Universidad Nacional de Rosario

Licenciatura en Ciencias
de la Computación

Tesina de Grado

Refinamiento Adaptivo de Código

Autor:
César Sabater

Directores:
Cédric Bastoul
Guillermo Grinblat

Legajo: S-4610/8

28 de marzo de 2017

Resumen

Las técnicas de optimización y paralelización automática son muy adecuadas para algunas clases de aplicaciones en simulación o procesamiento de señales, sin embargo normalmente no tienen en cuenta conocimiento de dominio específico ni la posibilidad de cambiar o eliminar cálculos modificando la semántica original pero manteniendo resultados "suficientemente buenos". Por otro lado, los códigos de producción en simulación y procesamiento de señales tienen capacidades adaptivas: están diseñados para computar resultados precisos solamente donde es necesario si el problema completo no es tratable o si el tiempo de cálculo debe ser corto. En este trabajo, presentamos una nueva manera de suministrar capacidades adaptivas de forma automática para códigos de cómputo intensivo. Se basa en conocimiento de dominio específico en el código de entrada suministrado por el programador a través de pragmas especiales y en técnicas de compilación poliédrica para regenerar en tiempo de ejecución un código que realiza cálculos complejos solamente donde es necesario en cada momento. Presentamos un caso de estudio en una aplicación de simulación de fluidos donde nuestra estrategia permite importantes ahorros en cálculos en la porción optimizada de la aplicación así como también mejoras significativas en el tiempo total de ejecución, manteniendo buena precisión, con un mínimo trabajo del programador.

Índice general

Capítulo 1

Introducción

1.1. Motivación y Descripción del Trabajo

Un amplio espectro de las aplicaciones de cómputo intensivo calculan resultados aproximados, siendo una de las formas más comunes el uso de aproximaciones por construcción. Esto es sumamente aplicado en códigos de simulación que están basados en modelos inherentemente imperfectos, que quieren emular con la mayor precisión posible un fenómeno o elemento del mundo real. También es utilizado en aplicaciones de procesamiento de señales que están limitadas en su precisión, por ejemplo, sensores de entrada o algoritmos de procesamiento. Otras aplicaciones también computan resultados aproximados por razones funcionales, por ejemplo para cumplir un *deadline* como el de decodificación de video en tiempo real o porque el resultado no es manejable o valioso como en la predicción tardía de terremotos. En aplicaciones de geofísica, el usuario necesita un resultado preliminar para luego llevar a cabo investigaciones más precisas. Para esas aplicaciones, los *kernels* [1] de cómputo "ideales", que serían convenientes con una cantidad infinita o inaccesible de poder computacional, son normalmente diseñados para para ajustar el algoritmo y debuguearlo. Luego, son optimizados a una versión "de producción" aprovechando posibles aproximaciones para escalar al tamaño real del problema, o para satisfacer un *deadline*. Traducir un código ideal es complejo, lleva tiempo, conduce a códigos menos mantenibles y hay que realizarlo de nuevo cuando hay un cambio grande en la estrategia inicial.

En este trabajo se presenta una nueva técnica de optimización, que llamamos *Refinamiento Adaptivo de Código(ACR)* [2] y que investiga la automatización de la conversión de *kernels* ideales a versiones que aprovechan

[1] Un *kernel* de cómputo es la función central de cálculo que apunta a resolver el propósito principal del código. Normalmente en el kernel está el algoritmo principal y tiene mucho uso, por lo que su optimización impacta significativamente en todo el programa.

[2] Las siglas remiten al nombre en inglés, *Adaptive Code Refinement*

las aproximaciones dinámicamente. Apunta a mejorar tanto la productividad del desarrollador como la calidad de la aproximación. Está inspirada en *Adaptive Mesh Refinement* [4], una técnica clásica de Análisis Numérico, que tiene la capacidad de modificar de forma dinámica una malla para lograr cálculos precisos solamente donde es necesario. Nosotros logramos ese objetivo aprovechando información de alto nivel proporcionada por el usuario, una estrategia de optimización dinámica y el estado del arte en técnicas de compilación poliédrica. El trabajo es producto del diseño y evaluación de formas para que programadores de simulación y procesamiento de señales ingresen conocimiento de dominio específico a sus códigos y formas en las que los compiladores usen esta información para generar versiones optimizadas adaptivas.

