluciones de 30km × 30km, 10km × 10km, 3,3km × 3,3km y 1,1km × 1,1km. Cabe destacar que la simulación se ejecuta paralelizando cálculo en CPU con la utilización de 8 cores gracias a la configuración `smpar` (del inglés, shared memory parallel) del WRF, la cual utiliza la API OpenMP [9] para la paralelización.

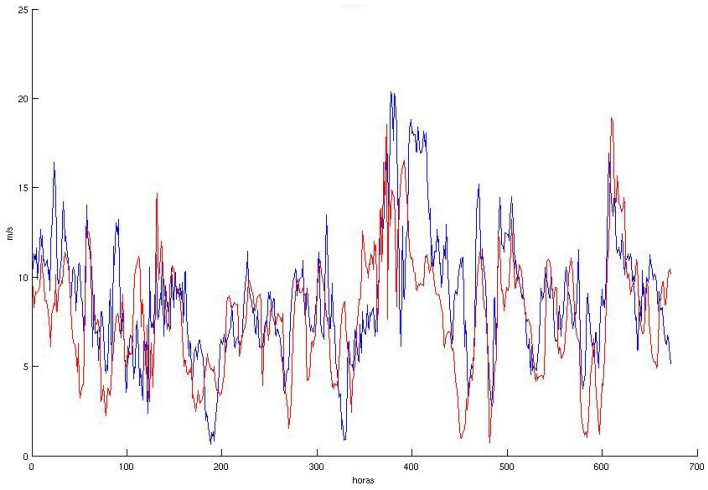

Figura 2.5: En azul velocidad del viento obtenida, en rojo velocidad del viento pronosticada en la "Sierra de los Caracoles", extraído de [1].

El WRF genera pronósticos de componentes W-E y N-S. A partir de los mismos se calcula la magnitud de la velocidad en m/s y la dirección en grados. En las Figuras 2.5 y 2.6 se muestran gráficos de velocidad y dirección medidos y pronosticados en la "Sierra de los Caracoles". La contrastación se realizó contra datos de velocidad y dirección medidos por UTE con un sistema de medición de anemómetros y veletas instalas en distintos puntos del territorio nacional.

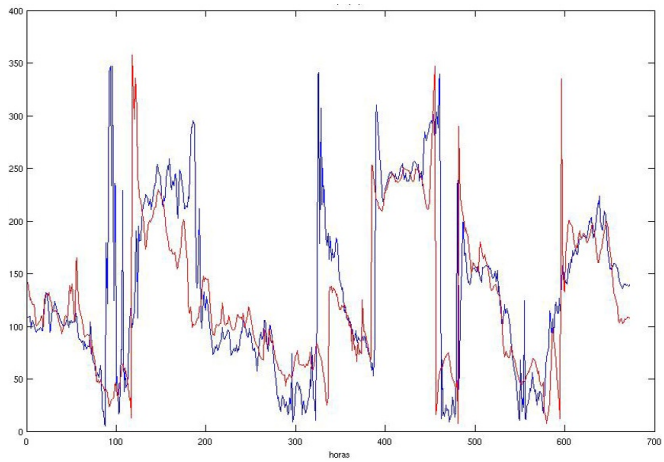

Figura 2.6: En azul dirección del viento obtenida, en rojo dirección del viento pronosticada en la "Sierra de los Caracoles", extraído de [1].

Capítulo 3

GPGPU

3.1. Introducción

La sigla GPGPU es el acrónimo de "General-Purpose Computing on Graphics Processing Units", es decir, utilizar las características y capacidad de las tarjetas gráficas para la resolución de problemas computacionales de índole general, en contraste con los problemas de computación gráfica.

En este capítulo se mencionan aspectos generales que motivan la utilización de GPUs para la resolución de problemas de propósito general, como así también aspectos de la arquitectura CUDA que se utiliza en este trabajo. También se ofrece un resumen del estado del arte en el uso de GPUs para acelerar modelos numéricos de gran porte, con particular atención en el WRF.

3.2. Motivación

Más allá de comparaciones de capacidad de cómputo que dependen de avances del punto de vista de hardware, conviene notar que las CPUs están diseñadas para ejecutar instrucciones de propósito general correspondientes a una cantidad pequeña de hilos de ejecución de forma eficiente, pero los procesamientos de datos de, por ejemplo, modelos de simulación, involucran cálculos que, dada la naturaleza del problema, pueden distribuirse de forma independiente en una gran cantidad de hilos de ejecución. Por lo tanto, desde el punto de partida, la CPU no es el mejor dispositivo para el procesamiento de datos que involucren este tipo de cálculos. Es por ello que hoy en día se intenta abordar el procesamiento de datos con otros dispositivos como las GPUs, que en su esencia fueron diseñadas para solucionar problemas que implican la realización de cálculos masivamente paralelizables, llegando a veces a obtener tiempos de ejecución varios órdenes menores para ciertos problemas en comparación con lo logrado por CPUs.

Existen múltiples aspectos que incentivan el estudio de técnicas sobre el uso de GPUs para la resolución de problemas de propósito general. Uno de ellos es la evolución del poder computacional de las GPUs respecto a las CPUs. La métrica generalmente utilizada para medir la capacidad de cómputo que se utiliza es FLOPS (Floating Point Operations per Second). Si se compara como evoluciona la capacidad de cómputo, se aprecia que la tasa de crecimiento en las GPUs es significativamente mayor que en las CPUs. En la Figura 3.1 se ofrece un análisis comparativo de esta evolución. Además del poder computacional interesa también su costo monetario, donde los números muestran la conveniencia costo/FLOPs de las GPUs por sobre las CPUs.

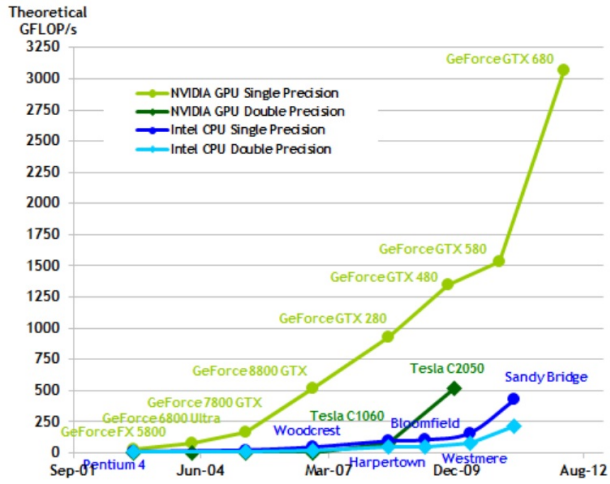

Figura 3.1: Evolución en GFLOPS de GPU vs CPU, extraído de [38].

A pesar de que las perspectivas eran alentadoras, existieron restricciones para poder sacar provecho de tal capacidad de cómputo. Una de ellas era la necesidad de poseer conocimientos muy específicos sobre GPUs, lo que provocaba una lenta curva de aprendizaje. Ésto, junto con la fuerte dependencia del conocimiento con el hardware, el cual cambia continuamente, desmotivaba a potenciales usuarios.

Recientemente han aparecido lenguajes de programación de alto nivel para programar las GPUs, esto ha motivado fuertemente a explorar el campo de GPGPU y ha repercutido en una masificación en la utilización de las GPUs para la resolución de problemas de propósito general. El desarrollo de arquitecturas como CUDA y OpenCL ha abierto las puertas para que personas que no son técnicamente expertas en GPUs puedan utilizar su poder de cómputo.

3.3. CUDA

El acrónimo CUDA, "Compute Unified Device Architecture", se refiere a la arquitectura de computación paralela definida por la empresa NVIDIA en 2007 [12] que sienta las bases para la utilización de tarjetas gráficas construidas por dicha empresa a partir de la serie 8. Dicha arquitectura es programable usando lenguajes de programación de propósito general, como C y FORTRAN, entre otros. La arquitectura busca aprovechar el diseño natural de las GPUs utilizadas en el procesamiento gráfico que implica generalmente cálculos masivamente paralelos en problemas de procesamiento general. Se distinguen tres componentes: Bibliotecas CUDA, CUDA Runtime y CUDA Driver.

Las bibliotecas CUDA, como por ejemplo CUFFT [39], CUSPARSE [40], CUBLAS [37], son bibliotecas que ya ofrecen rutinas de alto nivel que utilizan el poder de cómputo de la GPU para resolver problemas comunes en modelos matemáticos. Por ejemplo, CUBLAS ofrece una implementación de BLAS (Basic Linear Algebra Subroutines) y CUFFT ofrece rutinas para el manejo de Transformadas de Fourier. CUDA Runtime ofrece las APIs utilizadas en lenguajes de alto nivel como C o FORTRAN que permiten interactuar con el dispositivo. Finalmente, el módulo CUDA Driver es el encargado de interactuar con el dispositivo en lo referente a transferencia de datos.

3.3.1. Arquitectura

La arquitectura CUDA, está compuesta por un arreglo escalable de multiprocesadores junto con distintos tipos de memoria y mecanismos de interacción con el resto de las componentes de la arquitectura.

La arquitectura modela a la GPUs como un dispositivo compuesto por un conjunto de multiprocesadores (SMs) capaces de ejecutar múltiples hilos al mismo tiempo. Estos contienen una determinada cantidad de cores que ejecutan los hilos llamados *CUDA cores*, los cuales permiten realizar operaciones sobre enteros y punto flotante. Cada multiprocesador del dispositivo ejecuta independientemente el mismo código, llamado *kernel*, sobre distintos datos. Dicho kernel es ejecutado por hilos que están organizados por bloques en una grilla como se muestra en la Figura 3.2. Cada bloque de la grilla contiene un conjunto de hilos que representan una ejecución del kernel. Éste es el aspecto fundamental en el que se basa la paralelización del trabajo realizado, ya que cada bloque es asignado a un SM distinto pudiendo así ejecutarse en forma concurrente varios bloques de la grilla. El dispositivo tiene un scheduler interno el cual se encarga de asignar a cada SM múltiples bloques. De ésta manera la arquitectura establece un mecanismo que abstrae el diseño de la solución en CUDA de las capacidades particulares del dispositivo, permitiendo así también mantener a futuro la utilidad de la solución. Además, este scheduler realiza cambios de contexto entre distintos bloques asignados a un SM prácticamente sin costo, lo cual puede ser muy útil para ocultar la latencia en el acceso a memoria.

En cada bloque, los hilos son agrupados en conjuntos denominados *warps* que son los que efectivamente son ejecutados en un instante de tiempo determinado por el SM correspondiente a su bloque. Cuando la ejecución de un warp implica una espera de acceso a memoria, el scheduler asigna otro warp al SM de forma de intentar mantenerlo ocupado el máximo tiempo posible. Como el dispositivo responde a la arquitectura "SIMT" (Single Instruction Multiple Threads), cada hilo tiene un flujo de ejecución propio, de forma que también impacta en la performance la llamada *divergencia de hilos* en un warp. Es decir, todos los hilos de un warp ejecutan todos los caminos resultantes de dicha divergencia en forma secuencial, donde cada uno estará activo o no según si satisface las condiciones de cada ramificación. Es por esto que a nivel de diseño del kernel se debe evitar ésto lo máximo posible de forma de intentar mapear ramificaciones de ejecución directamente en conjuntos de hilos de un mismo warp, para así a nivel de warp todos sus hilos ejecutan una única ramificación evitando la divergencia.

Jerarquía de Memoria en CUDA

Otro aspecto determinante en CUDA es el uso de una Jerarquía de Memoria. En la Figura 3.2 se puede apreciar la jerarquía de memoria en la arquitectura CUDA. Cada hilo posee memoria privada que se compone tanto de registros como de memoria local, donde la primera es varios órdenes más rápida que la segunda pero significativamente menor en cantidad. La memoria local no es manejable por el usuario, sino que la GPU la utiliza cuando se agotan los registros en tiempo de ejecución. El siguiente nivel es la memoria compartida a nivel de bloque; ésta memoria es compartida entre todos los hilos de un bloque, lo que permite la cooperación entre ellos. Finalmente se tiene la memoria global, la memoria constante y la memoria de texturas que son compartidas entre todos los bloques de una grilla, lo que permite la cooperación entre bloques. Los distintos tipos

de memoria aquí descritos son utilizados tanto para lectura como escritura, con excepción de la memoria constante y de texturas que son solamente de lectura.

Resulta de extrema importancia entender las diferencias entre los diferentes tipos de memoria. Tanto la memoria de registros como la memoria compartida es "on-chip", por lo que la latencia de acceso es muy baja, resultando en la memoria más rápida disponible en el dispositivo. La memoria local de los hilos (la cual se mapea a memoria global), la memoria global, la memoria constante y la memoria de texturas son "off-chip", por lo que su acceso tiene un alto costo. De todas formas, la memoria constante, aunque es escasa incluye un caché optimizado para el almacenamiento de constantes [38]. En últimas versiones de la arquitectura CUDA, la memoria global también es cacheada. La memoria de texturas tiene características similares a la memoria constante, ya que posee un caché especial que aprovecha la localidad de datos, notar que esta memoria es la original de las GPUs para almacenar imágenes, y cuando se accede a un punto de éstas, se suele leer la imagen completa. La diferencia entre las propiedades de los distintos tipos de memoria es de extrema importancia para el buen diseño del código de un kernel, ya que si se logra que la memoria utilizada por los hilos sea mapeada a memorias de más rápido acceso, se puede obtener un mejor rendimiento.

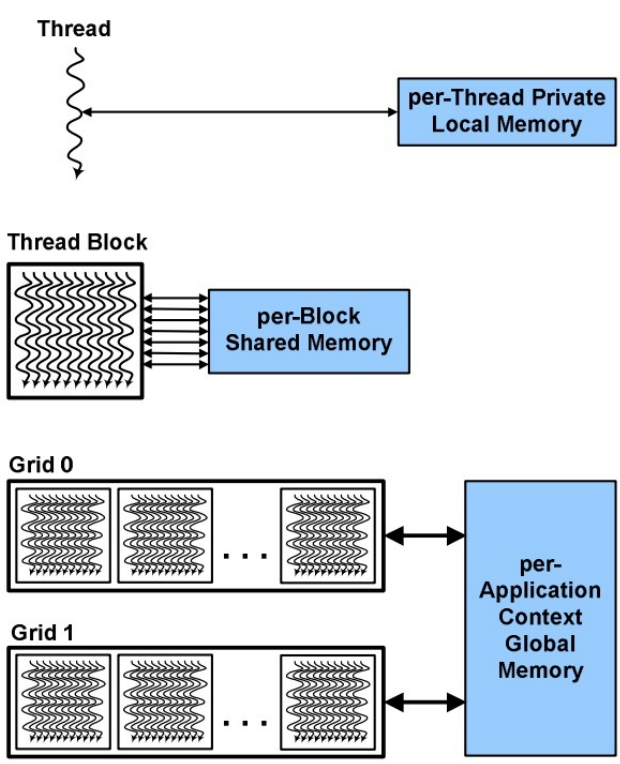

Figura 3.2: Memoria en CUDA, describe la jerarquía de CUDA respecto a hilos, bloques y grillas, y su relación con los distintos tipos de memoria, extraído de [36].

Evolución histórica

La arquitectura G80 fue la primer generación que dio soporte a CUDA [34]. Hasta antes de su aparición, el uso de GPUs para propósitos de procesamiento general involucraba por parte del programador un conocimiento muy cercano respecto a las APIs gráficas

ofrecidas por la GPU, intentando forzar el modelo de procesamiento gráfico para el procesamiento general. Ésto no permitía una fácil adopción de la GPU como recurso para resolver problemas de procesamiento general y sólo podía ser utilizado por un pequeño grupo de personas especialistas en la materia. El dar soporte a CUDA permitió abrir puertas a programadores no especialistas en GPU, por estar basado inicialmente en C y ser un lenguaje de programación general conocido. Esta generación también fue la primera arquitectura unificada a nivel de hardware. Antes de la misma, las distintas etapas del procesamiento gráfico eran resueltas por diferentes componentes como los *pixel shaders* y los *vertex shaders*, con su aparición, estas etapas pasaron a ser resueltas por los cores de la GPU, lo que permitió el mejor aprovechamiento del hardware.

Luego de que ésta arquitectura sentara las bases de la primera generación de arquitecturas orientadas a procesamiento general, y aprendiendo de la experiencia de G80 surge la arquitectura GT200 [35], la que incorpora una serie de cambios que mejoran la performance. Entre ellas, un incremento en la cantidad de CUDA cores, aumento de la memoria asociada a registros de cada core, mejores técnicas de acceso a la memoria global, soporte para operaciones de doble precisión, entre otras. Se puede decir que la arquitectura GT200 resulta ser una revisión menor y una mejora de la arquitectura G80.

En el año 2010 se lanza la tercera generación de CUDA llamada Fermi [36]. Ésta nueva arquitectura resulta ser una mejora considerable respecto a las arquitecturas anteriores. Algunos de los cambios más importantes son el cumplimiento de estándares referentes a operaciones con números en punto flotante en simple y doble precisión, introducción de memoria caché, incremento de capacidad en memoria por multiprocesador, y mejora de tiempos referentes a cambios de contexto entre hilos de ejecución y operaciones atómicas.

En general la arquitectura Fermi dota a cada SM de 32 procesadores CUDA, multiplicando por 4 dicha cantidad respecto a la arquitectura GT200. Incorpora mejoras en la precisión en operaciones de punto flotante, como así también, implementa la multiplicación entera de 32 bits, sustituyendo a la de 24 bits utilizada en arquitecturas anteriores.

Otro cambio importante es la unificación del espacio de direcciones de memoria. En arquitecturas anteriores el direccionamiento de memoria estaba identificado por un offset y una identificación de qué tipo de memoria era referenciada, memoria global o compartida, entre otros. En Fermi una instancia particular del espacio de direcciones ya define el tipo de memoria accedida además del offset, permitiendo dar soporte completo a lenguajes como C++. La capacidad de memoria total de tipo registro es de 128KB, y la cantidad de memoria compartida y de caché L1 es un aspecto configurable, permitiendo configuraciones de 16KB/48KB o 48KB/16KB por SM para ofrecer una mayor flexibilidad que permite explotar al máximo los recursos de hardware según las necesidades específicas del programa.

También en Fermi, a diferencia de arquitecturas anteriores, se permite ejecutar simultáneamente más de un kernel en la GPU, lo cual resulta ser otra medida que intenta evitar tiempos muertos o de poca utilización de la capacidad de procesamiento.

Finalmente, además de otras mejoras no mencionadas que podrán ser consultadas en [36], se incorpora un cambio considerable en el scheduler a nivel de SM, llamado *Dual Warp Scheduler*. En arquitecturas anteriores, cada SM sólo ejecutaba en un instante de tiempo un único warp, pero en ésta arquitectura se incorpora un segundo scheduler por SM permitiendo así ejecutar concurrentemente dos warps en un instante de tiempo. La intención de esta mejora es alcanzar al máximo posible el uso de todos los componentes

del SM, ya que, por ejemplo, mientras un warp no hace uso de componentes como las SFUs[1], se intenta aprovechar dichos componentes en otras instrucciones del kernel que están pendientes de ejecutar en otro warp, intentando alcanzar así la máxima performance a nivel de hardware.

En las Figuras 3.3 y 3.4 se presenta en forma gráfica la arquitectura Fermi (GF 100) de una GPU en general y la arquitectura a nivel de SM respectivamente.

Recientemente, se ha desarrollado una nueva arquitectura con soporte a CUDA llamada Kepler. Debido a que la misma escapa del alcance de este proyecto, no se entrará en detalle sobre su diseño. Para mayor información consultar [41].

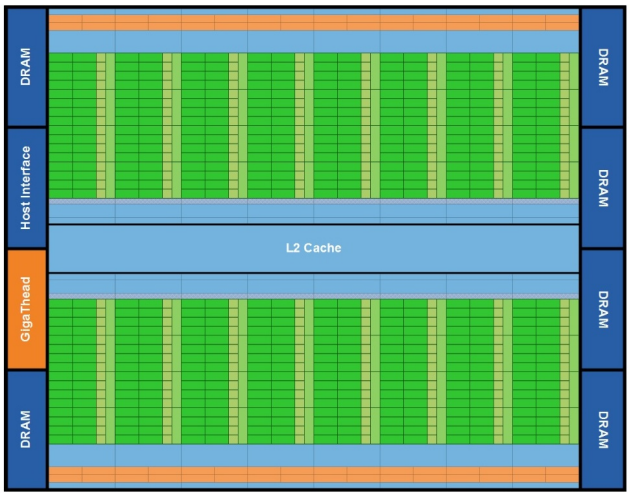

Figura 3.3: Arquitectura Fermi (GF100), extraído de [36].

Finalmente en la Tabla 3.1 se puede apreciar un resumen comparativo entre las tres arquitecturas mencionadas, lo que permite apreciar la tasa de crecimiento del poder computacional entre generaciones, mostrando un futuro prometedor.

Compute Capabilities

Para englobar las características de las tarjetas gráficas, NVIDIA utiliza el término *Compute Capabilities*. Dicha categorización está definida por dos números; el primero implica un cambio generacional mientras que el segundo refiere a una revisión sobre una arquitectura ya definida. Actualmente existen tres generaciones de tarjetas: la versión 1.x es la correspondiente a la arquitecturas G80 y GT200, la versión 2.x es la correspondiente a la arquitectura Fermi y la recientemente publicada arquitectura Kepler se identifica como 3.x.

En la Tabla 3.2 se presenta un cuadro resumido, con algunos datos de interés que se vieron modificados a lo largo de las diferentes Compute Capabilities. Para mayor detalle consultar la guía oficial de CUDA ofrecida por NVIDIA [38].

[1]SFUs: Unidades similares a las ALUs dedicadas al cómputo de funciones especiales como seno, coseno, raíz cuadrada, entre otras.

GPU	G80	GT200	Fermi (GF100)
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double Precision FP Capability	None	30 FMA ops/clock	256 FMA ops/clock
Single Precision FP Capability	128 MAD ops/clock	240 MAD ops/clock	512 FMA ops/clock
Special Function Units (SFUs)/SM	2	2	4
Warp schedulers (per SM)	1	1	2
Shared Memory (per SM)	16 KB	16 KB	Config. 48/16 KB
L1 Cache (per SM)	None	None	Config. 16/48 KB
L2 Cache	None	None	768 KB
ECC Memory Support	No	No	Yes
Concurrent kernels	No	No	Up to 16
Load/Store Address Width	32-bit	32-bit	64-bit

Tabla 3.1: Comparativa entre las tres arquitecturas CUDA, extraído de [36].

	1.0	1.1	1.2	1.3	2.x	3.0
Bloques residentes por SM	8					16
Warps residentes por SM	24		32		48	64
Dimensiones del grid	2					3
Máximo valor del eje x del grid	65535					65536 * 65536 -1
Máximo valor de los ejes y,z del grid	65535					
Máximo valor de los ejes x,y del bloque	512					1024
Máximo valor del eje z del bloque	64					
Cantidad de registros de 32 bit por SM	8k		16k		32k	64k
Máximo número de instrucciones por kernel	2 millones					512 millones

Tabla 3.2: Resumen de la evolución histórica de las Compute Capabilities.

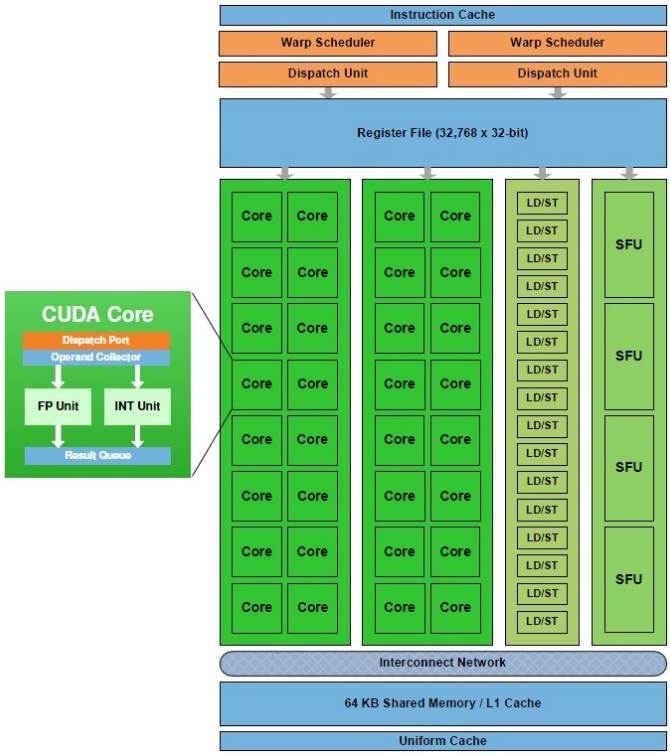

Figura 3.4: Arquitectura de un SM en Fermi (GF100), extraído de [36].

Accesos a memoria

Hasta Fermi, las operaciones que involucran acceso a memoria global se realizaban de a medio warp. Un acceso a datos allí ubicados puede desencadenar de una a dieciséis operaciones, dependiendo de la ubicación de los datos y de la capacidad de la tarjeta en unificar estos accesos. Esto se debe a que los accesos a memoria se realizan de a segmentos y estos están alineados a múltiplos de 32, 64 o 128 bytes, por lo tanto si los dieciséis hilos del medio warp necesitan leer cuatro bytes correlativos y debidamente alineados, se pueden fusionar estos accesos, en una sola lectura de 64 bytes. Esta capacidad de unificar accesos recibe el nombre de *coalesced*. La misma fue mejorando en las distintas Compute Capabilities (CC). En versiones iniciales (CC 1.0 o 1.1) para lograr esta unificación, los hilos debían operar con direcciones de memoria secuenciales (el i-ésimo hilo accede a la i-ésima palabra), en caso de que cada hilo accediese a 4 bytes, los 64 bytes correspondientes debían estar contenidos en el mismo segmento de 64 bytes y en orden, de no cumplirse eran necesarios dieciséis accesos independientes. Posteriormente (CC 1.2 o 1.3), se permiten accesos de los hilos en desorden dentro de un mismo segmento, incluso a una misma palabra, y en caso de que no cumplan con estas restricciones los dieciséis hilos pueden dividirse en subconjuntos, evitando así generar dieciséis operaciones a memoria. Finalmente (CC 2.x) se agrega un caché L1 para cada SM y una caché L2 compartida por todos los SM. Los accesos a memoria se realizarán primero a la caché y en caso de producirse un fallo en el acceso se accederá a la memoria global de forma similar a CC 1.2 y 1.3.

Los accesos a memoria compartida también se producen a nivel de medio warp. Esta memoria se divide en módulos de 32 bits llamados bancos, organizados de forma tal que palabras sucesivas de 32 bits pertenecen a bancos sucesivos. Cada banco puede atender solamente una solicitud por ciclo, por lo tanto en caso de que más de un hilo intente

acceder al mismo banco en un mismo ciclo, se deberá serializar el acceso, esto hace que se requieran tantos ciclos como el máximo número de accesos al mismo banco. Esto cambia si todos los hilos que acceden al banco leen del mismo, en este caso se realiza un broadcast entre los hilos y no se serializa.

Resumen

Llegados a este punto se tiene una imagen general de la arquitectura, comprendiendo lo indispensable para entender el diseño básico de una solución en CUDA. Aspectos más profundos de la arquitectura, como características propias de cierta versión del dispositivo a utilizar pueden ser relevantes para la toma de decisiones, sobretodo para su optimización, pero es necesario balancear la especificidad de la solución con una más general, que al no depender de versiones específicas de dispositivos pueda escalar más fácilmente en futuros dispositivos.

3.4. Aceleración del WRF utilizando GPUs

En esta sección se relevan y discuten los principales aportes referentes a la utilización de GPUs para acelerar el cómputo del WRF. Algunos de los trabajos se encuentran recopilados en el sitio web de "GPU Acceleration of NWP" [19] dirigido por J. Michalakes.

Uno de los principales aportes es el artículo de J. Michalakes y M. Vachharajani [20] gracias a su claridad y nivel de detalle. Los autores proponen en este trabajo la aceleración del módulo WSM5 del WRF, el mismo se utiliza para representar condensación, diferentes tipos de precipitación y efectos termodinámicos relacionados. Es uno de los módulos que implementa componentes físicos del WRF y a pesar de sólo presentar un 0.4 % del código total, suele insumir un cuarto del tiempo de procesamiento total en un solo procesador (un core).

El trabajo del WSM5 consiste en la realización de cálculos en un dominio del WRF donde, para cada punto de la grilla, se recorre la columna vertical realizando cálculos en los distintos niveles de la atmósfera, que resultan ser independientes de otros puntos de la grilla. Dependiendo del estado de la atmósfera se deberán computar aproximadamente 2400 operaciones de punto flotante para cada nivel de cada punto de la grilla.

Para acelerar el módulo con CUDA, primero se implementó la rutina en C para luego escribir el kernel necesario. Este se diseñó para que cada hilo sea encargado de ejecutar los cálculos correspondientes a un punto de la grilla. Para esto fue necesario convertir arrays multidimensionales en arrays unidimensionales y, para acelerar más los cálculos, se utilizó memoria compartida, sin embargo la misma resultó ser muy escasa para la cantidad de hilos de un bloque, por lo tanto solamente se utilizó para almacenar los datos más accedidos.

A pesar de que los resultados de los cálculos de punto flotante fueron ligeramente diferentes entre GPU y CPU, las diferencias en las salidas gráficas de ambos módulos son indistinguibles. En la evaluación de distintos benchmarks, se utilizó como escenario "The Storm of the Century" (SOC), un dominio de 30km de resolución con una grilla de 71×58 puntos y 27 niveles verticales, y se comparó el tiempo de ejecución entre un CPU Pentium-D 2.8GHz y un GPU 8800 GTX. Los resultados reportados en GPU resultaron ser alrededor de 17 veces más rápidos que en CPU aún contando transferencias entre la

memoria del host y del device. Los autores consideran que este trabajo se realizó con poco esfuerzo y dinero con resultados iniciales muy alentadores.

Otro trabajo de los mismos autores [21] se enfoca en el módulo encargado de la advección de distintos elementos. Se define advección como la variación de un escalar en un punto dado por efecto de un campo vectorial [4], siendo estos escalares o trazadores para el WRF concentraciones de vapor, nubes, lluvia, entre otros. En este trabajo se busca optimizar el cómputo necesario para calcular la variación de los trazadores involucrados, que comúnmente son varias decenas. Para llevar ésto a cabo no se pudieron aplicar soluciones similares a la del trabajo desarrollado previamente, ya que hay transmisión de datos entre las tres dimensiones (lo cual impide la división del problema en dimensiones) y hay sólo 0.76 operaciones de punto flotante por acceso a memoria. Pero por contrapartida dada la gran cantidad de trazadores que suelen haber, los accesos a memoria del host pueden ser solapables con operaciones realizadas en la GPU, logrando así ocultar la latencia que agregan las lecturas de datos (mientras se realizan los cálculos de un trazador, se van cargando asíncronamente, en otro buffer los datos necesarios para el próximo). Ésto se implementó sobre la memoria de texturas, la cual cuenta con suficiente tamaño como para albergar éste voluminoso conjunto de datos, y es muy eficiente para la lectura de datos en lugares dispersos (no necesita que sean secuenciales para un buen rendimiento). Los resultados obtenidos, reflejan una mejora de 6.7× por sobre el tiempo necesario en CPU, incluyendo tiempo de transferencias entre la memoria del host y del device. Los mismos se consiguieron utilizando un CPU Intel Xeon E5440 y una GPU Tesla C1060 sobre un modelo de 27km de resolución de 134×110 puntos de grilla y 35 niveles verticales, con 81 trazadores.

En el último trabajo [17] del sitio antes mencionado, los autores buscan optimizar una implementación sobre la cinética química en 3 plataformas multi-core, una con 2 CPUs Intel Xeon de la serie 5400, otra con una GPU NVIDIA Tesla C1060 y otra con 2 CBEA chipsets. El CBEA es un tipo de procesador utilizado en la industria de los videojuegos, el mismo contiene muchos cores conectados por medio de un bus entre ellos y con una memoria on-chip muy rápida de buena capacidad [10].

En este caso se optimizó el módulo "Regional Acid Deposition Model, versión 2" (RADM2) perteneciente al WRF-Chem, una versión del WRF enfocada en los vientos y a la química atmosférica. Éste módulo es comúnmente utilizado para predecir la evolución de las concentraciones de oxidantes y otros contaminantes en el aire. Concentra su costo en el algoritmo de "Rosenbrock", que incluye la construcción de una matriz Jacobiana y una descomposición LU, entre otros, para resolver una gran cantidad de ecuaciones diferenciales. Este algoritmo es ejecutado en cada punto de la grilla, donde los cambios en la concentración de sustancias dependen solamente de las concentraciones químicas y la meteorología en ese punto. Es aquí donde reside la capacidad de paralelización de este algoritmo. Debido a la gran cantidad de memoria requerida, la optimización en CUDA del módulo se desarrolló en parte en el CPU, sin embargo, las etapas fundamentales de la misma fueron desarrollados en distintos kernels, uno para la conformación de la matriz Jacobiana, otro para la descomposición LU, etc. Esta división en kernels sirvió entre otras cosas para resolver un cuello de botella que impedía que ejecutaran la mayor cantidad posible de cores en paralelo, ya que la cantidad de memoria utilizada en los cálculos era muy amplia, entonces la memoria compartida y los registros utilizados por cada hilo impedían que se ejecutara una mayor cantidad de hilos en paralelo. Los datos necesarios para los cálculos se mantuvieron en la memoria global y fueron accedidos de forma coalesced.

Como datos de entrada de cada implementación se utilizó una grilla de 40×40 con 20 niveles verticales. La aceleración obtenida con la GPU Tesla C1060 incluyendo tiempo de transferencias fue de 8.5×, con 2 CPU Intel Xeon 5400 fue de 7.5×, y con 2 PowerXCell 8i (CBEA chipset) fue de 41×. Los autores concluyen que a pesar de no haber obtenido grandes resultados con la GPU, el presupuesto utilizado en la misma fue el menor de las tres plataformas. Además, programar para un procesador CBEA presenta mayor complejidad que en GPU y se requiere conocer con detalle su arquitectura.

Otro artículo interesante sobre la temática es el presentado por J. Delgado *et al.* [18]. En este artículo se plantea estandarizar un método para portar aplicaciones desarrolladas en FORTRAN a CUDA C, ya que una buena cantidad de aplicaciones científicas con similares características (entradas y salidas en forma de arrays multidimensionales) están escritas en FORTRAN. El mismo hace especial hincapié en el WRF ya que este es fácil de portar debido a su modularidad. Se resalta la importancia de este trabajo dada la interesante metodología propuesta, la cual tiene aplicación directa en los objetivos de este proyecto.

El trabajo se apoya en el artículo de Michalakes y Vachharajani [20] descrito previamente, y además busca estandarizar el método utilizado. Se plantean las siguientes etapas: Profiling, Desarrollo, Testeo y Optimización. En la etapa de profiling se buscará determinar con distintas herramientas y casos de prueba cuáles son los módulos que consumen la mayor cantidad del tiempo. Para el desarrollo se recomienda portar en primera instancia de FORTRAN a C y luego a CUDA C, ya que portar directamente podría significar la inclusión de muchos errores por la naturalidad de estos lenguajes. Luego se construirá un driver y se testeará el módulo en forma aislada y se comparará la salida de GPU con la salida de CPU gráficamente con "difference plots" (gráficas de diferencias) debido a que una comparación bit a bit sería errónea por los redondeos de punto flotante. Para la etapa de optimización se pueden utilizar herramientas brindadas por CUDA, para ver la utilización real de recursos al correr el programa, o herramientas manuales. Dentro de las últimas se podrán realizar, tests con múltiples configuraciones de hilos y bloques, tratar de usar memoria compartida y registros e intentar acceder la mayor cantidad de veces a la memoria de forma coalesced, entre otras.

La propuesta es validada portando el módulo SWRAD del WRF a CUDA C logrando reducir a cerca de la mitad el tiempo de ejecución incluyendo transferencias, utilizando una GPU 9400M y comparándola con un CPU Core 2 Duo de 2.26 GHz. Cabe destacar que la GPU 9400M es una GPU de bajas prestaciones.

Otro artículo relevado es el de G. Ruetsch *et al.* [14]. Este artículo aborda el portado del componente Long-Wave Rapid Radiative Transfer Model (RRTM) a CUDA FORTRAN. En su introducción se hace especial hincapié en el uso de CUDA FORTRAN por sobre CUDA C. En particular se hace mención de la necesidad de poder escribir código CUDA en lenguaje FORTRAN ya que a pesar de que es posible mezclar código en FORTRAN con secciones de CUDA C, no resulta ser lo más natural. Además, se argumentan otras ventajas respecto a claridad de código y posible ventaja en robustez respecto a CUDA C. Luego se realiza una descripción del módulo RRTM, donde se pone a la vista los aspectos que sugieren explotar el uso de GPUs para su cálculo. Al igual que en el artículo [20], para portar el código se aprovecha la independencia entre las distintas columnas verticales de los puntos del dominio, donde para cada una se realiza el cálculo de la función del módulo. Además de este aspecto, se profundiza aun más en el análisis de cálculos hasta llegar, en ciertos puntos, a lograr un nivel de independencia a nivel de puntos del dominio

mucho más fino que por columnas, permitiendo así un mayor nivel de paralelismo y por lo tanto un mejor rendimiento. Finalizada la etapa de análisis de la representación del dominio, se hace un análisis de configuración de ejecución del kernel. En esta etapa se utilizan múltiples kernels, lo cual permite resolver lo estudiado en secciones anteriores del artículo, donde existen distintos niveles de paralelismo: a nivel de puntos del dominio, o a nivel de columnas del dominio. Es así que el uso de múltiples kernels permite explotar al máximo el nivel de paralelismo en distintas partes, además de intentar utilizar al máximo la GPU, ya que los mismos pueden ocultar tiempos de espera resultantes de transferencia de memoria entre CPU y GPU. Luego se analiza el uso de ciertos tipos de memoria que ofrece la GPU, y el impacto de la divergencia de hilos en el rendimiento. En este último punto se mencionan consejos de vital importancia para el diseño de soluciones en CUDA. Finalmente se presentan los resultados obtenidos, para los cuales se utiliza un CPU Intel Xeon E5540 corriendo a 2.83 GHz y una GPU Tesla C1060. La propuesta consigue una mejora de 8× a 10× (dependiendo de distintos parámetros de compilación) por sobre CPU incluyendo tiempo de transferencias. Sumado a ésto, se presenta la comparación de la exactitud de la versión de GPU respecto a la de CPU, obteniendo resultados de calidad similar.

Otros trabajos importantes en el área de optimización del WRF mediante GPUs son los realizados por B. Huang *et al.* [23, 24, 25, 26, 29, 30]. Los autores portaron varios módulos del WRF a CUDA C logrando muy buenos resultados, teniendo como meta seguir portando otros módulos hasta llegar a conseguir que el WRF ejecute completamente en GPU. Todos sus artículos siguen la misma línea, en dichos trabajos se incluye una descripción del módulo a portar, se introduce CUDA, se explica como portar el código en primera instancia a C y luego a CUDA C, y se finaliza comparando resultados de benchmarks del módulo ejecutando en CPU y en GPU. En uno de sus trabajos [23], se porta el módulo "Goddard Shortwave Radiation", éste incluye la absorción y dispersión de radiación solar en nubes, moléculas, superficies, entre otros. Para portar el código, el mismo se traduce a C y se verifica que sea correcto para luego traducirlo a CUDA C, de manera tal de que cada hilo compute valores de una posición espacial en todos los niveles de la atmósfera siguiendo la estrategia ya vista en otros artículos como en [20]. Además se realizan optimizaciones interesantes como utilizar dos kernels que ejecutan en paralelo para solapar la transferencia de datos a GPU y la ejecución a partir de los mismos, cambiar la cantidad de caché L1 configurándola en 48KB, convertir arrays temporales en valores individuales para ser almacenados en registros y utilizar acceso coalesced a memoria, entre otros. Para finalizar se realizan pruebas utilizando una configuración con dos GPUs NVIDIA GTX 590 y un CPU Intel i7 970, y como datos de entrada "CONUS", un dominio de 12km de resolución con 433×308 puntos y 35 niveles verticales. Se obtienen aceleraciones, utilizando doble precisión, de 116× y 141×, con y sin inclusión del tiempo de transferencia de datos. Además se realizan pruebas utilizando simple precisión y aritmética menos precisa (`use_fast_math`), donde los resultados alcanzan aceleraciones de 536× y 259×; sin embargo, a diferencia de los resultados con doble precisión, estos prueban tener un error poco deseable ya que grandes errores en predicciones podrían ocurrir debido a pequeños errores en el cómputo de la radiación.

Otro trabajo de los mismos autores [24] porta el módulo "Stony Brook University". Este es un módulo de microfísica similar al WSM5 que incluye procesos en nubes, vapor de agua y precipitaciones. Para portar el código, se traduce primero a C de forma de que luego traducir a CUDA sea sencillo, por ejemplo, se cambian arrays temporales por valores escalares que son recomputados si es necesario, ya que recomputar valores en CUDA en ocasiones es más rápido que transferirlos a memoria global. Luego de corroborar que el

código C funciona correctamente, se traduce a CUDA C utilizando la misma estrategia y optimizaciones que en el trabajo anterior. Sin embargo, en este caso se utilizan tres kernels que ejecutan en paralelo ya que la transferencia de datos es masiva y prueba ser un factor limitante en este módulo, y no se utiliza acceso coalesced, ya que para utilizarlo se deben transferir más datos a memoria global, y esta transferencia demora más que la optimización resultante de utilizar acceso coalesced. Se realizan pruebas utilizando la misma configuración de hardware y los mismos datos que en el trabajo anterior obteniendo aceleraciones de 352× y 896×, con y sin tomar en cuenta el tiempo de transferencia de datos respectivamente. Se utiliza simple precisión y `use_fast_math` ya que, generalmente, de esta manera los resultados no presentan errores notables. Se puede ver en los resultados cómo el factor limitante es la transferencia de datos, sin embargo, a medida que más módulos del WRF se porten, menos datos van a tener que ser transferidos a GPU y los tiempos van a disminuir.

Trabajo	**Módulo**	**GPU**	**CPU**	**Speedup máximo**
J. Michalakes, *et al.* 2008 [20]	WRF Single Moment 5 Cloud Microphysics (WSM5)	8800 GTX	Pentium-D 2.8 GHz	17×
J. Michalakes, *et al.* 2009 [21]	WRF Fifth Order Positive Definite Tracer Advection	Tesla C1060	Intel Xeon E5440	6.7×
J. Michalakes, *et al.* 2009 [17]	Regional Acid Deposition Model, ver. 2 (RADM2)	Tesla C1060	Intel Xeon E5440	8.5×
G. Ruetsch, *et al.* 2010 [14]	Longwave Radiation Physics (RRTM)	Tesla C1060	Intel Xeon E5540	8×-10×
J. Delgado, *et al.* 2011 [18]	Shortwave Radiation Physics (SWRAD)	9400M	Core 2 Duo 2.26 GHz	2×
B. Huang, *et al.* 2011 [30]	WRF Double Moment 5 Cloud Microphysics (WDM5)	2×GTX 590	Intel i7 970	147×
B. Huang, *et al.* 2011 [29]	Purdue Lin Cloud Microphysics	2×GTX 590	Intel i7 970	156×
B. Huang, *et al.* 2012 [23]	Goddard Shortwave Radiation	2×GTX 590	Intel i7 970	116×
B. Huang, *et al.* 2012 [24]	Stony Brook University 5-Class Cloud Microphysics (SBU-YLIN)	2×GTX 590	Intel i7 970	352×
B. Huang, *et al.* 2012 [25]	WRF Single Moment 5 Cloud Microphysics (WSM5)	2×GTX 590	Intel i7 970	357×
B. Huang, *et al.* 2013 [26]	Kessler Cloud Microphysics	2×GTX 590	Intel i7 970	70×

Tabla 3.3: Configuración y resultados de trabajos relevados.

Otros trabajos de los autores son [25, 26, 29, 30]. El primero tiene un objetivo ya mencionado con anterioridad en otros artículos; mejorar el tiempo de ejecución del módulo WSM5 utilizando GPUs como herramienta; el segundo, tercero y cuarto se plantean el mismo objetivo pero sobre los módulos WDM5, "Purdue Lin scheme" y "Kessler microphysics scheme", respectivamente. Éstos cuatro artículos, se basan en ideas ya mencionadas en los artículos anteriores de estos autores, parámetros de compilación, utilización de múltiples

kernels para mitigar tiempos de transferencia de datos, evaluación de distintas granularidades en el diseño de bloques y grilla, distintas configuraciones que ofrece la arquitectura Fermi respecto a las memorias caché, aplicación de padding para lograr acceso coalesced a memoria, entre otras. Para realizar pruebas, se utilizaron los mismos datos de entrada y la misma configuración de hardware de los trabajos anteriores. Los resultados obtenidos son muy alentadores. Se presentan los resultados con y sin tomar en cuenta el tiempo de transferencia de datos. En [25], se obtuvo una mejora de 357× y 1556×, en [30], 147× y 206×, en [29], 156× y 692× y en [26], 70× y 816×.

En la Tabla 3.3 se puede ver un resumen conteniendo datos relevantes en cuanto a configuración y resultados de los trabajos relevados.

Capítulo 4

Análisis del WRF

En el presente capítulo se expone el análisis de la ejecución del WRF, etapa fundamental para discernir que secciones del WRF y como afrontarlas para una buena optimización.

4.1. Plataformas utilizadas

Para el análisis, ejecución y optimización del WRF se utilizó una plataforma de hardware que incluye una CPU Intel y una GPU NVIDIA que son descritas en la Tabla 4.1:

#	**CPU**	**GPU**	**RAM**	**Sistema operativo**
1	Intel Core i3-2100 @3.10GHz (dual core HT)	GeForce GXT 480	8GB	Fedora 15 - 64 bits

Tabla 4.1: Plataforma utilizada.

El compilador de Fortran utilizado es el contenido en la versión 4.4.7 del GNU Compiler Collection, y el de CUDA es la versión 4.1.

4.2. Casos de prueba

El escenario utilizado para el análisis, ejecución y optimización del WRF, contiene información real para un pronóstico climático en el "Parque Emanuele Cambilargiu" del dia 15/07/2012, ejemplo que es representativo de un día de simulación. El escenario se utiliza para generar cuatro niveles de grilla, aumentando la resolución en cada paso de grilla, donde los niveles de grilla de mayor resolución asimilan la información que genera el modelo en los niveles de grilla de menor resolución. Las grillas generadas tienen resoluciones de 30km × 30km, 10km × 10km, 3,3km × 3,3km y 1,1km × 1,1km; mientras que sus dimensiones son de 74 × 61 puntos de grilla y 54 niveles verticales para la resolución de 30km × 30km, y 40×40 puntos de grilla y 54 niveles verticales para las demás.

4.3. Tiempos de ejecución del WRF

Los tiempos de ejecución del WRF se midieron para una ejecución secuencial y para una en paralelo utilizando la API OpenMP. La paralelización con OpenMP utilizada es la que se compila por defecto entre las opciones de compilación del WRF. La misma paraleliza distintas secciones del código, donde al ejecutarlas, se crean tantos hilos como

cores tenga el CPU[1] (virtuales o reales). En la Tabla 4.2 se muestran los tiempos de ejecución para las distintas configuraciones.

Plataforma	Tiempo	Cantidad hilos	Speedup
1	3:46:42 hs	1	-
1	2:06:46 hs	2	1.79×
1	1:43:23 hs	4	2.20×

Tabla 4.2: Tiempos de ejecución del WRF.

4.4. Profiling

Siguiendo las etapas del proyecto, se encuentra la de profiling, etapa fundamental del proyecto ya que establece los caminos a recorrer en etapas posteriores. Debido a esto, realizar un buen profiling es vital para alcanzar los objetivos planteados.

Se realizó un análisis de las herramientas disponibles y los distintos enfoques que existen para su funcionamiento. Por un lado se encuentran herramientas de profiling que realizan modificaciones en el código fuente en tiempo de compilación. Éstas incluyen código extra que realiza el relevamiento de información para la obtención de métricas. Otro enfoque es realizar el profiling en tiempo de ejecución, donde el profiler se comporta como depurador de la aplicación obteniendo la información en tiempo de ejecución sobre el código original.

Se decidió hacer el profiling con al menos una herramienta que trabajara con cada uno de los enfoques ya mencionados. En una primera instancia se utilizó el profiler *gprof* [27], el cual es considerado el profiler de-facto en ambientes académicos dado que está incluido en el proyecto GNU. Éste profiler, que trabaja bajo el primer enfoque explicado anteriormente, implica la utilización de flags especiales del compilador gfortran. Luego se ejecuta el programa (WRF) y se generan salidas especiales a interpretar por *gprof*.

La otra herramienta utilizada fue *valgrind* [46], un profiler que trabaja bajo el segundo enfoque de los comentados. El modo de trabajo de ésta herramienta es comportarse como un wrapper de la aplicación, similar a una máquina virtual con un "CPU sintético", ejecutando el proceso, obteniendo las métricas y generando el informe para su posterior análisis.

Es necesario comentar el motivo de la utilización de dos enfoques distintos para el profiling, junto con las ventajas y desventajas que tiene cada uno.

En un principio realizar el profiling con más de una herramienta aumenta la probabilidad de obtener resultados que se ajusten a la realidad, dado que si una herramienta tiene un comportamiento inesperado por un motivo desconocido, el resultado del segundo profiler será completamente distinto. Aún así, hay que tener especial cuidado en entender el modo de funcionamiento de cada profiler para así establecer si los resultados obtenidos por cada uno son comparables.

[1] Con la variable de ambiente `OMP_NUM_THREADS` también se puede especificar la cantidad de hilos OpenMP.

Dentro de las ventajas del profiler *gprof*, se encuentra que la lógica necesaria para la realización del trabajo es embebida en tiempo de compilación, por lo que el trabajo es realizado casi sin provocar overhead más allá de la intención original por la cual el código es agregado. Aun así, en proyectos de gran complejidad de dependencias de compilación como el WRF, es necesario conocer muy bien los archivos relevantes que indican las flags con que se compilará el programa, dado que ésta herramienta necesita que todos los módulos sean compilados y linkeados con flags especiales; si por algún motivo algún módulo no es configurado con la flag correcta, esto tendrá un impacto directo en no obtener las métricas para dicho módulo, distorsionando en menor o mayor medida a otro módulo y por lo tanto a los resultados generales. Para obtener la información relevante para la creación del call-tree, el *gprof* inserta al inicio de cada función de los módulos compilados con la flag de profiling, un código que cuenta determinísticamente la cantidad de llamadas a ésta y a la función padre que realizó la llamada. Ésto se hace mediante un análisis de la pila de llamadas (dado que allí se encuentra la dirección de retorno de la función) y analizando el program-counter en el segmento de código. Contrariamente, la medición de tiempos de una función no se realiza en forma determinística, sino estadística mediante "sampling", al inicio de la ejecución del programa. Se configura al kernel del sistema operativo para que cada cierta cantidad de milisegundos se llame a determinadas "syscalls", las cuales analizan el "program-counter" actual en la ejecución y aumentan el contador en un array que se mapea, en forma escalada, en segmentos de código del programa para así crear un histograma. De esta manera, al final de la ejecución del programa se puede saber el porcentaje de ejecución de cada función analizando el histograma creado. Claramente el hecho de que la medición se realice en forma estadística requiere entender posibles consecuencias y errores en la medición.

Respecto a la herramienta *valgrind*, se puede notar que por el enfoque que tiene en donde el proceso a analizar es ejecutado bajo un entorno creado por la herramienta, no es necesario realizar cambios en tiempo de compilación, evitando el riesgo que resulta ser la mayor desventaja de la herramienta *gprof*. Pero ésta ventaja, tiene un gran impacto en el tiempo de ejecución del profiler sobre el programa, llegado al punto en que resulta prácticamente imposible realizar una corrida completa del programa original; meta que el *gprof* logra alcanzar. Esto último es un factor de riesgo para la confiabilidad de los datos obtenidos dado que a pesar que la ejecución del programa tiene un estilo iterativo, el cuál realiza ciclos de tareas para cada instante de tiempo de simulación, no se tiene la certeza de que el costo entre iteraciones sea independiente entre sí. Al igual que el *gprof*, la forma de crear el call-tree es realizado en forma determinística, pero la forma de contar el tiempo de cada función es medida en forma diferente. Lo que realiza el *valgrind* es contar la cantidad de instrucciones de CPU realizadas por cada función. Notemos que ésta forma de realizar la medición solo toma como relevante el trabajo realizado a nivel de CPU, descartando cualquier posible tiempo transcurrido en la función para tareas de transferencia de datos, acceso a disco, entre otras. En nuestro caso, el WRF tiene su cuello de botella en tiempo de procesamiento, por lo que esta forma de medición es adecuada.

Obtenidos ambos informes generados bajo distintos enfoques de profiling, más allá de sus diferentes formas de trabajo y de problemas descritos en el Anexo I, los mismos fueron coherentes entre sí, sugiriendo ya un orden de funciones a analizar para identificar si las mismas son pasibles de ser portadas a CUDA para su optimización. En la Tablas 4.3 y 4.4 se pueden ver salidas resumidas generadas por las herramientas *gprof* y *valgrind* respectivamente, para la ejecución del WRF utilizando un solo hilo de ejecución. Éstas muestran las 25 subrutinas de mayor porcentaje de tiempo de ejecución para cada herramienta. El profiling realizado con la herramienta *gprof* corresponde a una ejecución completa del

WRF, generando una predicción climática de 12 horas en el "Parque Emanuele Cambialrgiu", mientras que el profiling realizado con la herramienta *valgrind* corresponde a una ejecución parcial del mismo, generando una predicción de aproximadamente 35 minutos en el "Parque Emanuele Cambilargiu", debido al extenso tiempo de ejecución del profiler sobre el programa.

% time	cumulative seconds	self seconds	calls	self Ks/call	total Ks/call	name
15.53	1343.49	1343.49	10682880	0.00	0.00	sintb_
12.04	2385.01	1041.52	57600	0.00	0.00	_module_advect_em_MOD_advect_scalar_pd
9.01	3164.41	779.40	172800	0.00	0.00	_module_advect_em_MOD_advect_scalar
6.61	3735.90	571.49	720480	0.00	0.00	_module_mp_wsm3_MOD_wsm32d
3.31	4022.44	286.54	1440960	0.00	0.00	_module_mp_wsm3_MOD_nislfv_rain_plm
3.31	4308.78	286.34	134400	0.00	0.00	_module_small_step_em_MOD_advance_uv
3.30	4594.43	285.65	758880	0.00	0.00	_module_bl_ysu_MOD_ysu2d
2.92	4847.22	252.79	134400	0.00	0.00	_module_small_step_em_MOD_advance_mu_t
2.90	5097.77	250.55	134400	0.00	0.00	_module_small_step_em_MOD_advance_w
2.17	5285.31	187.54	192000	0.00	0.00	_module_small_step_em_MOD_calc_p_rho
1.86	5446.18	160.87	115200	0.00	0.00	_module_big_step_utilities_em_MOD_horizontal_diff...
1.80	5602.23	156.05	57600	0.00	0.00	_module_big_step_utilities_em_MOD_calc_cq
1.74	5752.57	150.34	57600	0.00	0.00	_module_advect_em_MOD_advect_w
1.72	5901.67	149.11	2220720	0.00	0.00	_module_bc_MOD_set_physical_bc3d
1.72	6050.29	148.62	57600	0.00	0.00	_module_advect_em_MOD_advect_v
1.71	6198.20	147.91	57600	0.00	0.00	_module_advect_em_MOD_advect_u
1.54	6331.69	133.49	1478400	0.00	0.00	_module_big_step_utilities_em_MOD_zero_tend
1.35	6448.45	116.76	134400	0.00	0.00	_module_small_step_em_MOD_sumflux
1.27	6558.68	110.23	19200	0.00	0.00	_module_big_step_utilities_em_MOD_phy_prep
1.24	6665.60	106.92	57600	0.00	0.00	_module_small_step_em_MOD_small_step_prep
1.23	6772.40	106.80	57600	0.00	0.00	_module_em_MOD_rk_addtend_dry
1.21	6876.98	104.58	57600	0.00	0.00	_module_big_step_utilities_em_MOD_curvature
1.18	6979.25	102.27	57600	0.00	0.00	_module_big_step_utilities_em_MOD_rhs_ph
1.12	7274.16	96.79	57600	0.00	0.00	_module_big_step_utilities_em_MOD_coriolis
1.04	7458.69	90.34	57600	0.00	0.00	_module_big_step_utilities_em_MOD_horizontal_press...

Tabla 4.3: Informe profiling *gprof*.

Incl.	Self	Called	Function	Location
21.03	20.43	539280	sintb_	wrf.exe
7.32	7.32	623137125	_ieee754_powf	libm-2.14.so
6.50	6.50	942846902	_ieee754_expf	libm-2.14.so
4.64	4.64	2895	_module_advect_em_MOD_advect_scalar_pd	wrf.exe
4.08	4.08	8692	_module_advect_em_MOD_advect_scalar	wrf.exe
12.06	3.23	36230	_module_mp_wsm3_MOD_wsm32d	wrf.exe
3.11	3.11	6758	_module_small_step_em_MOD_advance_uv	wrf.exe
3.07	3.07	6758	_module_small_step_em_MOD_advance_mu_t	wrf.exe
4.50	2.87	38199	_module_bl_ysu_MOD_ysu2d	wrf.exe
2.83	2.83	6758	_module_small_step_em_MOD_advance_w	wrf.exe
3.10	2.60	72460	_module_mp_wsm3_MOD_nislfv_rain_plm	wrf.exe
2.24	2.24	770598609	_ieee754_logf	libm-2.14.so
2.15	2.15	5796	_module_big_step_utilities_em_MOD_horizontal_diffusion	wrf.exe
1.74	1.74	2897	_module_advect_em_MOD_advect_w	wrf.exe
1.71	1.71	2897	_module_advect_em_MOD_advect_v	wrf.exe
1.68	1.68	2897	_module_advect_em_MOD_advect_u	wrf.exe
1.54	1.54	9655	_module_small_step_em_MOD_calc_p_rho	wrf.exe
1.52	1.52	111903	_module_bc_MOD_set_physical_bc3d	wrf.exe
1.49	1.49	2897	_module_big_step_utilities_em_MOD_curvature	wrf.exe
1.43	1.43	2897	_module_big_step_utilities_em_MOD_calc_cq	wrf.exe
1.32	1.32	2897	_module_big_step_utilities_em_MOD_coriolis	wrf.exe
1.25	1.25	2897	_module_big_step_utilities_em_MOD_rhs_ph	wrf.exe
1.00	1.00	2897	_module_big_step_utilities_em_MOD_horizontal_pressure_gradient	wrf.exe
0.83	0.83	2897	_module_small_step_em_MOD_small_step_prep	wrf.exe
0.78	0.78	6758	_module_small_step_em_MOD_sumflux	wrf.exe

Tabla 4.4: Informe profiling *valgrind*.

La siguiente etapa consistió en analizar dichos módulos, estimar el esfuerzo necesario para entender su lógica, portarlo a código C, y a partir de este portarlo a código CUDA, analizando posibles riesgos generalmente conocidos en todos los procesos de aceleración mediante el uso de técnicas de GPGPU.

Capítulo 5

Aceleración del WRF

5.1. Introducción

En el presente capítulo se exponen las rutinas del WRF abordadas con el fin de acelerar el desempeño computacional del mismo utilizando GPUs. Dentro de estas rutinas se encuentran tanto módulos portados en este trabajo, como módulos portados por terceros que se lograron integrar a la propuesta, además de un análisis detallado de rutinas que no se portaron y los justificativos para estas decisiones.

5.2. `sintb`

Del profiling realizado se observa que la función `sintb` es la que consume mayor proporción del tiempo total de ejecución, lo cual motivó la búsqueda de mejoras en esta rutina para reducir el tiempo total de ejecución del modelo. A continuación, se presenta el análisis de la viabilidad de portar la rutina y las distintas versiones implementadas.

5.2.1. Análisis

Para estudiar la viabilidad de acelerar esta función haciendo uso de la GPU, se analizó el tamaño de los parámetros recibidos y el tiempo que representan sus transferencias a la memoria de la GPU, en relación al tiempo consumido por la ejecución de una sola llamada, resultando en una relación de cuatro a uno entre el tiempo de transferencias y el tiempo de ejecución. Llegado a este punto se analizó también la función `bdy_interp` que es quién invoca a `sintb`, buscando ocultar los tiempos de las transferencias de datos a GPU, intentando realizar las transferencias de forma asincrónica mientras se realizan otros cálculos en CPU que no afecten a los datos transferidos, o dejando las entradas y resultados recientemente utilizados en GPU (si seguido a una ejecución del kernel lo sigue otra que utilice esos mismos datos, ahorrando así las transferencias para esta última).

Se constató que `bdy_interp` realiza cuatro llamadas seguidas a `sintb` con las mismas matrices de entrada, pero con un conjunto de coordenadas sobre las que operar diferente. Seguido a estas invocaciones se encuentran asignaciones a otras matrices condicionadas por los datos obtenidos. Tanto las llamadas como las asignaciones se encuentran en un loop de aproximadamente cincuenta iteraciones, paralelizado mediante OpenMP. Se tiene entonces que las transferencias demoran cuatro veces más que una ejecución del `sintb`, pero a su vez, se necesita hacer una única transferencia para cuatro llamadas que se realizan consecutivamente, obteniendo así la posibilidad de obtener un portado exitoso.

5.2.2. Versión 1

Para esta versión se portó en una primera instancia el código de Fortran a C, y luego de comparar los resultados con la versión original, se portó el código de `sintb` a CUDA de forma natural, convirtiendo la cantidad de iteraciones en hilos y bloques. Se intentó realizar accesos a memoria de forma coalesced, uso de memoria compartida y se evitó el uso de mallocs y frees en cada llamada ya que el tamaño de las matrices utilizadas no es variable a lo largo de la ejecución del WRF, realizando todos los mallocs necesarios al principio del programa y luego utilizando dichas reservas repetidas veces, ahorrando así tiempos, que son mayores incluso a los tiempos de transferencia en algunos casos. No se ocultaron transferencias de memoria en esta versión. Luego de completarla se evaluó la performance obtenida, la cuál se encontraba lejos de alcanzar el tiempo original de la rutina multihilo (5.8×10^{-2} vs. 1.15×10^{-2} segundos).

5.2.3. Versión 2

En la segunda versión del portado, se intentó ocultar las transferencias que se utilizarían en la siguiente iteración de `bdy_interp`, mientras se realizan las asignaciones que se encuentran seguidas a las 4 llamadas ya mencionadas. Una vez cumplidos estos pasos la performance no tuvo mejoras significativas, por lo que se procedió a realizar un análisis de la ejecución en GPU, encontrando que el kernel se invocaba con un máximo de 256 hilos determinados por la estructura del problema. Por esta razón no se hacía un correcto aprovechamiento de los cores de la GPU[1], además del impedimento de la misma de ocultar los accesos a memoria global, ya que no existían suficientes hilos para ejecutar mientras otros estaban bloqueados en espera por una lectura de memoria.

5.2.4. Versión 3

En esta versión se procedió a buscar la forma de realizar más cálculos en una sola invocación al kernel para evitar los problemas descritos en versiones anteriores. Se constató que las coordenadas de las invocaciones a `sintb` no dependían del iterador, o sea que a lo largo de las cincuenta iteraciones de `bdy_interp` se mantenían los rangos de coordenadas sobre las que operaban cada una de las cuatro llamadas a `sintb`, la variante era la matriz sobre la que se operaba. Por esto, se decidió intercambiar las doscientas llamadas a `sintb` (50 iteraciones $\times$ 4 llamadas c/u) por una única llamada, dejando todas las asignaciones pendientes para el final. Con esto se perdió la copia asincrónica de las matrices de entrada, ya que se debió hacer las cincuenta transferencias en un principio. Contrarrestando, se multiplicó la cantidad de hilos en cincuenta, pudiéndose así explotar todo el potencial de la GPU.

De estos cambios se obtuvo un resultado exitoso, mejorando la performance de `bdy_interp` en un 40 % aproximadamente con respecto a la versión original multihilo. Los tiempos de esta versión son aproximadamente 8.2×10^{-3} segundos.

[1] En el caso de la GPU utilizada, GTX 480, 480 cores.

5.2.5. Versión 4

Las nuevas limitantes para alcanzar un mayor speedup son a las transferencias de las matrices resultados que ocupan un 25 % del tiempo total, y el código de asignaciones que anteriormente seguía a las cuatro invocaciones al `sintb`, que continuaba paralelizado mediante multi-threads (con OpenMP) en CPU, pero que no podía ejecutarse concurrentemente con el kernel dado que precisaba las matrices resultantes de este, y significaba más de un 30 % del tiempo total.

Se consideró dividir el kernel en dos partes, manteniendo las ventajas obtenidas en la tercera versión, pero permitiendo ocultar la copia de la primera mitad de los resultados, solapando esta copia con la segunda parte del kernel. Sin embargo, esta opción no mostró beneficios (en experimentos no formalizados) debido a que las dimensiones del caso de prueba utilizado implican mayor penalización al partir el problema (la GPU es subutilizada) que las ganancias obtenidas en este tipo de estrategias. Entonces se optó por migrar a CUDA el código que restaba en `bdy_interp` en un segundo kernel, no sólo buscando reducir el tiempo que este significaba, sino también aprovechando que la entrada que este necesitaba ya estaba en GPU. De esta manera, no es necesaria ninguna transferencia, con excepción de matrices donde actualizar los resultados finales, las cuáles se lograron copiar asincrónicamente a la GPU mientras se ejecuta el primer kernel, ocultando así las transferencias. Además, la dimensión de los resultados finales se vio reducido a la mitad de su tamaño original, disminuyendo significativamente el tiempo de transferencia de los mismos al host.

Para esta última versión se realizó también el análisis de registros y memoria compartida por bloque, buscando la mejor relación entre el uso de memoria compartida y registros que agilice los accesos a memoria, y la flexibilidad que le brinda al scheduler tener registros y memoria compartida libre en una SM para poder así correr distintos bloques y ocultar mejor los accesos a la memoria.

Para la GPU utilizada, un bloque de este kernel utilizaba un cuarto de los registros disponibles en una SM, permitiendo así un máximo de cuatro bloques concurrentes por SM, por lo que se hicieron pruebas experimentales sobre la performance del kernel, para los casos en que un bloque utilice aproximadamente 48, 24 o 12 KB de memoria compartida, restringiendo a uno, dos o cuatro bloques concurrentes por SM respectivamente. La mejor configuración resultó en la utilización de 24KB de memoria compartida por bloque.

También se identificó una carga de arrays con valores constantes, por lo que se decidió cargar el array en el kernel y guardarlo en la memoria compartida; este cálculo no se realizó en CPU para evitar transferencias. Además se realizaron otras mejoras que no aplican únicamente al contexto de las GPUs, como remover matrices auxiliares que provocaban tanto una escritura y lectura innecesaria de memoria, ya que existían cálculos que se asignaban a matrices, con un comportamiento equivalente al de una variable temporal. Cabe destacar que se analizó la posibilidad de utilizar banderas de compilación como `use_fast_math`, la cual mejora el tiempo de procesamiento en el kernel sacrificando precisión en los cálculos. Finalmente no se utilizó dado que la mayor parte del tiempo de ejecución se encuentra en accesos a memoria, por lo que la ganancia sería baja y no valdría la pena la incorporación de un nuevo error numérico.

5.2.6. Evaluación

En la Figuras 5.1 y 5.2 se presentan gráficos mostrando la evolución de los tiempos de ejecución y del speedup de las distintas versiones de `bdy_interp` junto a `sintb`.

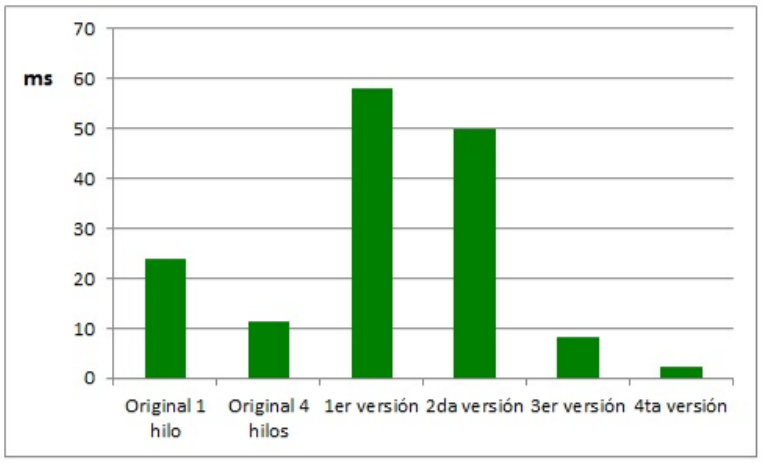

Figura 5.1: Evolución del tiempo de ejecución en las versiones de `bdy_interp` y `sintb` portadas a CUDA con respecto a su versión original multihilo.

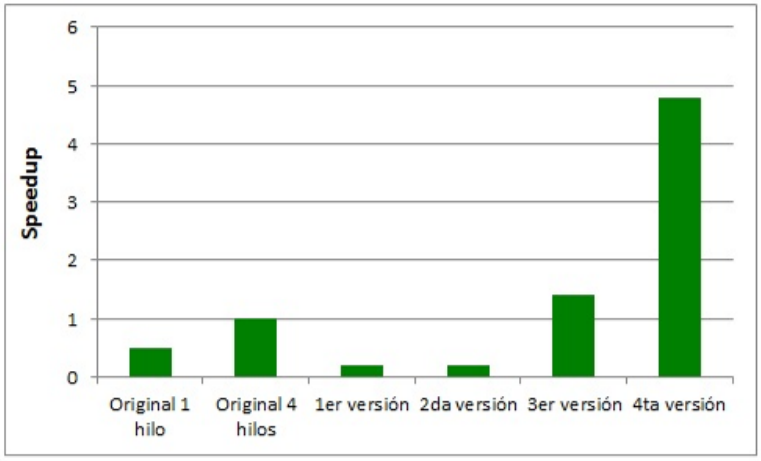

Figura 5.2: Evolución del speedup en las versiones de `bdy_interp` y `sintb` portadas a CUDA con respecto a su versión original multihilo.

Finalizado el proceso de optimización, se constató una mejora de performance del `bdy_interp` junto a `sintb` de 4.8× aproximadamente con respecto a la versión original multihilo, llegando a demorar 2.35 $\times 10^{-3}$ segundos, que se corresponden a una reducción del tiempo de ejecución total del WRF en un 10 % aproximadamente, es decir alrededor de 11 minutos en el caso de prueba estudiado. Del tiempo de la versión final, las transferencias se adjudican el 19 %, mientras que la ejecución 81 %, 4.4 $\times 10^{-4}$ y 1.9 $\times 10^{-3}$ segundos respectivamente. Todas las pruebas de performance realizadas en estas subrutinas fueron ejecutadas en la Plataforma 1.

5.3. slope_wsm3

Al buscar funciones para portar, se analizaron también funciones que a pesar de no aparecer como protagonistas en los informes de profiling, podrían consumir un tiempo considerable por hacer uso de las funciones intrínsecas de Fortran (`pow`, `exp` y `log`), las cuáles no son medidas por el profiler *gprof* ya que no fueron compiladas con las banderas necesarias pero si por el profiler *valgrind*, aunque como funciones individuales y no como tiempo propio de quien las invoca (Ver Anexo I). Dentro de las funciones que cumplen con estas características se destaca `slope_wsm3`, la cual es candidata no sólo por su tiempo de ejecución sino por su extensión de código, que la hace fácil de abordar. Para ésta se constató que demora un 3.6 % del tiempo total de ejecución del WRF ejecutado con solamente un hilo. Esto la posiciona entre las 5 funciones con mayor tiempo de ejecución propio[2] del WRF. Cabe aclarar que la misma fue medida con un sólo hilo dado que es llamada dentro de la función `wsm32D`, y la misma se llama con todos los hilos OpenMP, por lo que no se puede medir fácilmente el tiempo real de `slope_wsm3`, ya que la suma del tiempo de cada hilo en llamarla no es necesariamente el tiempo real de ejecución de la misma.

5.3.1. Análisis

La función `slope_wsm3` es llamada dos veces por `wsm32D`, la cual resulta ser la función principal del módulo WSM3. El mismo es uno de los módulos más importantes en la implementación de componentes físicos del WRF, es encargado de representar la condensación, diferentes tipos de precipitación y efectos termodinámicos relacionados. Además se puede ver en las Tablas 4.3 y 4.4 que varias funciones del WSM3 se adjudican grandes tiempos de ejecución, por lo que portar a `slope_wsm3` puede ser un punto inicial para el portado del módulo.

`slope_wsm3` es llamada por cada hilo OpenMP de forma independiente y tiene como parámetros 9 matrices bidimensionales de tamaño 37×53 reales, de las cuales 4 son de entrada y 5 de salida. Las dos invocaciones que se llevan a cabo, se realizan con las mismas matrices, sin embargo, dos de ellas sufren modificaciones entre una llamada y otra. Todo el código de la función se encuentra en una gran iteración que recorre las matrices por columnas, realiza cálculos necesarios y actualiza las matrices de salida. Dentro de las iteraciones se encuentran dos grandes condicionales anidados que dependiendo de dos condiciones que refieren a valores de las matrices de entrada, actualizan a las matrices de salida de una u otra manera, lo que resulta en 4 formas diferentes de actualizar las matrices. Esto podría suponer un gran problema para el portado a CUDA ya que podría traducirse a un gran nivel de divergencia que descartaría el portado, sin embargo, se comprobó que durante una gran cantidad de iteraciones se ejecuta bajo la misma condición, lo que hace que la divergencia de hilos sea casi nula.

5.3.2. Versión 1

Para esta versión primero se procedió a portar de Fortran a C y se compararon los resultados con la versión original. Luego se portó a CUDA de forma básica, el iterador del primer for se convirtió en la cantidad de bloques y el segundo en la cantidad de hilos. Las transferencias a memoria de GPU se realizaron inmediatamente antes e inmediatamente

[2]Sin tomar en cuenta el tiempo de ejecución de invocaciones a otras rutinas.

después de la llamada al kernel. Los tiempos de ejecución de esta versión resultaron ser de entre 1× y 2× por llamada con respecto a la versión original multihilo, el cual implica un tiempo de ejecución por llamada de aproximadamente 3 $\times 10^{-4}$ segundos. Cabe destacar que para esta versión también se realizó el código para reservar memoria para las matrices en GPU al comienzo de la ejecución del WRF, para luego únicamente obtener las direcciones de memoria para realizar las transferencias cuando sean necesarias.

5.3.3. Versión 2

En esta versión se identificó que el mayor consumo de tiempo se encontraba en las transferencias de memoria (2 $\times 10^{-5}$ segundos para el kernel y entre 3 $\times 10^{-4}$ y 5 $\times 10^{-4}$ segundos para transferencias), por lo que se modificó ligeramente el código de `wsm32D` para cargar casi todas las matrices de entrada de `slope_wsm3` antes de la primer llamada, y poder así enviarlas a GPU al empezar a ejecutar `wsm32D` de forma asincrónica. A partir de ello, se notó que el kernel en una amplia mayoría de sus invocaciones comenzaba a ejecutar sin tener que esperar por las transferencias. Los tiempos de ejecución presentaron una mejora de entre 2.5× y 3× aproximadamente por llamada con respecto a la versión original multihilo. Es importante aclarar que existen algunas transferencias de matrices en la primer llamada a `slope_wsm3`, y otras en la segunda, que no se pudieron ocultar debido a la forma de uso de los datos.

5.3.4. Versión 3

Una inspección detallada del código mostró que a pesar de que la función presenta 5 matrices de salida, entre la primera y segunda invocación a la misma solamente se utiliza 1, por lo que enviar las 4 restantes a la memoria de la CPU carecía de sentido. Por este motivo se modificó el código para traer las 5 matrices solo en caso de estar bajo la segunda invocación, y solamente una en caso contrario. Con esto los tiempos de ejecución presentaron una mejora de entre 6.5× y 2.5× aproximadamente con respecto a la versión original multihilo, correspondientes a mejoras con respecto a la primer y segunda invocación respectivamente.

5.3.5. Versión 4

Para esta última versión se procedió a realizar mejoras adicionales en el kernel. Para el tamaño de matrices utilizadas, se identificó que entraban solamente 6 en memoria compartida (menos de 8KB por matriz). Por lo tanto se configuró el kernel para que cargue y utilice las 3 matrices más accedidas en la misma, con el fin de que se puedan ejecutar 2 bloques simultáneamente por multiprocesador. Sin embargo, luego de realizar este cambio los tiempos no mejoraron debido a que estas matrices se utilizan muy poco, y llevarlas y traerlas a memoria compartida degradaba los tiempos de ejecución, por lo que solamente se llevó a memoria compartida la matriz más utilizada, lo que produjo una ligera mejora en los tiempos de ejecución. Cabe destacar que la función original recorre las matrices por columnas, y las mismas se encuentran almacenadas en memoria por columnas, por lo que al portar el código, hilos adyacentes accedieron a posiciones

adyacentes de memoria sin realizar cambios adicionales, es decir que la gran mayoría de los accesos se realizan de forma coalesced.

En esta versión también se realizaron otras mejoras adicionales como la remoción de matrices auxiliares que provocaban una escritura y lectura innecesaria en memoria, ya que existían cálculos que se asignaban a matrices, con un comportamiento equivalente al de una variable temporal. Además se analizó la inclusión de banderas de optimización de cálculos en la compilación, al igual que en `sintb`, se decidió no utilizarlas ya que el mayor tiempo de ejecución se encuentra en accesos a memoria, por lo que la ganancia sería baja y se incorporarían errores de precisión.

5.3.6. Evaluación

En las Figuras 5.3 y 5.4 se presentan gráficos mostrando los tiempos de ejecución y la evolución del speedup de la primer invocación a `slope_wsm3` en sus diferentes versiones. En las Figuras 5.5 y 5.6, se muestra la misma información para la segunda invocación.

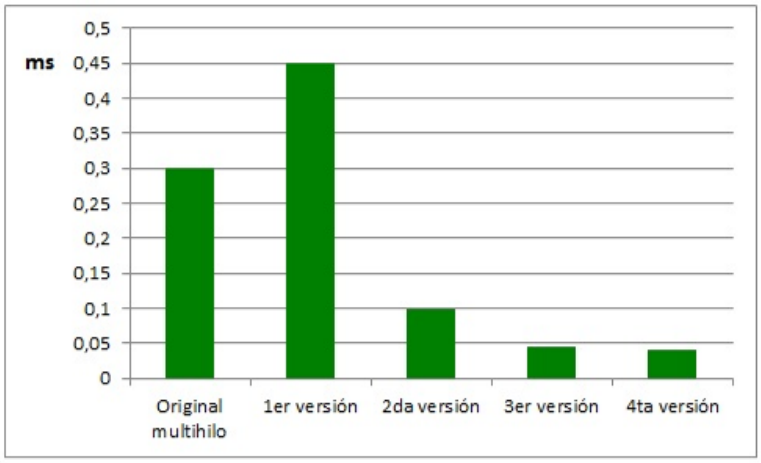

Figura 5.3: Evolución del tiempo de ejecución de la primer invocación a `slope_wsm3` en sus diferentes versiones portadas a CUDA con respecto a su versión original multihilo.

Finalizado el proceso de optimización, las mejoras resultaron ser de entre 3× y 7× aproximadamente más rápidos que el tiempo de la versión original multihilo. Los tiempos finales de ejecución de la subrutina portada a CUDA fueron 4 $\times 10^{-5}$ segundos aproximadamente para la primera invocación y 1.1 $\times 10^{-4}$ segundos aproximadamente para la segunda, que se corresponden a una reducción del tiempo de ejecución total del WRF de un 3 % aproximadamente, es decir alrededor de 3 minutos. Del tiempo de la versión final de la primer invocación a `slope_wsm3`, las transferencias se adjudican un 55 %, mientras que la ejecución un 45 %, 2.2 $\times 10^{-5}$ y 1.8 $\times 10^{-5}$ respectivamente. Para la segunda invocación, las transferencias involucran el 83 %, y la ejecución el 17 %, 9.2 $\times 10^{-5}$ y 1.8 $\times 10^{-5}$ segundos respectivamente. Todas las pruebas de performance realizadas en esta subrutina fueron ejecutadas en la Plataforma 1. Notar que en un contexto donde la ejecución del WRF sea mayormente en GPU, las aceleraciones alcanzadas (sin computar tiempos de transferencias) serían cercanas a 17× para ambas invocaciones.

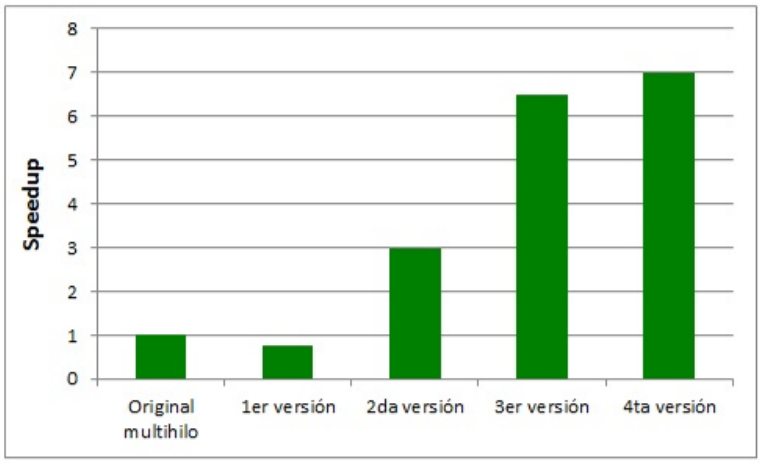

Figura 5.4: Evolución del speedup de la primer invocación a `slope_wsm3` en sus diferentes versiones portadas a CUDA con respecto a su versión original multihilo.

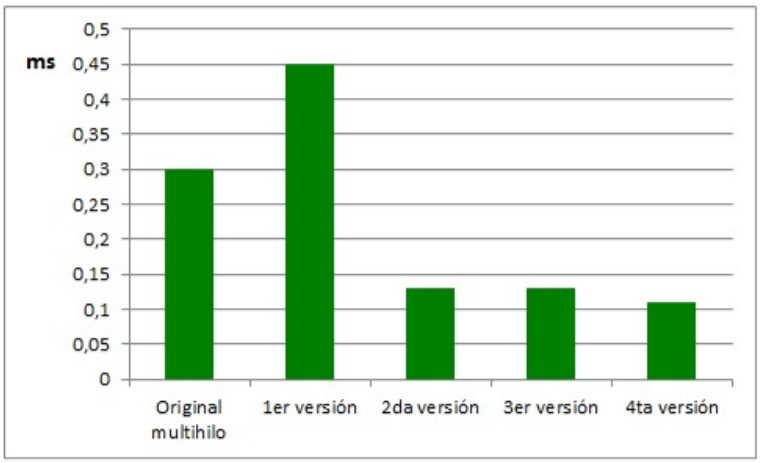

Figura 5.5: Evolución del tiempo de ejecución de la segunda invocación a `slope_wsm3` en sus diferentes versiones portadas a CUDA con respecto a su versión original multihilo.

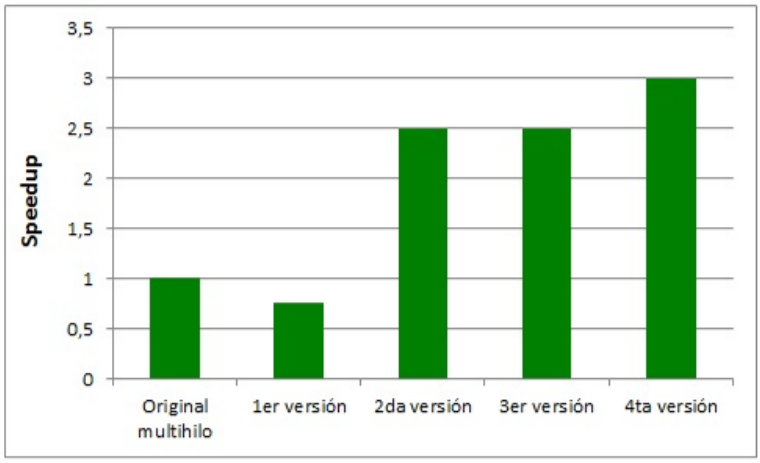

Figura 5.6: Evolución del speedup de la segunda invocación a `slope_wsm3` en sus diferentes versiones portadas a CUDA con respecto a su versión original multihilo.

5.4. Integración de trabajos previos disponibles públicamente

Como se presentó en la Sección 3.4, se han desarrollado diversos esfuerzos tendientes a utilizar GPUs para acelerar el cómputo del WRF. En este apartado se describe el trabajo realizado para incorporar algunos de los mismos a la ejecución de forma operativa del WRF.

5.4.1. WSM5

El módulo de microfísica WSM5, como ya se mencionó en el trabajo [20] fue portado a CUDA C. Dicho módulo fue elegido para ser portado debido al tiempo que insume y a la independencia entre cálculos que presenta, es decir, reune condiciones favorables para ejecutarse en GPU. El código fuente del mismo es libre y se puede obtener de la web [19], sin embargo, dada la configuración del modelo del WRF utilizado hasta este momento en la herramienta desarrollada por el IMFIA, los cálculos relativos a la microfísica de este módulo se realizaban con un modelo simplificado del WSM5, el WSM3. El WSM3 realiza un trabajo similar al WSM5, pero representando menos variables climáticas. Por esto se procedió a alterar la configuración del modelo corriente, para que utilice el módulo WSM5 en lugar del WSM3, teniendo presente que la precisión se vería mejorada o en su defecto se mantendría incambiada, sacrificando potencialmente tiempo de cálculo. Cabe destacar que el WSM3 con la configuración original conllevaba uno de los tiempos de ejecución más altos del modelo.

Luego de corroborar la similitud de las predicciones obtenidas con ambos modelos, se procedió a integrar el modelo original con el código fuente del trabajo [20]. Esto se llevó a cabo sin mayores inconvenientes, ya que existía documentación sobre su integración al WRF, e incluso código dentro del WRF, pensado para llevar a cabo este propósito.

Después de ratificar los resultados, se hizo el estudio de tiempos de ejecución. El módulo WSM3 en su conjunto ocupaba un tiempo próximo al 12 % del total de la ejecución multihilo del WRF utilizando el WSM3, es decir unos 13 minutos. Por su parte, el WSM5 demoraba aproximadamente 34 minutos en la ejecución multihilo del WRF utilizando el WSM5, alrededor de 28 % del total, mientras que cada llamada al mismo significaba unos 6×10^{-2} segundos. Luego de medir los tiempos de las llamadas al módulo portado a GPU, se notó que cada una conllevaba un tiempo cercano a 9×10^{-3} segundos, resultando en una mejora de aproximadamente 6.7× respecto al tiempo de la versión original del WRF utilizando el WSM5; y de 29 minutos menos en la ejecución total del modelo en la Plataforma 1 (aproximadamente un 23 %). Sin embargo, estos resultados se muestran por debajo de los presentados en el artículo original, donde se aseguraba que se obtenía 17× utilizando una GTX 8800 en comparación con un Pentium-D 2.8 GHz. Esta diferencia se debe posiblemente a que la comparación realizada por los autores es contra el WRF secuencial, donde se aprovecha solamente un core del Pentium-D, o al juego de datos diferente con el que fue probado.

En la Tabla 5.1 se pueden apreciar los tiempos de ejecución del WRF multihilo utilizando el WSM5 y el WSM3, y en la Tabla 5.2, los tiempos de ejecución de los módulos WSM5 y WSM3 multihilo. De los resultados obtenidos se puede apreciar que a pesar de que el WSM5 realiza un cálculo mucho más intenso que el WSM3, utilizando la versión portada a GPU los tiempos resultaron ser menores.

Descripción de ejecución	Tiempo
WRF multihilo utilizando el WSM3 en CPU	1:43:23 hs
WRF multihilo utilizando el WSM5 en CPU	2:02:22 hs
WRF multihilo utilizando el WSM5 en GPU	1:34:17 hs

Tabla 5.1: Tiempos de ejecución del WRF utilizando el WSM5 y WSM3 en sus diferentes versiones.

Descripción de ejecución	Tiempo
Módulo WSM3 multihilo en CPU	13:14 mins
Módulo WSM5 multihilo en CPU	34:25 mins
Módulo WSM5 multihilo en GPU	5:33 mins

Tabla 5.2: Tiempos de ejecución de los módulos WSM5 y WSM3 en sus diferentes versiones.

5.4.2. Advect

Dentro de las funciones que conllevan mayor tiempo de cálculo del modelo se encuentran las pertenecientes al modulo de Advección. Resultado de la revisión de trabajos realizados sobre el WRF se encuentra una implementación en GPU del módulo Advect [21]. Al igual que con el trabajo descrito anteriormente, su código fuente es libre y se puede obtener en su sitio web. Sin embargo, a diferencia del código anterior, para este no se menciona en el trabajo que haya sido realizado para una integración completa al WRF, sino que es presentado como una prueba de concepto para exponer el speedup obtenido para cierto juego de datos.

La naturaleza del código de este módulo hace que portar su funcionamiento a GPU se torne complejo si se compara con otros módulos como el WSM5. Dentro de éstos problemas se encuentran: baja tasa de operaciones de punto flotante respecto a accesos a memoria, dependencia de datos entre puntos de la grilla y gran transferencia de datos entre CPU y GPU. Los autores mencionan que el speedup obtenido con el juego de datos utilizado, al comparar la versión en GPU respecto a la versión original del WRF, es del entorno de 6.7× utilizando una GPU Tesla C1060 y un hilo de ejecución en un CPU Intel Xeon E5440.

Dada la complejidad del módulo portado y su aceptable speedup obtenido, se decidió realizar máximos esfuerzos para alcanzar una integración del código presentado en el trabajo [21] al WRF, a pesar de haber sido portado parcialmente. Para esto, no se dispuso de documentación que describiese los pasos a realizar ni que definiera el contexto para el cual el portado fuese válido, aún así se realizó un trabajo de investigación para intentar lograr la integración.

La primer dificultad encontrada fue la necesidad de realizar ajustes de compilación para obtener solamente el código independiente del módulo portado a CUDA, ya que el código fuente del trabajo tiene una fuerte conexión con macros de compilación utilizadas para compilar el módulo para que ejecute de forma separada al WRF.

El siguiente paso fue analizar el código que efectivamente llama al código en CUDA, la función `rk_scalar_tend`. Originalmente ésta función fue pensada como driver para una prueba standalone, por lo que su firma no es exactamente la misma que la firma

perteneciente al código original en el WRF. Esto requirió un análisis detallado de cuales son los parámetros faltantes y su participación en la función. Resultado de éste análisis, se concluyó que el módulo portado sólo correspondía a un posible flujo de ejecución de la función. Ésto requirió un nuevo análisis en tiempo de ejecución que permitiese confirmar que los parámetros omitidos no fueran necesarios en este flujo. En dicho análisis, se relevaron los valores de los parámetros faltantes en el flujo de ejecución asumido en el código portado. Efectivamente, se constató que dichos parámetros siempre tenían valores constantes por lo que parecía razonable que fuesen omitidos; de hecho un subconjunto de los mismos eran de salida pero aún así en el flujo nunca eran modificados. Se constató que un tercio de las llamadas a la función `rk_scalar_tend` seguían el flujo mencionado.

Luego de analizado el código portado y delimitada su participación en el contexto de la función original, se procedió a realizar una compilación de dicho código que resultó en múltiples errores. Ejemplos de éstos errores fueron: variables no definidas, ajustes de macros de compilación influyentes en la generación final de la función, entre otros.

Para la integración final se procedió a mantener la función `rk_scalar_tend` original, y se realizó la llamada a la función portada en caso de que se cumplieran los valores requeridos para que ocurra el flujo de ejecución portado. Un detalle no menor es que el código en CUDA no estaba realizado para ser utilizado en un entorno multihilo, y como la configuración utilizada del WRF lo es, no era correcto omitir esto ya que podrían darse problemas de concurrencia. Para solucionar este problema, se decidió retirar las macros de OpenMP para este flujo de ejecución, pero se mantuvieron para los restantes en pos de causar el menor impacto posible en el rendimiento del resto de la ejecución del modelo. Al realizar esto, también se evitó una partición de datos que resultaba en múltiples llamadas a la función portada en dominios muy pequeños. De este modo es posible realizar una única llamada en un dominio más grande, logrando paralelizar más trabajo y mejorar el speedup. Otro cambio que se realizó fue modificar el pedido y liberación de memoria en GPU para que se haga una única vez, ya que estaban siendo realizados cada vez que se llamaba a la función, resultando en un overhead innecesario.

A pesar del esfuerzo realizado, no se logró un speedup comparable al obtenido por los autores del artículo. El speedup obtenido del flujo de ejecución optimizado fue aproximadamente 1.15× con respecto a su versión original multihilo (2.8×10^{-3} vs. 3.2×10^{-3} segundos), cuando los resultados presentados eran de 6.7×. Al igual que con el código portado del WSM5, esto se debe a que los autores se compararon contra la ejecución del WRF en forma secuencial o al juego de datos diferente con el que fue probado. De todas formas se considera importante haber realizado dicha integración intentando que otros trabajos realizados en el área puedan aportar al objetivo planteado en este proyecto. Cabe destacar que el flujo de ejecución portado a GPU se da una de cada tres veces que se invoca `rk_scalar_tend` en la ejecución del WRF, lo que hace aún menor el speedup total de la rutina.

5.5. Propuestas de trabajo solapando CPU y GPU

5.5.1. Análisis

Al relevar datos sobre la forma de ejecución del WRF, se constataron particularidades en el cálculo de la radiación. El mismo es realizado por el módulo `module_ra_rrtm` y tiene dos aspectos a tener en cuenta. El primero, es que modelo físico utilizar para simular la magnitud, y el otro es la frecuencia de cálculo. Un aumento de la frecuencia de cálculo

afecta en forma directa en la performance de la ejecución, por lo que su correcto ajuste es necesario para obtener el mejor costo/beneficio.

Al ser un cálculo realizado cada un período fijo de tiempo, su resultado no es estrictamente necesario en forma inmediata dentro del flujo de ejecución del programa, por lo que es posible realizar dicho trabajo en forma asíncrona sin afectar tiempos de espera para el hilo principal de ejecución. Cambiar a esta forma de computar el resultado permite tener más flexibilidad en ajustes de la configuración respecto a la frecuencia de cálculo ya que se paraleliza trabajo, y más aún si se involucra al recurso de GPU para el cómputo para así liberar de carga a la CPU.

Se realizó un profiling del módulo `module_ra_rrtm` para la configuración existente en el WRF al momento del análisis inicial, descubriendo que los cálculos involucrados en el mismo no ocupaban un porcentaje significativo del tiempo de cómputo total de la ejecución completa. Se relevó información acerca de otras posibles configuraciones para el módulo, y se cambió el modelo para que ejecutara al `module_ra_rrtm` de forma óptima según la documentación del WRF [47], aumentando la frecuencia de ejecución y manteniendo el modelo de simulación de radiación. Se realizó nuevamente un profiling, y en este caso el cálculo de radiación si llegó a ser significativo (4.8 %). En dicha configuración la función `radiation_driver` es la encargada del cálculo de radiación, invocando cada cierto período de tiempo a la función `rrtmlwrad`. Cabe destacar que el modelo físico utilizado actualmente para la simulación de radiación, RRTM Long Wave Radiation, no está entre los modelos más pesados computacionalmente, existen otros (p.e, Goddard Radiation, CAM Radiation, etc.) que implican incrementos importantes en el tiempo de ejecución.

5.5.2. Implementación asíncrona de `rrtmlwrad`

La implementación del cómputo asíncrono que se realizó se basó en transformar la función `rrtmlwrad` para que en cada llamada devuelva el último resultado disponible según los parámetros de entrada. Este deberá coincidir con la llamada anterior siempre y cuando la frecuencia configurada sea mayor que el tiempo de ejecución de la función; en el caso de la configuración establecida, el tiempo de cálculo es despreciable respecto a la frecuencia por lo que esto se cumple. Para el caso de borde de la primera ejecución, o de llamadas a la función con parámetros de entrada no esperados (para los cuales no tiene sentido la forma de cálculo), la ejecución se realiza en forma síncrona. Dado que se cambió el paradigma de cálculo de la función de síncrona a asíncrona, existen recursos de ejecución compartidos entre distintos hilos como variables de salida, por lo que fue necesario utilizar herramientas que coordinaran su acceso. Esto se hizo mediante el uso de semáforos y su API correspondiente nativa en C, ya que Fortan no soporta en su forma nativa la creación de semáforos.

La evaluación del impacto del cambio en los resultados finales de la ejecución completa del modelo respecto a la versión síncrona, permitió comprobar que la mayoría de las diferencias encontradas no eran significativas. Las mismas se encuentran detalladas en la Sección 5.7.1.

Cabe destacar que éste cambio de paradigma es independiente de la configuración de radiación en el modelo, ya que se basa en la idea de cambiar la ejecución de una función llamada en una frecuencia de tiempo establecida configurable, cuyo resultado no es requerido en forma inmediata, por lo que puede ser extendido a cualquier otra configuración de modelo de radiación elegida. Obviamente, cuanto mayor la frecuencia de invocación del cálculo mejores resultados numéricos se obtienen.

Diseño de devolución retrasada de valores

Al calcular la radiación, la función `radiation_driver` realiza una partición del dominio en tantas partes iguales como hilos OpenMP se encuentren configurados para la CPU, y para cada una de éstas llama a `rrtmlwrad` con un hilo OpenMP, lo que implica que no es posible retornar directamente el valor de la última llamada a la función, dado que el mismo puede no corresponder al valor del último retorno para la partición que está siendo calculada. De no tener este cuidado se obtendrían valores antiguos incoherentes para cada partición. Para resolver dicho problema, internamente se considera en que número de partición se está trabajando y se obtiene el último valor calculado para cada partición, pudiendo así devolver el valor antiguo que corresponda.

Delegación de trabajo a hilos concurrentes

Dado que Fortran en su forma nativa no soporta manejo de hilos, para poder calcular un nuevo valor de `rrtmlwrad`, se resolvió que cada hilo OpenMP llame a una función escrita en C, la cual solamente creará un hilo con la utilización de la API de *pthreads* que será el encargado de ejecutar el código original de `rrtmlwrad` en Fortran mientras se devuelve el valor retrasado y se continúa con la ejecución del WRF. Esta forma de ejecución conllevó a enfrentar problemas con el manejo de recursos compartidos en la ejecución del código original de `rrtmlwrad` (Ver Anexo II). Resultado de ello, se decidió ofrecer una función dedicada para cada hilo. La sección de código de éstas funciones son idénticas, dado que todos los hilos realizan su trabajo bajo la misma lógica con distintos datos.

Dado que la cantidad de hilos surge del número de hilos OpenMP, y que éstos son un parámetro del sistema, se decidió realizar la creación e invocación de las funciones idénticas mencionadas de manera tal que sea transparente a la cantidad de hilos OpenMP. De esta forma si en un futuro llegase a cambiar dicha cantidad, se creará una función para cada hilo. Para realizar esto, la creación de las funciones a nivel de código fue resuelto con macros de preprocesador que en función de parámetros de configuración, generan las funciones. Del mismo modo se resolvió para la generación de código que decide cual función asignar a cada hilo.

Como trabajo futuro se puede considerar analizar con más profundidad las causas que derivaron a tomar ésta decisión de diseño, que a pesar de no ser errónea desde el punto de vista de correctitud de cálculo, se considera que no genera código que siga las mejores prácticas respecto a evitar redundancia de código en la implementación de software. En la Figura 5.7 se puede ver un esquema de las implementaciones sincrónica y asincrónica de `rrtmwrad`.

5.5.3. Portado a CUDA de `rtrn`

Al realizar cálculos asíncronos, si se utiliza la GPU como recurso de ejecución extra, además de paralelizar cálculos se libera carga de la CPU, lo que posibilita que esta pueda continuar con la ejecución del hilo principal. Esto permite prácticamente descartar código portado como costo de cómputo adicional para el WRF. Es importante destacar que, para llevar a cabo esta idea, no es necesario que el kernel este optimizado; es suficiente con que el kernel ejecute en un tiempo menor al tiempo de cómputo entre invocación e invocación, dado que se está utilizando un recurso ocioso, y no se ahorra tiempo alguno si el kernel culmina en un tiempo menor.

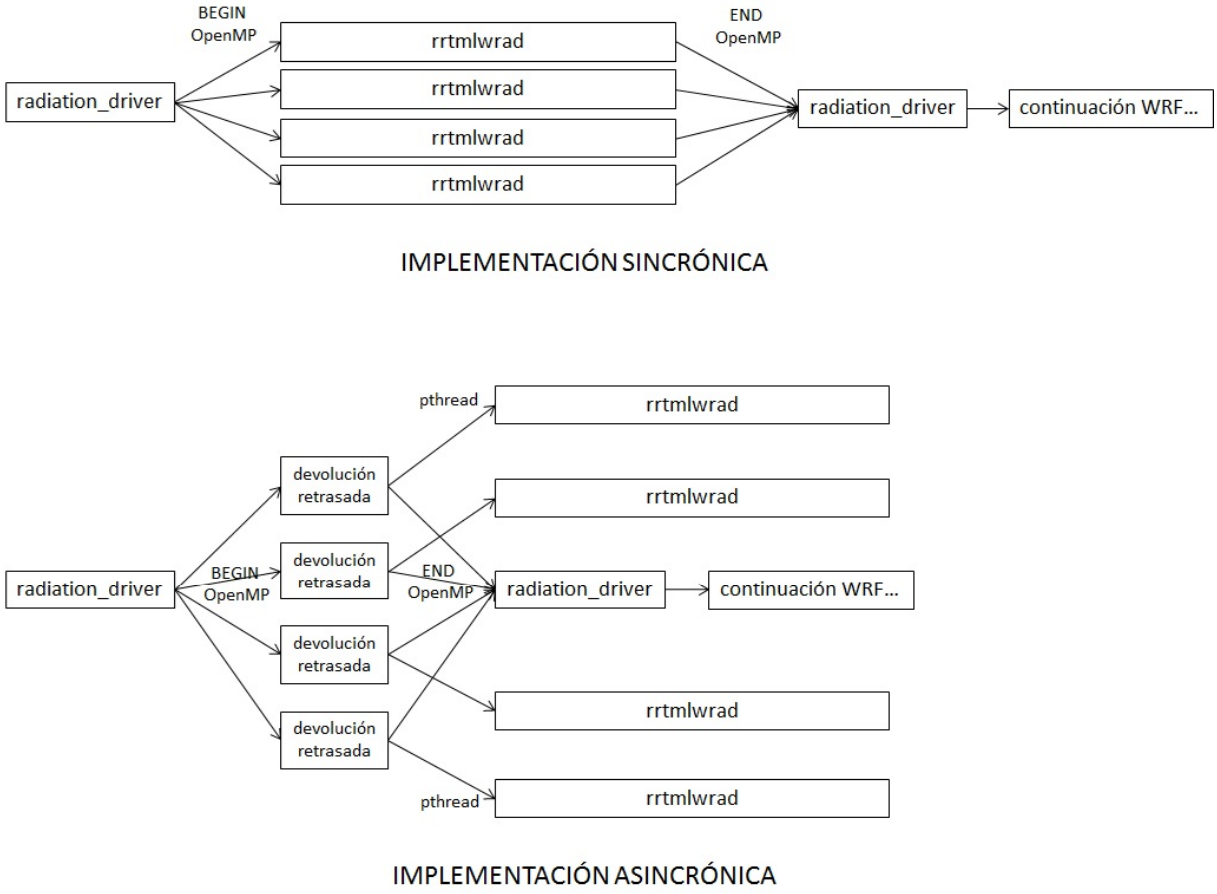

Figura 5.7: Arriba implementación sincrónica de `rrtmlwrad`, abajo implementación asincrónica de `rrtmlwrad`.

Partiendo de estas premisas, se analizó el código de `rrtmlwrad` para descubrir la mejor rutina a portar, ya que dada la extensión de código de `rrtmlwrad` resultaba inviable hacer un portado completo además que el objetivo era solamente realizar una prueba de concepto. De este análisis surge la función `rtrn`, la cual a pesar de no ser muy extensa consumía aproximadamente un 2 % de la ejecución total del WRF utilizando la configuración óptima de radiación.

El portado de `rtrn` se realizó siguiendo las mismas ideas de otros portados, pero sin realizar mayores optimizaciones dado el paradigma de ejecución. Sin embargo, se presentaron algunos inconvenientes intrínsecos al código, dada la dependencia de datos existente entre distintas iteraciones de un mismo loop. Esta dependencia pudo resolverse, ya que en ambos casos donde ocurrió, se trataba de una doble iteración, y la dependencia de datos era únicamente regida por el loop exterior, pudiéndose entonces paralelizar el loop interior. Fue por esto que se dejó la iteración exterior en C y se invocó a un kernel encargado de la iteración interior. Otra particularidad de este portado a diferencia de los otros realizados, es que corre asíncronamente tantas veces como hilos OpenMP existan, requiriendo entonces, tantas reservas de memoria como hilos concurrentes. Como las reservas se llevan a cabo únicamente al inicio de la ejecución del WRF, es necesario la asociación de cada memoria reservada a su hilo correspondiente en cada invocación. Para identificar que parte de la memoria reservada le corresponde al hilo actual, fue utilizado el atributo `name` de *pthread*, el cual fue previamente asignado en el momento de su creación.

Es importante aclarar que el módulo de radiación ya fue portado a GPU anteriormente [14], y que el código fuente del mismo es de libre acceso, sin embargo, el portado se realizó utilizando CUDA Fortran. Esto implica la utilización de una herramienta de compilación [45] de licencia privada y código cerrado, razones por las cuales se prefirió evitar su uso para este trabajo. Además de este inconveniente, el escenario utilizado para el estudio es diferente, ya que en el trabajo mencionado la función `gasabs` es la que representaba la mitad del tiempo de cómputo de radiación, cuando en la ejecución del WRF realizada para este trabajo era mucho menor. Debido a esto, y a que CUDA C permite mayor granularidad para el manejo de la GPU, es que se optó por portar el módulo de radiación en CUDA C, con foco en la función `rtrn`. De todas formas, se reconoce que el trabajo de los autores es un esfuerzo importante en la aceleración con GPUs del WRF, que sería muy útil de integrar en otro contexto.

5.5.4. Evaluación de tiempos de ejecución

Luego de culminar la implementación, se midieron los tiempos de ejecución del WRF utilizando la implementación asíncrona de `rrtmlwrad`. Éstos mostraron estar ligeramente por encima del tiempo del WRF multihilo totalmente sincrónico, por lo que se procedió a realizar un análisis minucioso para descubrir las posibles causantes.

Uno de los aspectos a tener en cuenta es el costo extra de CPU que tiene la versión asíncrona respecto a la síncrona. En primer lugar se encuentran los tiempos de transferencia de información necesarios entre el hilo principal y los hilos que realizan el trabajo asíncrono. A continuación se presentan dichas transferencias de información junto con el orden de magnitud de tiempos:

- Copia de parámetros para hilo asíncrono por parte del hilo principal: 10^{-3} segundos.
- Lectura de resultado guardado de cálculo anterior por parte del hilo principal: 10^{-4} segundos.
- Escritura de resultado obtenido por hilo asíncrono para disponibilidad de hilo principal: 10^{-5} segundos.

Sumado a esto se encuentra el tiempo de creación de hilos, el cuál tiene orden 10^{-5} segundos. Podemos concluir entonces que el sobrecosto de la solución asíncrona está en el orden de 10^{-3} segundos por llamada, el cuál no es considerable con respecto a tiempos de cálculo de `rrtmlwrad` (aproximadamente 0.8 s.).

Otro aspecto a tener en cuenta es el manejo del scheduler a nivel de CPU. El WRF original ya estaba computando por intermedio de múltiples hilos, por lo que es relevante realizar un análisis del impacto en la creación de nuevos hilos en el rendimiento. El aspecto más importante a tener en cuenta es la cantidad de núcleos reales de la CPU en donde se ejecuta la aplicación. En ambientes multicore, la recomendación general es ejecutar en forma concurrente tantos hilos como cores (reales o virtuales) existan. Más hilos pueden mejorar la performance de la aplicación en contadas ocasiones y según la intensidad de operaciones de entrada/salida, sincronización de hilos, esperas bloqueantes de recursos, etc. Dado que en grandes aplicaciones un análisis teórico es complejo, la mejor práctica es realizar pruebas bajo distintas configuraciones.

La Plataforma 1 posee un procesador de dos núcleos con tecnología Hyperthreading, esta tecnología aparenta para el sistema operativo tener el doble de cantidad de núcleos

físicos, de forma tal de aprovechar recursos ociosos del núcleo para un hilo. La mejora en performance de ésta tecnología es dependiente del tipo de trabajo que se realice, y asumir que cada core es dedicado podría implicar una pérdida en eficiencia [43]. El WRF es ejecutado en un CPU de dos núcleos mediante 4 hilos, y al agregar el paradigma asíncrono de radiación, llegan a ejecutar 8 hilos en simultáneo, lo cual puede degradar los tiempos de ejecución.

Por otra parte, se analizaron los tiempos de ejecución de la rutina `rtrn` portada a CUDA, para la misma se obtuvo una duración de 4 ms por llamada, mientras que el código original demoraba un cuarto de ese tiempo. Sin embargo si de esos 4 ms, el tiempo utilizado por CPU fuera menor a 1 ms, nos encontraríamos ante una situación de ganancia. Las mediciones de tiempos resultaron en 2 ms para los kernels y 2 ms para transferencias de memoria. Asumiendo que el CPU queda libre durante la ejecución de los kernels, en gran parte del tiempo de transferencias el recurso de CPU es solicitado, por lo que el tiempo en CPU para esta función podría ser mayor al de la función original simplemente por transferencias. Además, si bien el tiempo transcurrido entre el inicio del primer kernel y la finalización del último representaban 2 ms, esto no necesariamente significa una liberación total del recurso de CPU durante todo ese tiempo. Luego de cada kernel ejecutado en los loops con dependencia de datos descritos en la Sección 5.5.3, se realiza una sincronización para luego comenzar el siguiente kernel, lo cual se estaba llevando a cabo aproximadamente 150 veces por iteración exterior, es decir 150 sincronizaciones y llamadas a GPU, las que representan un tiempo extra considerable de uso de CPU [32]. Buscando reducir esta cantidad de llamadas y sincronizaciones, se trasladaron las iteraciones realizadas en C hacia GPU, lo que requirió un mayor nivel de sincronización entre los hilos, pero evitó los costos antes descritos. Luego de esta mejora se constató una reducción del tiempo consumido en aproximadamente un 25 %. A pesar de esto, no resulta beneficiosa la utilización de esta función portada a CUDA, para que la misma sea beneficiosa se debería portar todo el módulo de radiación, o al menos una sección mayor del mismo, para así evitar tranferencias y liberar el CPU.

Se efectuaron mediciones del WRF ejecutado con dos y con un hilo. Al trabajar con dos hilos y ejecutar `rrtmlwrad` asincrónicamente, se notó una leve mejora respecto al WRF base ejecutado con 2 hilos.

Al ejecutar el WRF con un solo hilo y `rrtmlwrad` asincrónicamente, se tiene disponible un core más que será utilizado para realizar todo el trabajo asíncrono, el cual simula un recurso extra de cómputo, sin interferir con otro recurso de procesamiento. Al hacer esta prueba (en experimentos no formalizados), se notó una mejora en los tiempos muy superior a la alcanzada con dos hilos, llegando a ocultar casi la totalidad del tiempo de ejecución de `rrtmlwrad`. Es importante notar que con la configuración actual, el módulo de radiación se adjudica un tiempo de ejecución bajo en el total del WRF, por lo que un buen speedup a nivel del módulo puede implicar un speedup poco considerable respecto al WRF completo.

5.6. Otras funciones analizadas

Analizando la salida de los profilers, se pueden identificar muchas subrutinas pertenecientes a dos módulos, el módulo `module_small_step_em` y el módulo `module_big_step_utilities`. Éstas se analizaron ya que subrutinas presentes en el mismo módulo suelen llamarse en secciones cercanas de código del WRF, lo que podría aprovecharse para evitar transferencias portando varias que compartan parámetros.

5.6.1. module_small_step_em

Dentro del módulo module_small_step_em se encuentran las subrutinas advance_uv, advance_mu_t, advance_w, calc_p_rho, sumflux y small_step_prep. Todas estas son llamadas desde la subrutina solve_em, la cual es la principal encargada de avanzar la grilla del pronóstico de entrada una unidad de tiempo.

Todas estas subrutinas se parecen, tienen costos similares de tiempo de ejecución por llamada, no son demasiado extensas, y contienen muchas iteraciones realizando cálculos sobre puntos de matrices de 2 y 3 dimensiones pasadas por parámetro. Cada llamada es realizada por todos los hilos definidos para el procesador, los cuales se encargan de realizar las iteraciones y cálculos sobre distintas secciones de las matrices.

Dada la naturaleza de las subrutinas, las mismas tienen la capacidad de ser computadas con fuerte paralelismo, por lo que resultan candidatas de ser portadas a CUDA, por lo que se procedió a analizarlas más detalladamente, para identificar si las reservas y transferencias de memoria no hacen inútil su portado parcial.

Durante el análisis se relevaron los siguientes datos sobre cada una de las mismas: tiempo de ejecución, cantidad de parámetros, tamaño de los parámetros, último uso de los parámetros de entrada antes de la llamada, primer uso de los parámetros de salida luego de la llamada, lugar de llamada y parámetros compartidos entre ellas.

Los parámetros significativos de estas subrutinas son matrices de 3 dimensiones del mismo tamaño que mantienen la mayoría de las veces constante, y cuando lo cambian (solamente a un tamaño diferente), en llamadas posteriores vuelven a tener el tamaño anterior, lo cual significa que las reservas de espacio en la memoria de GPU podrían ser realizadas una única vez. Este tamaño es de aproximadamente 50^3 reales de 4 bytes (500 KB aproximadamente), y existen entre 5 y 15 de estas matrices de entrada y salida para cada subrutina.

Al hacer pruebas experimentales para comparar los tiempos de ejecución de las subrutinas con los tiempos potenciales de transferencia de datos a GPU se constató que los tiempos de transferencias son al menos un orden superior, y a pesar de que varias subrutinas se llaman en lugares cercanos y comparten parámetros que no cambian entre sus llamadas, resulta imposible que el tiempo de ejecución supere al tiempo de transferencia aun si se portaran varias subrutinas, por lo que se descartó el portado de este módulo.

5.6.2. module_big_step_utilities

Dentro del módulo module_big_step_utilities se encuentran las subrutinas zero_tend, horizontal_diffusion, phy_prep, calc_cq, curvature, coriolis, rhs_ph y horizontal_pressure_gradient. Luego de identificadas las mismas se procedió a realizar un análisis similar al realizado el módulo module_small_step_em.

La subrutina phy_prep es llamada desde first_rk_step_part1 y la subrutina calc_cq es llamada desde rk_step_prep, las cuales son llamadas desde solve_em. Estas tienen similar naturaleza a las subrutinas del módulo module_small_step_em detalladas anteriormente, forma de parámetros, extensión de código, paralelismo, etc. Ambas son subrutinas

aisladas de otras candidatas a ser portadas, y portarlas conlleva un alto costo de transferencias a GPU, por lo que se descartaron.

El resto de las subrutinas mencionadas de este módulo son llamadas desde `rk_tendency`, la cual es llamada desde `solve_em`. A diferencia de las subrutinas del módulo `module_small_step_em`, éstas no son paralelizadas internamente con OpenMP, sino que toda la subrutina `rk_tendency` es paralelizada con OpenMP, lo que hace más difícil el portado, ya que al dejar parte de `rk_tendency` sin portar, las salidas de las subrutinas portadas deberían sincronizarse con todos los hilos OpenMP para que estos puedan utilizarlas. La naturaleza de estas subrutinas, con la excepción de `zero_tend`, se mantiene similar a las anteriores, son ejecutadas por todos los hilos que hacen distintos cálculos sobre distintas secciones de matrices de entrada, tienen similar cantidad y forma de parámetros, extensión de código, etc.

La subrutina `zero_tend` no hace más que setear un rango de una matriz recibida como parámetro con ceros, por lo tanto la misma demora muy poco y la transferencia de esta matriz a memoria de GPU hace inviable su portado, sin embargo, se intentó optimizar usando funciones intrínsecas de Fortran para la asignación de matrices, pero las mismas resultaron ser contraproducentes aumentando el tiempo de ejecución.

Las subrutinas `rhs_ph` y `horizontal_pressure_gradient` son llamadas consecutivamente, pero a pesar de esto tienen demasiadas matrices de gran tamaño (50^3 reales) que no comparten y que no se pueden ocultar, por lo tanto se descartó su portado.

Las subrutinas restantes son llamadas con el siguiente patrón: `coriolis`, `curvature`, y 3 llamadas a `horizontal_diffusion`. Estas 3 llamadas se realizan solamente en una de cada tres llamadas a `rk_tendency`, sin embargo se atribuyen un gran porcentaje de tiempo total de ejecución del módulo. Por el tamaño, cantidad y reuso de parámetros entre las 3 subrutinas es inviable portar las mismas por separado, sin embargo, del análisis de las 5 llamadas en conjunto, se desprende que al portarlas se necesitan transferir solamente 13 matrices significativas de entrada y 6 de salida, de las cuales se podría ocultar la transferencia de 10 matrices de entrada dada la naturaleza del código de `rk_tendency`, lo que hace que el tiempo de ejecución se acerque al tiempo de transferencia. A pesar de la conclusión alcanzada, el tiempo de ejecución sigue sin ser suficiente, y existen muchas matrices con las que trabajar, lo que hace que no todas puedan ser alojadas en memoria compartida en el portado, por lo que los accesos a memoria global en el kernel serían demasiados, lo que podría hacerlo costoso.

En base a este análisis fue que se tomó la decisión de no portar estas subrutinas, sin embargo, se deja una alternativa de portado a futuro en la medida en que las arquitecturas de GPU mejoren los tiempos de transferencias o se disponga de otros módulos del WRF en GPU que permitan directamente evitar transferencias.

5.7. Evaluación final de resultados

Esta sección describe las pruebas de performance realizadas luego de la integración de todos los módulos portados, así como también la evaluación de la calidad de los resultados numéricos obtenidos, los cuales podrían haber sufrido alteraciones debido a cambios de compilador, de paradigmas de ejecución, entre otros.

La ejecución de la pruebas fueron realizadas con la Plataforma 1.

5.7.1. Estudio de calidad de resultados

Para evaluar la calidad de las predicciones obtenidas, se utilizó un script proporcionado por el IMFIA que toma como entrada la salida del WRF, y a partir de la misma obtiene los valores de distintas variables climáticas en la zona donde se encuentran los molinos de vientos en la "Sierra de los Caracoles". Para cada variable, se obtiene un valor por cada hora de predicción, en cada nivel de la atmósfera, cuando corresponde.

A continuación se detalla la evaluación de la calidad de los resultados, indicando para cada módulo portado, si éste afecta considerablemente o no en las predicciones obtenidas. Para esto se compararon las dos variables de salida del WRF más relevantes en la predicción eólica, *V12p1* y *UST*. La primera es la velocidad del viento, mientras que la segunda es la velocidad de fricción [8], la cual afecta directamente la componente vertical de la velocidad del viento.

Al ejecutar el script, el mismo reflejó que tanto los portados de las rutinas `sintb` y `bdy_interp`, como los de `slope_wsm3` no reflejan variaciones en los resultados, más allá de diferencias esperadas por redondeos en cálculos de punto flotante. De igual modo la rutina `rk_scalar_tend` integrada al modelo no afectó las estimaciones de las variables comparadas.

En lo que respecta a la integración del portado del módulo WSM5, se corroboró que la inclusión de la GPU para su cálculo no afecta la precisión de las predicciones. Por otra parte, sí se observaron alteraciones al comparar las salidas de las distintas configuraciones del módulo de microfísica, utilizando el WSM5 en lugar del WSM3. Al cambiar de configuración, la diferencia relativa en la variable *UST* resulta relevante, en cambio, a pesar de que *V12p1* también presenta diferencias, éstas no resultan significativas para la predicción.

En las Figuras 5.8 y 5.9, se muestran las salidas de la variable *V12p1* en los primeros niveles de la atmósfera para las dos configuraciones mencionadas, y la diferencia entre ambas respectivamente. En las Figuras 5.10 y 5.11, se muestra la misma información para la variable *UST*. A pesar de observar diferencias no despreciables entre la utilización de ambos módulos, no podemos afirmar si estas se deben a un error o a una mejor aproximación. Para ello resulta necesario realizar un estudio junto a un experto con conocimiento del modelo de física, que valide los datos y estime el impacto en corridas más prolongadas en el tiempo en términos de la amplificación del error. Esto excede al alcance de este proyecto, por lo que se plantea como trabajo futuro.

También resulta importante mencionar que si bien el portado del módulo de radiación no afectó significativamente las estimaciones de estas variables, sí presentó diferencias considerables en variables de menor interés que requieren de un postprocesamiento para ser analizadas. Análogamente a lo dicho en el párrafo anterior, esto merece un estudio posterior para establecer la correctitud de la propuesta.

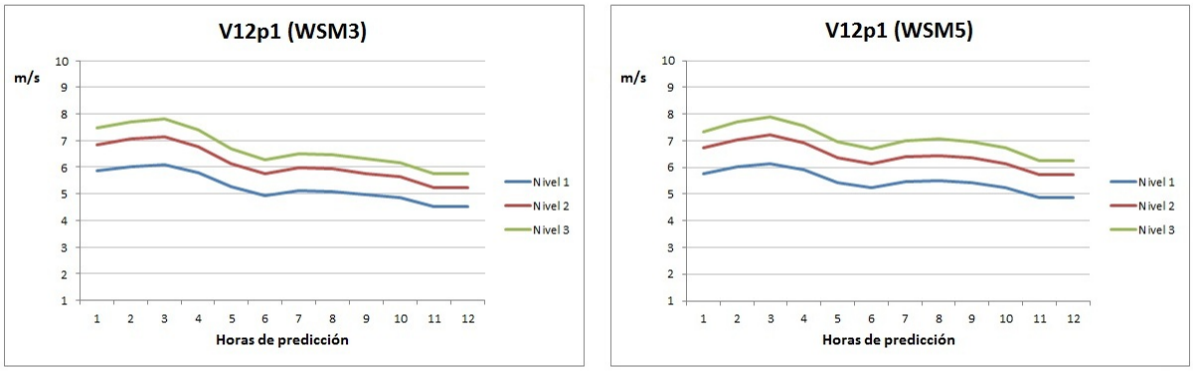

Figura 5.8: A la izquierda, gráfica de la variable de salida *V12p1* para la ejecución del WRF utilizando el WSM3. A la derecha, gráfica de la variable de salida *V12p1* para la ejecución del WRF utilizando el WSM5.

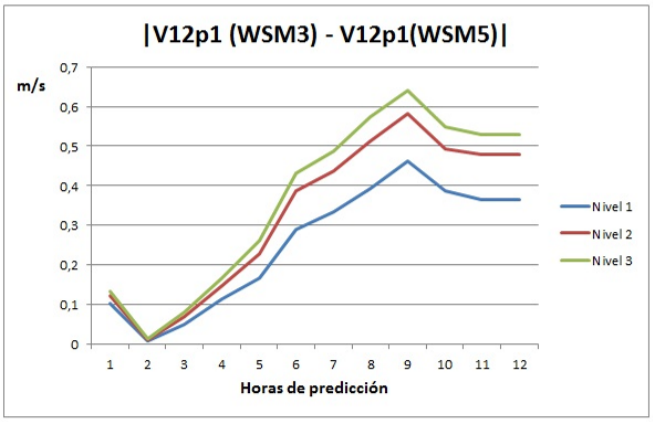

Figura 5.9: Diferencia entre las variables de salida *V12p1* de las ejecuciones del WRF utilizando el WSM3 y el WSM5.

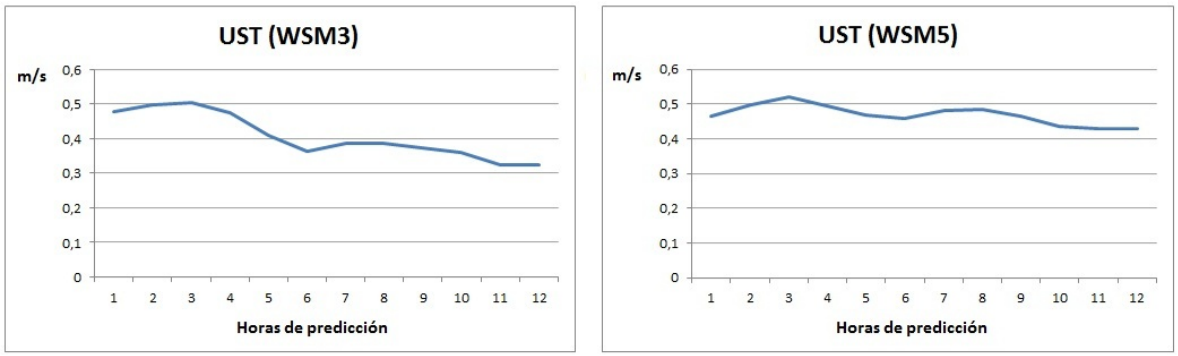

Figura 5.10: A la izquierda, gráfica de la variable de salida *UST* para la ejecución del WRF utilizando el WSM3. A la derecha, gráfica de la variable de salida *UST* para la ejecución del WRF utilizando el WSM5.

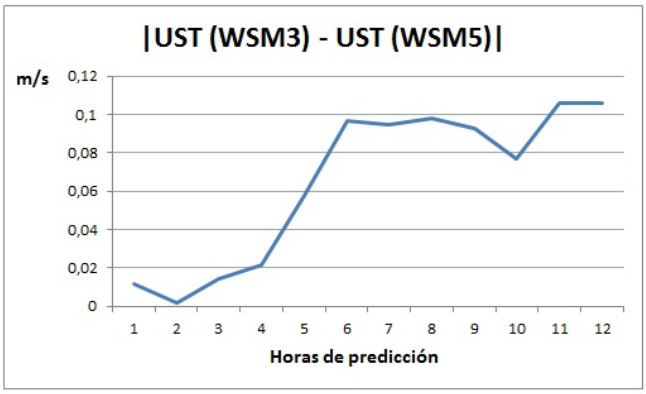

Figura 5.11: Diferencia entre las variables de salida *UST* de las ejecuciones del WRF utilizando el WSM3 y el WSM5.

5.7.2. Estudio de desempeño computacional

Para evaluar los tiempos de ejecución obtenidos, se procedió a realizar dos configuraciones distintas en el WRF, ambas contienen los módulos portados de `bdy_interp` y `rk_scalar_tend`, pero difieren en el modelo de físicas utilizado. Uno usa el portado del WSM5, y el otro utiliza el WSM3 para ejecutar el portado de `slope_wsm3`. Dentro de estas configuraciones, se comparó el tiempo original de las mismas (single-thread y multi-thread) contra su ejecución en GPU. En la Tabla 5.3 pueden apreciarse los resultados obtenidos. También se recapitulan los speedups logrados para todas las rutinas portadas en la Tabla 5.4. Es importante aclarar que los módulos en Fortran fueron compilados utilizando la bandera de optimización `-O3`.

Descripción de ejecución	Cantidad hilos	Tiempo	Aceleración (%)
WRF original, modelo de físicas: WSM3	4	1:43:23 hs	-
WRF `bdy_interp` + `rk_scalar_tend` + `slope_wsm3` GPU	4	1:29:48 hs	14 %
WRF original, modelo de físicas: WSM5	4	2:02:22 hs	-
WRF `bdy_interp` + `rk_scalar_tend` + WSM5 GPU	4	1:22:42 hs	32 %
WRF original, modelo de físicas: WSM3	1	3:46:42 hs	-
WRF `bdy_interp` + `rk_scalar_tend` + `slope_wsm3` GPU	1	3:08:02 hs	17 %
WRF original, modelo de físicas: WSM5	1	4:29:34 hs	-
WRF `bdy_interp` + `rk_scalar_tend` + WSM5 GPU	1	2:21:17 hs	47 %

Tabla 5.3: Ejecuciones del WRF realizadas para la evaluación de performance.

Rutina	Tiempo CPU	Tiempo GPU	Speedup
`bdy_interp`	1.15×10^{-2} s	2.35×10^{-3} s	4.80 ×
`slope_wsm3`	3.00×10^{-4} s	1.10×10^{-4} y 4×10^{-5} s	3.00 × y 7.00 ×
`rk_scalar_tend`	3.20×10^{-3} s	2.80×10^{-3} s	1.15 ×
WSM5	6.00×10^{-2} s	9.00×10^{-3} s	6.70 ×

Tabla 5.4: Tiempos de ejecución individuales de rutinas portadas a CUDA vs. ejecución en CPU con 4 hilos.

Al analizar los resultados obtenidos, se aprecian aceleraciones muy significativas, las cuales se obtienen gracias al portado de rutinas de gran impacto en el tiempo de ejecución

total del modelo. Cabe aclarar que dadas las condiciones del código del WRF, pueden existir muchos módulos a ser portados a GPU que logren mayores speedups individuales, pero con mejoras globales menores a las obtenidas.

Se aprecia que la mayor aceleración se da al incluir el módulo portado por J. Michalakes *et al.* [20]. Como los autores mencionan, este módulo reúne condiciones muy favorables para un portado exitoso, además de ocupar un tiempo de ejecución cercano al 25 % del WRF generalmente. Al integrar este módulo se nota además un tiempo de ejecución más bajo que ejecutando con WSM3, lo que permite no solo reducir el tiempo, sino también acceder a predicciones climáticas de mayor precisión como se describe en la Sección 5.4.1.

Si comparamos los speedups individuales por rutina obtenidos con los de otros trabajos relevados en la Sección 3.4, los mismos son similares a la mayoría, con la excepción de los trabajos de B. Huang. *et al.*, los cuales presentan resultados no alineados con el resto de los esfuerzos. Estos últimos, a pesar de obtener resultados asombrosos, portan rutinas de poco impacto en el tiempo de ejecución total del WRF, y lo hacen utilizando banderas de optimización de CUDA (`use_fast_math`) que deterioran la calidad de resultados, además de comparar sus versiones únicamente contra configuraciones single-thread del WRF. Por otra parte, el escenario que utilizan para validar sus resultados tiene dimensiones mucho mayores que la de otros trabajos, lo que puede hacer que la relación de tiempo de cómputo CPU/GPU aumente notablemente. Dicho escenario es aproximadamente 30 y 80 veces mayor que los utilizados en este trabajo (433×308×35 vs 71×61×54 y 40×40×54).

Otro punto importante a destacar es que muchas de las rutinas portadas a GPU tienen altos costos de transferencia de datos entre host y device, el cual se verá disminuido en la medida que más rutinas sean portadas a CUDA, ya que se podrán utilizar datos de salida de unas como entrada de otras, sin tener que pasar por CPU, lo que disminuirá considerablemente los tiempos de ejecución.

Capítulo 6

Conclusiones y trabajo futuro

Analizando los objetivos planteados al inicio del proyecto, se puede afirmar que éstos han sido alcanzados con éxito, planteando además nuevos caminos para profundizar el trabajo en esfuerzos futuros.

Inicialmente, el proyecto proponía investigar la posibilidad de uso de GPUs para la mejora de performance del WRF con especial énfasis en el uso de la herramienta desarrollada por el IMFIA. La etapa de profiling estableció las líneas de trabajo que permitieron concentrar el esfuerzo en los módulos del WRF que más trabajo de cómputo poseen, logrando así que su potencial portado influya en mayor proporción en el tiempo total de ejecución. En esta etapa se pudo observar que en la ejecución original no existen módulos que acaparen la mayor parte del tiempo de cómputo, por lo que se debió portar más de un módulo para lograr avances significativos en la disminución de los tiempos de ejecución totales. También se notó que las rutinas que consumían mayor tiempo de ejecución, lo hacían en el transcurso de cientos de miles de llamadas, por lo que se debió diseñar distintas estrategias de portado para asignar mayores cantidades de trabajo a la GPU, logrando así que la misma no quedara subutilizada.

Siguiendo las estrategias descritas en el párrafo anterior, fue que se portaron y optimizaron las funciones `sintb` y `slope_wsm3` en GPU, tomando como punto de partida el código original del WRF. Para `sintb`, se logró un speedup en la función de aproximadamente 4.8× respecto a su versión multihilo, y una reducción mayor a 10 % en el tiempo total del WRF original. Para `slope_wsm3`, se logró un speedup cercano a 7×, reduciendo en un 3 % el tiempo total de ejecución multihilo del WRF original.

Es importante notar el esfuerzo realizado en la reutilización de trabajo existente. Esta tarea se marcó como un objetivo importante del proyecto dado que los aportes existentes siguen la dirección general de avanzar en el portado del WRF a GPU. El trabajo reutilizado corresponde al módulo WSM5 y al de Advección, que logran speedups de 6.7× y 1.15× respectivamente sobre estos módulos, los cuales logran disminuir el tiempo de ejecución total multihilo del WRF cerca de un 25 %.

Un aspecto interesante a destacar, es el cambio de configuraciones del WRF para el uso de distintos módulos en la realización del cálculo de ciertas magnitudes. En nuestro caso, la utilización del WSM5 y el WSM3. Se puede apreciar que, a pesar de utilizar un modelo más pesado para la realización de los cálculos, se abren nuevas posibilidades de módulos que son naturalmente grandes candidatos a ser portados, y por ello en GPU pueden disminuir el tiempo total de ejecución a pesar de aumentarlo en su versión original, además de mejorar potencialmente la precisión de los pronósticos generados.

Una de las ideas más prometedoras respecto a las implementaciones realizadas, fue la introducción del paradigma asíncrono en el módulo de radiación, tomando ventaja de la naturaleza del cálculo realizado en períodos fijos (configurables) de tiempo. A pesar de que dicho trabajo fue realizado con un enfoque de prueba de concepto, se obtuvieron resultados alentadores, que podrán generar buenos resultados en caso de continuar el trabajo realizado. La introducción de este paradigma permite explorar nuevos caminos en la utilización de módulos de radiación mas pesados, como así también variar la frecuencia con la que éstos son llamados en la ejecución del modelo, pudiendo potencialmente mejorar la calidad de los resultados obtenidos.

Los módulos `module_small_step_em` y `module_big_step_utilities` fueron foco de un análisis que culminó en la decisión de no ser portados. Para ambos módulos se poseen varias rutinas que naturalmente pueden ser paralelizadas, pero el mayor problema enfrentado fue el tamaño total de los parámetros con los que se invocan, haciendo que los tiempos de transferencia de datos sean al menos un orden mayor al tiempo de cómputo. Futuras arquitecturas podrían resolver estos problemas, como también el avance de nuevos módulos portados que hagan innecesarias las transferencias por ya poseer los datos requeridos en GPU.

En cuanto a la evaluación del trabajo realizado, se puede afirmar que los resultados obtenidos superan las expectativas y además son competitivos respecto a otros trabajos existentes en la literatura. También es importante notar que el punto de partida de este esfuerzo fue la de optimizar la ejecución de una aplicación bajo un escenario real, en contraposición con otros trabajos relevados los cuales tomaron foco en módulos naturalmente paralelizables del modelo, utilizando un escenario conveniente para la evaluación de resultados. Además, el ambiente de ejecución de este proyecto es multihilo, por lo que la evaluación final no fue realizada bajo un ambiente monohilo como es usualmente realizado en gran parte de los trabajos relevados.

A partir de los resultados obtenidos, se presentó el artículo "GPU Acceleration of a tool for Wind Power Forecasting" al *13th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2013*, que describe la problemática de este proyecto, y específicamente el trabajo realizado en las rutinas `sintb` y `bdy_interp`. Dicho artículo fue aceptado por los revisores del congreso. La versión final del artículo puede ser consultada en el Apéndice.

Se considera que existen diversos caminos interesantes a analizar que pueden ser considerados como trabajos futuros.

En primer lugar, es importante mencionar que aún se considera de utilidad profundizar la etapa de Profiling para encontrar más módulos que resulten de interés portar a GPU. A su vez es necesario avanzar en la validación de los resultados numéricos que presentaron ciertas diferencias con los obtenidos utilizando versiones originales del WRF. También es necesario evaluar las propuestas en GPUs de última generación (Kepler), que presentan desempeños muy superiores a las tarjetas utilizadas en este proyecto [41].

En segundo lugar, parece prometedor seguir avanzando en la línea de trabajo del paradigma asíncrono del módulo de radiación, dado que cuanto más trabajo sea asignado a la GPU permitirá liberar más carga de trabajo a la CPU. De esta forma es posible dejar lo menos ociosa posible la tarjeta, para aprovechar por tanto, capacidad de cómputo potencial que allí se encuentra.

Sumando a las líneas de trabajo, se considera que intercambiar los módulos utilizados para la simulación de ciertas magnitudes, puede abrir nuevos caminos para obtener mejoras tanto de performance como de precisión de resultados. Como ya fue mencionado, una prueba realizada con un módulo distinto que a priori era más pesado computacionalmente pero con mejor precisión numérica, abrió la posibilidad a reutilizar trabajo existente en su portado a GPU, que finalmente culminó en menores tiempos de ejecución que el módulo que era utilizado originalmente.

Sabiendo de la existencia de herramientas de portado automático de código a GPU [31], se considera como un camino interesante a explorar. A pesar de que por el momento dichas herramientas no generan código comparable con un portado manual, es importante seguir observando futuras mejoras de las mismas.

Finalmente, estudiar otros lenguajes de programación para GPUs es un trabajo interesante a considerar, sobretodo en lenguajes estándares como OpenCL [28], que permiten no estar ligados a un fabricante en particular (como lo es CUDA con NVIDIA). Esto podría permitir el uso de distintas GPUs, obteniendo un ambiente híbrido para realizar el trabajo, posiblemente asignando distinto trabajo a las tecnologías que exploten de mejor manera el problema a resolver.

Anexo I - Problemas con profilers

Resulta necesario comentar problemas que se encontraron durante la utilización de las herramientas de profiling descritas en el informe. Para entender dichos problemas, es necesario entender en forma general cuales son los datos que contienen los resultados generados por éstas herramientas. En términos generales, las métricas que más interesan son el porcentaje de tiempo de ejecución de cada función del programa ejecutado respecto al tiempo de ejecución total, y el call-tree que permite realizar un seguimiento de la forma de ejecución de cada función.

La primera métrica es la principal para detectar cuáles son los módulos en los que conviene invertir esfuerzo para su optimización, ya que un speedup alcanzado en un módulo tiene un impacto proporcional a su tiempo de ejecución en la aplicación completa. La segunda métrica es útil para entender la lógica de ejecución de los módulos, establecer relaciones padre-hijo entre funciones y además conocer la cantidad de llamadas realizadas de una función para cada uno de sus padres.

Un detalle no menor es que el resultado generado por la herramienta *gprof* incluía información inconsistente en el call-tree del profiling, dado que establecía relaciones inexistentes en el código fuente entre funciones padre-hijo. Por este motivo se tomó la decisión de considerar una segunda herramienta para el profiling. La herramienta *valgrind* fue consistente con el call-tree generado, y además sus resultados fueron coherentes con los resultados de *gprof*; lo cual da la seguridad que el objetivo de la etapa de profiling se alcanzó de forma satisfactoria.

Éste problema motivó una investigación más a fondo sobre el impacto de aplicaciones multihilo en el comportamiento de los profilers utilizados. El profiler *gprof* no es apto para análisis de aplicaciones multihilo dado que solo realiza el profiling del hilo principal, no realizando las tareas necesarias para la recopilación de la información cada vez que un hilo es creado. En el caso del *valgrind*, en ambientes multihilo los hilos son serializados en la ejecución mediante el uso de un lock que solo puede tener como dueño un único hilo.

Conociendo las implicancias de una aplicación multihilo en las herramientas de profiling, se decidió compilar la aplicación para que se ejecute utilizando un único hilo para así de esta manera tener un entorno donde el profiler *gprof* se ejecute de forma correcta. El impacto de éste cambio en la herramienta *valgrind* es mínima, dado que ambientes de un solo hilo son soportados y además en ambientes multihilo éstos eran serializados y unificadas las métricas de cada uno, obteniendo a efectos prácticos resultados equivalentes. Realizado dicho cambio se procedió a realizar nuevamente el profiling obteniendo los resultados esperados; el call-tree generado por el *gprof* fue coherente y el *valgrind* obtuvo resultados similares al ambiente multihilo.

Es necesario aclarar también una diferencia fundamental entre los informes generados por ambos profilers. La herramienta *valgrind* realiza el profiling de funciones en bibliotecas

GNU, como son `exp`, `pow` y `log`, entre otras, mientras que el *gprof* no lo hace dado que dichas bibliotecas no fueron compiladas con la flag de profiling (solamente se realizó en el WRF). Ésto implicó que dichas funciones sean detalladas en el profiling del *valgrind* pero no en el del *gprof*, haciendo que en las funciones del WRF donde se utilizan impacte o no en su tiempo de ejecución total. En el caso del *gprof* el tiempo consumido por las funciones GNU no son contabilizadas y en el *valgrind* son contabilizadas como funciones independientes. Ésto motivó a un análisis de padre-hijo sobre las funciones de bibliotecas ajenas al WRF (`exp`, `pow` y `log`) dado que éstas consumen un tiempo considerable respecto al tiempo de ejecución total, encontrando que eran pocas las funciones del WRF que hacían uso de ellas, dando mayor relevancia a dichas funciones en la lista de funciones posibles a optimizar.

Anexo II - Problemas en implementación de radiación asíncrona

En fases iniciales de testing del desarrollo de la solución descrita en la Sección 5.5.2, surgieron problemas de inestabilidad de la ejecución del programa. El problema ocurría cuando dos hilos en forma concurrente realizaban trabajo sobre una misma función, aparentando sobreesribir valores de variables locales entre sí. Este comportamiento no fue considerado inicialmente dado que cada ejecución de una función tiene un contexto propio de variables en la pila para el hilo en ejecución.

Una investigación sobre la documentación existente acerca del manejo de Fortran de las variables locales reveló que al no ser un lenguaje que soporte en forma nativa hilos concurrentes (sino que depende de bibliotecas fuera del estándar del lenguaje, como OpenMP), no es estrictamente necesario respetar contextos individuales para hilos de ejecución.

En versiones anteriores a Fortan 90, todas las variables locales de las funciones internamente eran definidas como globales, en donde el estándar definía que luego de terminada una ejecución de la función, el valor de las variables locales se consideraba indefinido. Aún así se podría comprobar como valores de variables aún no asignadas en una función, pueden contener valores de la última ejecución del programa. De hecho es necesario considerar flags de compilación especiales para soportar funciones recursivas, dado que éste manejo de variables locales no lo permitiría.

Al utilizar Fortran 90, éste comportamiento es dependiente del compilador que se utilice. Si se utiliza el compilador Intel de Fortran, se sigue obteniendo el mismo comportamiento [16]. Para gfortran depende del máximo tamaño de las variables locales de la función, donde si superan cierto umbral se comporta igual que el compilador de Intel [15].

Para ambos compiladores existen flags de compilación (p.e: `-frecursive`) que permiten modificar el manejo de variables locales de las funciones, pero se debe leer la documentación con suficiente cuidado ya que su comportamiento puede depender entre versiones del compilador, además de medir el impacto del cambio, ya que según el contexto en que la función es llamada, se podrían suponer valores iniciales en variables locales de la misma debido a invocaciones anteriores, los cuales no estarán asignados utilizando estas flags.

Anexo III - Tablas de resultados

A continuación se presentan todos los resultados relevantes obtenidos en el proceso de optimización del WRF.

Descripción	**Tiempo**	**Speedup sobre versión multihilo CPU**
`bdy_interp` y `sintb` originales. Ejecución con 1 hilo en CPU	2.40 $\times 10^{-2}$ s	0.48×
`bdy_interp` y `sintb` originales. Ejecución con 4 hilos en CPU	1.15 $\times 10^{-2}$ s	-
`bdy_interp` y `sintb` 1ra versión CUDA. Ejecución en GPU	5.80 $\times 10^{-2}$ s	0.20×
`bdy_interp` y `sintb` 2da versión CUDA. Ejecución en GPU	5.00 $\times 10^{-2}$ s	0.23×
`bdy_interp` y `sintb` 3ra versión CUDA. Ejecución en GPU	8.20 $\times 10^{-3}$ s	1.40×
`bdy_interp` y `sintb` 4ta versión CUDA. Ejecución en GPU	2.35 $\times 10^{-3}$ s	4.80×

Tabla 6.1: Tiempos de ejecución individuales de las distintas versiones de las rutinas `sintb` y `bdy_interp` portadas a CUDA.

Descripción	**Tiempo**	**Speedup sobre versión multihilo CPU**
`slope_wsm3` original. Ejecución con 4 hilos en CPU	3.00 $\times 10^{-4}$ s	-
`slope_wsm3` 1ra versión. 1er llamada a la rutina. Ejecución en GPU	4.50 $\times 10^{-4}$ s	0.75×
`slope_wsm3` 2da versión. 1er llamada a la rutina. Ejecución en GPU	1.30 $\times 10^{-4}$ s	2.50×
`slope_wsm3` 3ra versión. 1er llamada a la rutina. Ejecución en GPU	1.30 $\times 10^{-4}$ s	2.50×
`slope_wsm3` 4ta versión. 1er llamada a la rutina. Ejecución en GPU	1.10 $\times 10^{-4}$ s	2.80×
`slope_wsm3` 1ra versión. 2da llamada a la rutina. Ejecución en GPU	4.50 $\times 10^{-4}$ s	0.75×
`slope_wsm3` 2da versión. 2da llamada a la rutina. Ejecución en GPU	1.00 $\times 10^{-4}$ s	3.00×
`slope_wsm3` 3ra versión. 2da llamada a la rutina. Ejecución en GPU	4.60 $\times 10^{-5}$ s	6.50×
`slope_wsm3` 4ta versión. 2da llamada a la rutina. Ejecución en GPU	4.20 $\times 10^{-5}$ s	7.00×

Tabla 6.2: Tiempos de ejecución individuales de las distintas versiones de las rutina `slope_wsm3` portada a CUDA.

Descripción	**Tiempo**	**Speedup sobre versión multihilo CPU**
WSM5 original. Ejecución con 4 hilos en CPU	6.00 $\times 10^{-2}$ s	-
WSM5 portado a CUDA. Ejecución en GPU	9.00 $\times 10^{-3}$ s	6.70 ×

Tabla 6.3: Tiempos de ejecución individuales de la rutina WSM5 portada a CUDA.

Descripción	**Tiempo**	**Speedup sobre versión multihilo CPU**
`rk_scalar_tend` original. Ejecución con 4 hilos en CPU	3.20 $\times 10^{-3}$ s	-
`rk_scalar_tend` flujo portado a CUDA. Ejecución en GPU	2.80 $\times 10^{-3}$ s	1.15 ×

Tabla 6.4: Tiempos de ejecución individuales del flujo de la rutina `rk_scalar_tend` portada a CUDA.

Descripción de ejecución	**Cantidad hilos**	**Tiempo**	**Aceleración (%)**
WRF original, modelo de físicas: WSM3	4	1:43:23 hs	-
WRF `bdy_interp` GPU, modelo de físicas: WSM3	4	1:32:55 hs	11 %
WRF `slope_wsm3` GPU, modelo de físicas: WSM3	4	1:40:37 hs	3 %
WRF `bdy_interp` + `rk_scalar_tend` + `slope_wsm3` GPU, modelo de físicas: WSM3	4	1:29:48 hs	14 %
WRF original, modelo de físicas: WSM5	4	2:02:22 hs	-
WRF WSM5 GPU, modelo de físicas: WSM5	4	1:33:36 hs	23 %
WRF `bdy_interp` + `rk_scalar_tend` + WSM5 GPU, modelo de físicas: WSM5	4	1:22:42 hs	32 %
WRF original, modelo de físicas: WSM3	1	3:46:42 hs	-
WRF `bdy_interp` + `rk_scalar_tend` + `slope_wsm3` GPU	1	3:08:02 hs	17 %
WRF original, modelo de físicas: WSM5	1	4:29:34 hs	-
WRF `bdy_interp` + `rk_scalar_tend` + WSM5 GPU	1	2:21:17 hs	47 %

Tabla 6.5: Ejecuciones del WRF realizadas para evaluar las aceleraciones obtenidas. Se exhiben aceleraciones individualmente por rutina y generales integrando todas las rutinas portadas.

Bibliografía

[1] A. Gutiérrez, G. Cazes y J. Cataldo. Aplicación del modelo WRF-ARW a la predicción de la generación de energía eléctrica en parques eólicos. *Encuentro de Potencia, Instrumentación y Medidas de Uruguay*, 2010.

[2] A. Gutiérrez y G. Cazes. Pronósticos Numéricos Operativos en Uruguay. `http://www.fing.edu.uy/cluster/eolica/`. Último acceso: 30/04/2013.

[3] American Meteorological Society. Meteorology Glossary. `http://glossary.ametsoc.org/wiki/Model_output_statistics`. Último acceso: 30/04/2013.

[4] American Meteorological Society. Meteorology Glossary. `http://glossary.ametsoc.org/wiki/Advection`. Último acceso: 30/04/2013.

[5] American Meteorological Society. Meteorology Glossary. `http://glossary.ametsoc.org/wiki/Pressure_gradients`. Último acceso: 30/04/2013.

[6] American Meteorological Society. Meteorology Glossary. `http://glossary.ametsoc.org/wiki/Coriolis_effect`. Último acceso: 30/04/2013.

[7] American Meteorological Society. Meteorology Glossary. `http://glossary.ametsoc.org/wiki/Buoyancy`. Último acceso: 30/04/2013.

[8] American Meteorological Society. Meteorology Glossary. `http://glossary.ametsoc.org/wiki/Friction_velocity`. Último acceso: 30/04/2013.

[9] B. Barney, Lawrence Livermore National Laboratory. OMP Tutorial. `https://computing.llnl.gov/tutorials/openMP/`. Último acceso: 30/04/2013.

[10] C. Johns and D. Brokenshire. Introduction to the cell broadband engine architecture. *IBM Journal of Research and Development*, 51(5):503–519, 2007.

[11] CEFIR. Atlas de energías renovables del Mercosur. `http://cefir.org.uy/atlas/`. Último acceso: 30/04/2013.

[12] D. Kirk and W. Hwu. *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.

[13] Directorio de la Agencia Nacional de Investigación e Innovación. Fondo Sectorial de Promoción de Investigación, Desarrollo e Innovación en el Área de Energía - Reglamento Operativo. 2011. Disponible online: `http://www.anii.org.uy/web/static/DOC.INS_.043_Reglamento_Operativo_-_Fondo_Sectorial_de_Energia.pdf`.

[14] G. Ruetsch, E. Phillips and M. Fatica. GPU Acceleration of the Long-Wave Rapid Radiative Transfer Model in WRF using CUDA Fortran. 2010. Disponible online: `http://www.pgroup.com/lit/articles/nvidia_paper_rrtm.pdf`.

[15] GCC Team. The GNU Fortran Compiler. http://gcc.gnu.org/onlinedocs/gfortran/Code-Gen-Options.html. Último acceso: 30/04/2013. 2.9 Options for code generation conventions.

[16] Intel Corporation. Intel Software. http://software.intel.com/en-us/forums/topic/271008. Último acceso: 30/04/2013. Forums. Multithreaded Fortran application - data race. Threads seem to share local variables.

[17] J. C. Linford, J. Michalakes, M. Vachharajani and A. Sandu. Multi-core acceleration of chemical kinetics for simulation and prediction. *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 7:1–11, 2009.

[18] J. Delgado, G. Gazolla, E. Clua and S. M. Sadjadi. A Case Study on Porting Scientific Applications to GPU/CUDA. *Journal of Computational Interdisciplinary Sciences*, 2:3–11, 2011.

[19] J. Michalakes and M. Vachharajani. GPU Acceleration of Numerical Weather Prediction. http://www.mmm.ucar.edu/wrf/WG2/GPU/. Último acceso: 30/04/2013.

[20] J. Michalakes and M. Vachharajani. GPU Acceleration of Numerical Weather Prediction. *Parallel Processing Letters*, 18(4):531–548, 2008.

[21] J. Michalakes and M. Vachharajani. GPU Acceleration of Scalar Advection. 2009. Disponible online: http://www.mmm.ucar.edu/wrf/WG2/GPU/Scalar_Advect.htm.

[22] J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock and W. Wang. The Weather Research and Forecast Model: Software architecture and performance. *11th ECMWF Workshop on HPC in Meteorology*, pages 156–168, 2004.

[23] J. Mielikainen, B. Huang, H.A. Huang and M.D. Goldberg. GPU Acceleration of the Updated Goddard Shortwave Radiation Scheme in the Weather Research and Forecasting (WRF) Model. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 5(2):555–562, 2012.

[24] J. Mielikainen, B. Huang, H.A. Huang and M.D. Goldberg. GPU Implementation of Stony Brook University 5-Class Cloud Microphysics Scheme in the WRF. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 5(2):625–633, 2012.

[25] J. Mielikainen, B. Huang, H.A. Huang and M.D. Goldberg. Improved GPU/CUDA Based Parallel Weather and Research Forecast (WRF) Single Moment 5-Class (WSM5) Cloud Microphysics. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 5(4):1256–1265, 2012.

[26] J. Mielikainen, B. Huang, H.A. Huang and M.D. Goldberg. Compute Unified Device Architecture (CUDA)-based Parallelization of WRF Kessler Cloud Microphysics Scheme. *Computers & Geosciences*, 5(2):292–299, 2013.

[27] J. Osier and B. Baccala. GNU gprof Manual, 1993. Disponible online: ftp://ftp.gnu.org/pub/old-gnu/Manuals/gprof/html_chapter/gprof_1.html.

[28] J. Tompson and K. Schlachter. An Introduction to the OpenCL Programming Model. 2012. Disponible online: http://cs.nyu.edu/~lerner/spring12/Preso07-OpenCL.pdf.

[29] J. Wang, B. Huang, H.A. Huang and M.D. Goldberg. GPU Acceleration of the WRF Purdue Lin Cloud Microphysics Scheme. *Proc. SPIE 8183, High-Performance Computing in Remote Sensing*, 2011.

[30] J. Wang, B. Huang, H.A. Huang and M.D. Goldberg. Parallel Computation of the Weather Research and Forecast (WRF) WDM5 Cloud Microphysics on a Many-Core GPU. *IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1032–1037, 2011.

[31] M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, chapter 14, pages 244–263. Springer Berlin / Heidelberg, 2010.

[32] M. Boyer. CUDA Kernel Overhead. `http://www.cs.virginia.edu/~mwb7w/cuda_support/kernel_overhead.html`. Último acceso: 30/04/2013.

[33] NOAA/National Weather Service. National Weather Service - NCEP Central Operations - NOAA. `http://www.nco.ncep.noaa.gov`. Último acceso: 30/04/2013.

[34] NVIDIA Corporation. NVIDIA GeForce 8800 GPU Architecture Overview. Technical report, 2006. Disponible online: `http://www.nvidia.com/object/IO_37100.html`.

[35] NVIDIA Corporation. NVIDIA GeForce GTX 200 GPU Architecture Overview. Technical report, 2008. Disponible online: `http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf`.

[36] NVIDIA Corporation. Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Technical report, 2009. Disponible online: `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[37] NVIDIA Corporation. CUBLAS Library. Technical report, 2012. Disponible online: `http://docs.nvidia.com/cuda/pdf/CUDA_CUBLAS_Users_Guide.pdf`.

[38] NVIDIA Corporation. CUDA C Programming Guide. Technical report, 2012. Disponible online: `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`.

[39] NVIDIA Corporation. CUFFT Library. Technical report, 2012. Disponible online: `http://docs.nvidia.com/cuda/pdf/CUDA_CUFFT_Users_Guide.pdf`.

[40] NVIDIA Corporation. CUSPARSE Library. Technical report, 2012. Disponible online: `http://docs.nvidia.com/cuda/pdf/CUDA_CUSPARSE_Users_Guide.pdf`.

[41] NVIDIA Corporation. Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Technical report, 2012. Disponible online: `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`.

[42] OMM. Organización Meteorológica Mundial. `http://www.wmo.int/`. Último acceso: 30/04/2013.

[43] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini. An Empirical Study of Hyper-Threading in High Performance Computing Clusters. *Linux HPC Revolution*, 2002.

[44] TEMPUS. Programa de Energía Eólica en Uruguay. http://www.energiaeolica.gub.uy/. Último acceso: 30/04/2013.

[45] The Portland Group. PGI CUDA Fortran Compiler. http://www.pgroup.com/resources/cudafortran.htm. Último acceso: 30/04/2013.

[46] Valgrind Developers. Valgrind User Manual, 2012. Disponible online: http://valgrind.org/docs/manual/manual.html.

[47] W. Skamarok, J. Klemp, J. Dudhia, D. Gill, D. Barker, X. Huang, W. Wang and J. Powers. A Description of the Advanced Research WRF Version 3. Technical report, 2008. Disponible online: http://www.mmm.ucar.edu/wrf/users/docs/arw_v3.pdf.

[48] World Meteorological Organization. Introduction to GRIB Edition1 and GRIB Edition 2. 2003. Disponible online: http://www.wmo.int/pages/prog/www/WMOCodes/Guides/GRIB/Introduction_GRIB1-GRIB2.pdf.

Apéndice

Proceedings of the 13th International Conference
on Computational and Mathematical Methods
in Science and Engineering, CMMSE 2013
24–27 June, 2013.

GPU Acceleration of a Tool for Wind Power Forecasting

Marcel Burdiat[1], José Ignacio Hagopian[1], Juan Pablo Silva[1], Ernesto Dufrechou[1], Alejandro Gutiérrez[2], Martín Pedemonte[1], Gabriel Cazes[2] and Pablo Ezzatti[1]

[1] *Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay*

[2] *Instituto de Mecánica de los Fluidos e Ingeniería Ambiental, Facultad de Ingeniería, Universidad de la República, Uruguay*

emails: mburdiat@gmail.com, jsign.uy@hotmail.com, silvaljp01@hotmail.com, edufrechou@fing.edu.uy, aguti@fing.edu.uy, mpedemon@fing.edu.uy, agcm@fing.edu.uy, pezzatti@fing.edu.uy

Abstract

Uruguay is currently undergoing a gradual process of inclusion of wind energy in its matrix of electric power generation. In this context, a computational tool for wind power forecasting has been developed in order to predict the electrical power that will be injected into the electrical grid. The tool is based on the Weather Research and Forecasting (WRF) numerical model, which is the computational bottleneck of the application. For this reason, and in line with several successful efforts of other researchers, this paper presents advances in porting the WRF to GPU. In particular, we present and study the implementation of `sintb` and `bdy_interp1` routines on GPU. The results obtained on a Nvidia's GTX 480 GPU show speedup values of up to 10× when compared with the sequential WRF and almost 5× when compared with the four-threaded WRF. This improvement impacts in a 10% reduction in the total runtime of the WRF.
Key words: Wind power, WRF, GPU, `sintb` *routine,* `bdy_interp1` *routine.*

1 Introduction

Although windmills have been used to generate electricity in a domestic scale, mainly in rural properties, they have not been considered in Uruguay as an alternative energy source until recently. The Wind Power Program (WPP) [3], established in 2007, aims to significantly

 ISBN: 978-84-616-2723-3

increase the generation of electricity from wind energy encouraging the installation of large-scale wind farms in different parts of the country. For this reason, several projects are under development, and it is estimated that the wind energy penetration factor will be more than 20% in 2018 [2]. Under the WPP, UTE, the public utility responsible for the generation and transfer of electrical energy, installed the wind farm complex *Ing. Emanuele Cambilargiu*. After three years of continuous operation, it produces an average of 40% of its capacity.

Wind energy has different features compared with other sources of energy.Wind energy is not easy to be stored as potential energy, so the kinetic energy of the air flow is transformed into rotational mechanical energy which is used for electric power generation that has to be injected into the electrical grid right away. Moreover, wind energy has a large variability due to the nature of its origin. For this reason, and due to the importance that wind power will have for electric power generation in Uruguay, it is necessary to have computational tools to forecast the generated power. These tools would allow to take decisions in the management of the electrical grid in order to meet the energy demand.

With this goal, a computational tool for wind power forecasting of the UTE's wind farm complex was developed. The forecasting process runs four times a day and involves the use of the WRF numerical model [8, 16], which is the computational bottleneck of the whole process. The process generates a low resolution forecast for the whole country and a high resolution forecast where the wind farm complex is located, which are publicly available on [4]. This process involves a runtime of approximately two hours on a server with eight cores. This runtime is considered high, mainly because it will become necessary could be desirable to increase the number of wind farms that are predicted.

In recent years, the execution time of several numerical models have been significantly reduced through porting the numerical model computation, either partially or fully, to Graphics Processing Units (GPUs) [6]. Some of the most relevant numerical models that have been successfully ported to GPU can be consulted on [17].

In this paper we study the implementation of `sintb` and `bdy_interp1` routines on GPU in order to accelerate the tool for wind power forecasting. Our experiments show that just porting these routines to GPU is possible to reduce a 10% the total runtime of the WRF. The rest of the paper is structured as follows. In Section 2, we describe the tool for wind power forecasting and the WRF. Then, in Section 3, we review the related works. In Section 4, we introduce the methodology followed for porting routines from CPU to GPU and also describe the implementation of the routines on GPU. In Section 5 we present experimental results and, finally, we discuss some conclusions and future work in Section 6.

2 Tool for Wind Power Forecasting

The forecasting process performed by the computational tool involves several steps. In the first place, the tool obtains public global meteorological forecasts generated by the

 ISBN: 978-84-616-2723-3

NOAA/National Weather Service using the general circulation model of the atmosphere Global Forecast System. These forecasts are generated four times a day, at 00:00, 06:00, 12:00 and 18:00 GMT and have a grid resolution of $1^\circ \times 1^\circ$ (approximately 100km×100km). The WRF is executed locally in our server and uses this data, as well as other parameters as inputs to generate a low resolution forecast for the whole country and one high resolution forecast for the region of the wind farm complex. A forecast of the wind energy fed into the grid by each wind turbine is produced using the predictions of the wind speed and direction at the height of each turbine shaft generated by the WRF, and corrections of these predictions through model output statistics. These results are published automatically in 6-hour intervals [4], having four daily forecasts that predict the outcome for the next 48 hours.

The computationally most expensive step of the whole process corresponds to the WRF execution, which requires approximately two hours on a server with eight cores. Therefore, the performance of the WRF should be tackled in order to increase the performance of the whole tool. A description of the WRF is presented below.

2.1 Weather Research and Forecasting Model (WRF)

The WRF is a numerical model for weather prediction and an atmospheric simulation system for research and operational applications [8]. The WRF model is Eulerian (uses a fixed coordinate system with respect to the earth), non-hydrostatic (includes explicit equations to calculate the pressure and the gravitational force on the vertical axis) and compressible (considers density variations suffered by the various fluids involved). The WRF code is written in Fortran and C, and it currently has about half a million lines of code (version 3.4.1). The WRF execution partitions the domain into rectangular patches that can be assigned to different processors. These patches are subdivided into tiles that can be executed on different threads. Usually, patches need data from other neighboring patches, so in the border tiles it is necessary to propagate changes between different patches.

The tool for wind power forecasting uses the ARW ("The Advanced Research WRF") core of the WRF, which was primarily designed for research purposes, but it is also used for weather prediction. The WRF generates forecasts of W-E and N-S components that are used to calculate the magnitude of the velocity in m/s and the direction in degrees. To that end, four nested grid levels (30km×30km, 10km×10km, 3.3km×3.3km and 1.1km×1.1km) are generated, where levels of grid with a higher resolution assimilate the information generated by the model in the levels of grid with a lower resolution. The values predicted by the tool have been confirmed with real speed and direction data measured by UTE using anemometers and wind vanes installed in different parts of the country. It is noteworthy that WRF is executed in parallel in eight cores using the shared memory parallelism configuration, which uses the OpenMP API for parallelization.

 ISBN: 978-84-616-2723-3

3 Related work

This section is centered on reviewing previous efforts on porting WRF routines to GPU. The website [10] collects some of the pioneering works on this subject.

In the seminal paper on this subject, Michalakes and Vachharajani [9] address the acceleration of WRF Single Moment 5 Cloud Microphysics (WSM5) module on CUDA C. Although WSM5 only represents 0.4% of the total WRF code, it usually takes up to a quarter of the total processing time on a single core. Each GPU-thread computes the calculations corresponding to a point of the grid, since the calculations of the vertical column of each grid point are independent of other grid points. The authors evaluated their implementation in a domain with a grid resolution of 71×58 points and 27 vertical levels, using a NVidia GeForce 8800 GTX connected to an Intel Pentium-D at 2.8GHz. The numerical results showed slight differences between the GPU and CPU versions, but the visualization outputs of both versions are indistinguishable. The GPU implementation runs 17× faster than the single-threaded CPU version, including the time of the transferences between the host and the device. From this result the authors estimate that it translates in a 1.25× reduction in the total application runtime.

Later, the same authors [11] ported the WRF Fifth Order Positive Definite Tracer Advection. In this case, it is not possible to make a dimensional division of the problem, as in the previous work. Since the number of floating point operations per memory access is only 0.76 and a large number of tracers are usually executed, the GPU implementation overlaps the calculations of a tracer with the asynchronous data transfer required for next tracer. The authors evaluated their implementation in a domain with a grid resolution of 134×110 points, 35 vertical levels and 81 tracers, using a Tesla C1060 GPU connected to a quad-core Intel Xeon E5440. The GPU implementation runs 6.7× faster than the single-threaded CPU version.

Finally, Michalakes et al. [7] evaluated the parallelization of Regional Acid Deposition Model version 2 (RADM2) module on a multicore processor (two quad-core Intel Xeon 5400), a GPU (Tesla C1060) and a Cell Broadband Engine Architecture (CBEA) [5] device (PowerXCell 8i). RADM2 requires to run the Rosenbrock's algorithm, which involves constructing a Jacobian matrix and a LU decomposition, independently for each grid point. In the GPU implementation, some calculations have to be computed on the CPU due to the large amount of memory required, even though the fundamental steps of the algorithm are executed on the GPU. The parallel implementations were evaluated in a domain with a grid resolution of 40×40 points and 20 vertical levels. The speedup with respect to a single-threaded CPU version was 7.5× for the multicore processor, 8.5× for the GPU and 41× for the CBEA. Despite failing to achieve high speedup values for the GPU implementation, the authors note that the GPU is the cheapest platform of the three used and they also highlight the simplicity of its programming compared to CBEA.

In another line of work, Ruetsch et al. [18] ported the Long-Wave Rapid Radiative

 ISBN: 978-84-616-2723-3

Transfer Model (RRTM) module to CUDA Fortran. As in other works, the implementation exploits the independence between the different vertical columns of the domain, even achieving a higher level of independence in some cases, resulting in a finer grain parallelism. The authors evaluated their implementation in a domain with a grid resolution of 73×60 points and 27 vertical levels, using a Tesla C1060 GPU connected to a quad-core Intel Xeon E5440 at 2.83 GHz. The speedup of the GPU implementation versus a single-threaded CPU version is between 8× and 10×. The numerical results showed similar results between the GPU and CPU versions.

Huang et al. developed other relevant works on the subject, in which several modules were ported to CUDA C, with the final objective of completely porting WRF to GPU. In one of their works [12], the authors port the Goddard Shortwave Radiation module exploiting the independence between the different vertical columns of the domain, as the other authors. The GPU implementation also uses two streams in order to overlap kernel computation on the GPU with transferences between the host and the device. The authors evaluated their implementation in a domain with a grid resolution of 433×308 points and 35 vertical levels, using two GeForce GTX 590, i.e. four GPUs, connected to a six-core Intel i7 970. The speedup is 116× for the GPU implementation versus a single-threaded CPU version, including the time of the transferences between the host and the device.

Following the same idea, the authors ported the Stony Brook University 5-Class Cloud Microphysics module to GPU [13]. In this work, the authors used single precision floating point arithmetic and compiled using the CUDA flag `-use_fast_math` that uses faster but less accurate computation. Using the same scenario and the same platforms as in the previous work, the speedup value reported is 352×. Huang et al. also worked on porting the WSM5 [14], WDM5 Cloud Microphysics [19], Purdue Lin Cloud Microphysics [20] and Kessler Cloud Microphysics [15] modules to GPU using the same ideas, scenario and platforms as in the previous works, reporting speedup values of 357×, 147×, 156× and 70× respectively.

While speedups reported by Huang et al. are impressive, there are some aspects that should be taken into account in order to explain these values. In the first place, the scenario used by Huang et al. is more than 30 times larger than the ones used by other authors, which undoubtedly contributes to the improvement in the speedup. Another aspect that certainly impacts on these speedup values reported is the use of the CUDA compile flag `-use_fast_math`. Finally, an important issue is that the criterion followed by other authors to select which routines port to GPU is based on the importance of the routine for the execution of the WRF, while it is not clear the criterion used by Huang et al.

4 Improving the Performance of the WRF

Our work is focused on improving the runtime of the WRF, and in particular the modules required by the forecasting tool developed for the wind farm complex *Ing. Emanuele Cam-*

 ISBN: 978-84-616-2723-3

bilargiu. Therefore, an empirical approach is used to determine in which module concentrate our effort. To that end, we follow the guidelines suggested by Michalakes and Vachharajani [9], and Delgado et al. [1] to port applications developed in Fortran to CUDA C.

Four different steps are identified to port an application from Fortran to CUDA C: profiling, development, testing and optimization. The profiling stage aims to determine which are the modules or routines that require longer execution times. Then, in the development stage, it is recommended to port first Fortan to C, and then C to CUDA, since directly porting Fortran to CUDA can incorporate many errors due to the different characteristics of the languages involved. In the testing stage, it has to be evaluated that the results obtained by the GPU are similar to the ones obtained by the CPU, taking into account that there might be floating point rounding differences caused by the parallelism. Finally, in the optimization stage, the CUDA implementation is fine tuned in order to reduce the runtime.

The tools for profiling applications follow two completely different philosophies. On the one hand, there are tools that make changes in the source code at compile time, including extra code to obtain metrics of the application execution. On the other hand, there are tools that work like a debugger or a virtual machine that obtain the metrics in run-time using the original code. Since both approaches are complementary, we decided to profile the WRF using one tool of a kind. We use *gprof*, which follows the former approach and it is considered the de-facto profiler in academic environments since it is included in the GNU project, and *valgrind* that follows the latter approach. The reports of the single-threaded execution of WRF generated with both tools showed a great consistency in the results, despite following different philosophies. The routine with longer runtime was `sintb` that required around the 15% of the total execution time. For this reason, we focus our work in porting this routine to GPU. Even though each invocation of `sintb` does not take a large runtime, the routine is invoked more than 10 million times during the WRF execution.

4.1 sintb and bdy_interp1 Routines

To begin with, we study the feasibility of porting `sintb` routine to GPU analyzing the relationship between transference and computation time for a call to `sintb`. We found that this ratio was four to one, so we examined `bdy_interp1` routine, which is the only routine that calls `sintb`. `bdy_interp1` makes four consecutive calls to `sintb` with the same input matrices, but with different sets of positions. For this reason, a single transference is required to make the four invocations to `sintb` thus compensating the high relative computational cost of the transference. So, it was considered a better option to also port code from the `bdy_interp1` routine to the GPU instead of only porting `sintb`.

After the four `sintb` calls, the `bdy_interp1` routine makes assignments to other matrices using the results from the invocations. Both invocations and assignments are in a loop parallelized using OpenMP in the original code, but the set of positions of each invocation to `sintb` does not depend on the iteration step. Therefore, we decided to group all calls to

 ISBN: 978-84-616-2723-3

`sintb` in a single call, postponing all assignments for the end. Following that approach, the Fortran code of `sintb` was ported to C, and it was verified that the results were similar to those obtained by the original version. Then, the C code of `sintb` was ported to CUDA.

Now, the main bottlenecks were the transference of the result matrices (25% of the total runtime) and the final assignments to other matrices in `bdy_interp1` (30% of the total runtime), since they could not be run concurrently with the kernel. For this reason, we chose to migrate the final assignments in `bdy_interp1` to a second kernel in CUDA, not only to reduce the computation time of this task, but also taking benefit from the fact that all the inputs of the routine are already on the GPU. This brought a couple of other improvements. On the one hand, the final results of `bdy_interp1` are nearly half of the size of the intermediate results produced by the invocation to `sintb`, so the transference time from GPU to CPU is significantly reduced. On the other hand, the matrices where the final results are updated can be copied from the GPU to the CPU asynchronously and concurrently with the execution of the first kernel, which hides this transference time. The pseudocode of the host side of the CUDA implementation outlined above is presented in Algorithm 1.

Algorithm 1 `bdy_interp1` Host Side Pseudocode

```
1: synchronous transferences required by sintb, host to device, stream 1
2: asynchronous transferences to be updated with final results, host to device, stream 2
3: invoke sintb kernel through stream 1
4: synchronize stream 2                ▷ Blocks host until all calls in stream 2 are completed
5: invoke second kernel through stream 1                          ▷ Updates the final results
6: synchronous transferences of final results, device to host, stream 1
```

The memory accesses of the GPU implementation are coalesced and the memory spaces allocated on the GPU are reused in order to reduce calls to malloc and free. Finally, we performed an analysis to find the best balance between the number of registers and shared memory used per block. The best configuration consists in using only 24KB of shared memory per block, allowing to execute up to two concurrent blocks on each multiprocessor.

5 Experimental Results

This section describes first the scenario used for the experimental study and the execution platform. Then, the results obtained are presented and analyzed.

5.1 Test Instance

The test instance consists in real information of the 07/15/2012 for a 12 hours climate forecast of the wind farm complex *Ing. Emanuele Cambilargiu*. The test instance is con-

 ISBN: 978-84-616-2723-3

sidered representative of a normal day of operation of the park. The numerical model uses four nested grid levels of 30km×30km, 10km×10km, 3.3km×3.3km and 1.1km×1.1km that assimilate information from the lower resolution to the higher resolution levels of the grid. The 30km×30km domain has a grid resolution of 74×61 points and 54 vertical levels, while the others domain has a grid resolution of 40×40 points and 54 vertical levels.

5.2 Test Environment

The execution platform for the CPU runs is a PC with a dual core Intel Core i3-2100 processor at 3.10 Ghz with 8 GB using the Fedora 15 Linux operating system. It should be noted that the processor supports hyper-threading. The CPU versions were compiled using the `-O3` flag. The execution platform for the GPU runs is an Nvidia's GeForce GTX 480 (480 CUDA cores, Fermi architecture, Compute Capability 2.0) connected to the CPU platform. The GPU versions were compiled using the Nvidia's CUDA compiler 4.1 version with the `-O3` flag. All the reported total runtime of the GPU executions always include the transference time of data between CPU and GPU.

5.3 Experimental Analysis

We begin our analysis with the parallelization of the WRF using OpenMP. Table 1 presents the experimental results regarding the performance obtained. The table includes the execution time of the single-threaded, two-threaded and four-threaded (using hyper-threading) WRF, as well as the speedup values of the parallel executions. The speedup value obtained using two threads is not close to linear; this shows that the parallelization of the WRF is not a trivial task and that an adequate load balance has to be achieved in order to benefit from additional processing units. The execution time with two and four threads allows us to affirm that in this case the use of hyper-threading is justified.

Table 1: OpenMP Parallelization of WRF

Single Thread	Two Threads		Four Threads	
Runtime (mins)	Runtime (mins)	Speedup	Runtime (mins)	Speedup
226.70	126.77	1.79	103.38	2.19

Table 2 presents the execution time of the single-threaded and four-threaded (using hyper-threading) of `bdy_interp1` routine, as well as the speedup value of the parallel version. The runtimes reported are the average of all the executions of the routine (the routine is executed more than 2.5 million times). The speedup value obtained with four threads is slightly worse than for the entire application which may indicate that the routine has further difficulty to scale with a multithreaded parallelization.

We continue our analysis with the parallelization of the WRF using a GPU. In the first place it should be noted that there are no significant differences regarding the numerical

 ISBN: 978-84-616-2723-3

Table 2: OpenMP Parallelization of `bdy_interp1`

Single Thread	Four Threads	
Runtime (ms)	Runtime (ms)	Speedup
24.00	11.50	2.09

accuracy between the results obtained by the GPU and the CPU implementation. There are several alternatives to evaluate the performance of the GPU implementation. The two key aspects that influence the definition of speedup that can be considered are how many threads run on the CPU implementation taken as a reference and whether the transference time is included in the runtime of the GPU implementation or not. Four different definitions of speedup arise from the combination of both aspects: parallel CPU run and including the transference time (Eq. 1), parallel CPU run and not including the transference time (Eq. 2), single-threaded CPU run and including the transference time (Eq. 3) and single-threaded CPU run and not including the transference time (Eq. 4).

$$\text{Speedup}_I = \frac{\text{Runtime of 4-threaded } \texttt{bdy\_interp1}}{\text{GPU total runtime of } \texttt{bdy\_interp1}} \quad (1) \qquad \text{Speedup}_{II} = \frac{\text{Runtime of 4-threaded } \texttt{bdy\_interp1}}{\text{GPU total runtime of } \texttt{bdy\_interp1}} \quad (2)$$

$$\text{Speedup}_{III} = \frac{\text{Runtime of 1-threaded } \texttt{bdy\_interp1}}{\text{GPU total runtime of } \texttt{bdy\_interp1}} \quad (3) \qquad \text{Speedup}_{IV} = \frac{\text{Runtime of 1-threaded } \texttt{bdy\_interp1}}{\text{GPU total runtime of } \texttt{bdy\_interp1}} \quad (4)$$

All the previous works reviewed in the related work section use Speedup_{III} to evaluate the performance of the GPU implementation. Huang et al. [12, 13, 14, 15, 19, 20] also report Speedup_{IV} since they claim that when the WRF is fully ported to GPU it would not be necessary to transfer the data. We believe that the speedup values, which are calculated against the four-threaded CPU execution (Speedup_I and Speedup_{II}), are fairer to evaluate the performance of the GPU implementation because it is a more realistic execution scenario since it is a little artificial to limit the computing capacity of the CPU for the evaluation. Anyway, we include Speedup_{III} and Speedup_{IV} in our analysis in order to be able to compare our results with other works.

Table 3 presents the runtime of the GPU implementation of the `bdy_interp1` routine with the transference and computation time disaggregated, as well as the four speedup values described above. To begin with, the transference time represents 20% of the total execution time so the increase in the speedup when the transference is not considered is only 20%. Although it is a small percentage of the total time, it can be explored the alternative of using asynchronous transferences instead of synchronous ones, in order to overlap them with computation on the GPU or the CPU, thus reducing the time involved in the transferences. In the second place, the Speedup_I value obtained when compared

 ISBN: 978-84-616-2723-3

with the four-threaded execution of the WRF is 4.89. An almost five times reduction in the runtime of the routine can be considered a good result, since the improvement is accomplished versus an already parallel implementation that was developed by the staff of developers of the WRF. Finally, the Speedup$_{III}$ value obtained is 10.21 that is comparable to the values reported by Michalakes et al. and Ruetsch et al. We could not reach similar values to the ones reported by Huang et al., but the scenario we used is significantly smaller and fast math was not used due to the characteristics of the routine that we ported to GPU.

Table 3: GPU Parallelization of `bdy_interp1`

Runtime (ms)			Speedups			
Transference	Computation	Total	Speedup$_I$	Speedup$_{II}$	Speedup$_{III}$	Speedup$_{IV}$
0.45	1.90	2.35	4.89	6.05	10.21	12.63

Table 4 presents the runtime of the single-threaded and four-threaded WRF with `bdy_interp1` routine executing on the GPU, as well as the corresponding speedup values. The implementation of this routine in GPU produces a reduction of more than 10 minutes (10.26% of the total runtime) in the runtime of the four-threaded WRF and of 20 minutes (8.75% of the total runtime) in the runtime of the single-threaded WRF.

Table 4: GPU Parallelization of WRF

Single Thread + `bdy_interp1` on GPU		Four Threads + `bdy_interp1` on GPU	
Runtime (mins)	Speedup	Runtime (mins)	Speedup
206.87	1.10	92.77	1.11

6 Conclusions and Future Work

In this paper we have studied the acceleration of a tool for wind power forecasting on a GPU. To this end, we profiled the WRF model, which is the main bottleneck of the application, and determined that `sintb` is the routine with longer runtime (around 15% of the total runtime of WRF). A further analysis showed that it was a better alternative besides porting `sintb` to GPU also porting `bdy_interp1` routine. The implementation of `sintb` and `bdy_interp1` routines on Nvidia's GTX 480 GPU obtained speedup values of up to 10× and almost 5× when compared with the single-threaded and four-threaded execution on CPU, respectively. The acceleration obtained implies a 10% reduction in the total runtime of the WRF.

We identify three lines for future work. The first one is to identify additional routines to port to GPU in order to accelerate the tool for wind power forecasting. This leads to a second line of interest, which consists in executing the WRF with the two routines on the GPU for a whole forecast for the next 48 hours. Finally, we aim to use as execution platform a GPU with Kepler architecture that recently came to market.

 ISBN: 978-84-616-2723-3

Acknowledgements

G. Cazes, E. Dufrechou, P. Ezzatti, A. Gutiérrez and M. Pedemonte acknowledge support from Agencia Nacional de Investigación e Innovación (ANII), Uruguay and Administración Nacional de Usinas y Trasmisiones Eléctricas (UTE), Uruguay. E. Dufrechou, P. Ezzatti and M. Pedemonte also acknowledge support from Programa de Desarrollo de las Ciencias Básicas (PEDECIBA), Uruguay.

References

[1] J. Delgado, G. Gazolla, E. Clua and S. Sadjadi, *A Case Study on Porting Scientific Applications to GPU/CUDA*, Journal of Computational Interdisciplinary Sciences **2**, (2011) 3–11.

[2] *Dirección Nacional de Energía - Ministerio de Industria, Energía y Minería (in Spanish)*, `http://www.dne.gub.uy/`.

[3] Dirección Nacional de Energía - Ministerio de Industria, Energía y Minería, *Programa de Energía Eólica en Uruguay (in Spanish)*, `http://www.energiaeolica.gub.uy/`.

[4] A. Gutiérrez and G. Cazes, *Pronósticos Numéricos Operativos en Uruguay (in Spanish)*, `http://www.fing.edu.uy/cluster/eolica/`.

[5] C. Johns and D. Brokenshire, *Introduction to the Cell Broadband Engine Architecture*, IBM Journal of Research and Development **51(5)**, (2007) 503–519.

[6] D. Kirk and W. Hwu, *Programming Massively Parallel Processors, Second Edition: A Hands-on Approach*, Morgan Kaufmann, 2012.

[7] J. Linford, J. Michalakes, M. Vachharajani and A. Sandu, *Multi-core Acceleration of Chemical Kinetics for Simulation and Prediction*, SC '09, Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, (2009) 1–11.

[8] J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock and W. Wang, *The Weather Research and Forecast Model: Software architecture and performance*, 11th ECMWF Workshop on HPC in Meteorology, (2004) 156–168.

[9] J. Michalakes and M. Vachharajani, *GPU Acceleration of Numerical Weather Prediction*, Parallel Processing Letters **18(4)**, (2008) 531–548.

 ISBN: 978-84-616-2723-3

[10] J. Michalakes and M. Vachharajani, *GPU Acceleration of Numerical Weather Prediction*, http://www.mmm.ucar.edu/wrf/WG2/GPU/.

[11] J. Michalakes and M. Vachharajani, *GPU Acceleration of Scalar Advection*, Available online: http://www.mmm.ucar.edu/wrf/WG2/GPU/Scalar_Advect.htm.

[12] J. Mielikainen, B. Huang, H.A. Huang and M.D. Goldberg, *GPU Acceleration of the Updated Goddard Shortwave Radiation Scheme in the Weather Research and Forecasting (WRF) Model*, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing **5(2)**, (2012) 555–562.

[13] J. Mielikainen, B. Huang, H.A. Huang and M.D. Goldberg, *GPU Implementation of Stony Brook University 5-Class Cloud Microphysics Scheme in the WRF*, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing **5(2)**, (2012) 625–633.

[14] J. Mielikainen, B. Huang, H.A. Huang and M.D. Goldberg, *Improved GPU/CUDA Based Parallel Weather and Research Forecast (WRF) Single Moment 5-Class (WSM5) Cloud Microphysics*, IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing **5(4)**, (2012) 1256–1265.

[15] J. Mielikainen, B. Huang, H.A. Huang and M.D. Goldberg, *Compute Unified Device architecture (CUDA)-based Parallelization of WRF Kessler Cloud Microphysics Scheme*, Computers & Geosciences **52**, (2013) 292–299.

[16] National Center for Atmospheric Research, *Weather Research and Forecasting Model*, http://www.wrf-model.org/index.php.

[17] Nvidia, *CUDA Community Showcase*, http://www.nvidia.com/object/cuda_showcase_html.html.

[18] G. Ruetsch, E. Phillips and M. Fatica, *GPU acceleration of the Long-wave Rapid Radiative Transfer Model in WRF Using CUDA Fortran*, Available online: http://www.pgroup.com/lit/articles/nvidia_paper_rrtm.pdf.

[19] J. Wang, B. Huang, H.A. Huang and M.D. Goldberg, *Parallel Computation of the Weather Research and Forecast (WRF) WDM5 Cloud Microphysics on a Many-Core GPU*, IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS), (2011) 1032–1037.

[20] J. Wang, B. Huang, H.A. Huang and M.D. Goldberg, *GPU Acceleration of the WRF Purdue Lin Cloud Microphysics Scheme*, Proc. SPIE 8183, High-Performance Computing in Remote Sensing, (2011) 81830R.

 ISBN: 978-84-616-2723-3