La estrategia utilizada está soportada por información de alto nivel suministrada por el usuario y un enfoque de generación de código estático-dinámico. Esta información de alto nivel describe qué valores de la ejecución se deben tener en cuenta, así como también de qué forma se debe transformar (y posiblemente relajar) las dependencias del código. También indica con qué nivel de detalle actúa la adaptividad del código. Los valores que se monitorean generalmente sirven para poder determinar las regiones del espacio de iteración que son complejas y las que son simples. Para ello, se diseñaron un conjunto de pragmas para darle al usuario medios para suministrar información para el cálculo de aproximaciones estático y dinámico. Esto permite al usuario enfocarse solamente en versiones simples "ideales" de *kernels* computacionales. Los pragmas están descriptos en la Sección 4.1.

Una vez que se obtuvo la información necesaria para identificar diferentes regiones de complejidad, ACR debe generar en tiempo de ejecución código nuevo encargado de realizar los cómputos solamente necesarios para satisfacer un nivel de precisión, ahorrando cálculos. Para realizar de forma eficiente esta tarea, fue necesaria una forma de representar matemáticamente el espacio de iteración de un kernel de computación y de qué manera se llevan a cabo los cálculos dentro de éste, así como también una forma de regenerar este código omitiendo y/o modificando cálculos. Para ello ACR se basa fuertemente en un modelo conocido como "Modelo Poliédrico" [3] para la representación de bucles anidados. Este modelo establece una representación de las iteraciones de un bucle como puntos de un látice dentro de un politopo. Utiliza transformaciones afines, y otras no-afines más generales, aplicadas sobre bucles anidados y engloba tanto la representación de programas, su transformación y la generación de nuevo código. En el Capítulo 2 se describe de forma más detallada las características del Modelo Poliédrico haciendo énfasis en sus características más utilizadas en el trabajo.

Nuestra técnica usa herramientas de compilación poliédrica para realizar estas tareas de manera dinámica y relajando las dependencias que generan

[3] en inglés, *Polyhedral Model*

cálculos innecesarios según la información ingresada a través de los pragmas. El uso de estas herramientas lo unificamos en una sola llamada Spot, que es el primer aporte de este trabajo. Esta herramienta nos permite ingresar dominios de interés asociados a computaciones alternativas o donde se quiera eliminar completamente los cálculos. Spot puede utilizarse directamente por el programador y también es utilizada por ACR para la generación de código. Esta se detalla en el Capítulo 3.

Para trasladar el conocimiento de dominio y la compilación poliédrica al plano dinámico, ACR descompone el espacio de cálculo en una cuadrícula y monitorea valores específicos en cada celda. La información necesaria para la regeneración del código está en la cuadrícula. Dependiendo de su evolución, ACR ajustará la cantidad de cálculos o algoritmo de computación que le corresponderá a la región de cómputo representada por cada celda para asegurar cálculos complejos solamente donde son necesarios. De acuerdo con este monitoreo, utilizará la compilación poliédrica para computar los espacios de iteración del código optimizado, manteniendo al máximo posible el orden de los cálculos y las dependencias, y generará una versión optimizada del código de compilación. Ésta versión será compilada dinámicamente y cuando esté lista para ser ejecutada, se verificará que la optimización todavía sirva para el estado de la simulación: si es así, se intercambiará el código en ejecución por la versión optimizada. Las tareas de monitoreo y generación de código están realizadas por *threads* dedicados a cada una para minimizar el tiempo de espera. El monitoreo continuo y la generación de código nuevo permiten un fuerte ahorro en cálculos limitando la falta de precisión. El proceso completo está detallado en el Capítulo 4.

La técnica se probó sobre una implementación de Simulación de Fluidos para Gráficos en tiempo real. Esta simula fluidos con parámetros configurables como su viscosidad y difusividad en el ambiente. Describe los fluidos a través de una cantidad fija de partículas dispuestas regularmente en el espacio con información del fluido en cada punto. Es un tipo de simulación muy adecuado para la aplicación de ACR. A partir de la cantidad de fluido en distintas zonas, ACR determina la complejidad de los cálculos a realizar.

Para evaluar la eficacia de la técnica comparamos el código optimizado utilizando ACR con el código original en aspectos de tiempo, ahorro de cálculos y precisión. Los resultados en estos aspectos también se compararon con una alternativa estática de optimización que imita a ACR pero sin generar código dinámicamente, es decir, ahorrando la misma cantidad de cálculos pero con código compilado antes de la ejecución. La implementación de ACR en la simulación y también los resultados de las pruebas se describen en el Capítulo 5.

Como resultado de esto, se muestra evidencia empírica de que ACR permite ahorros en tiempo y cálculos manteniendo precisión con esfuerzos mínimos, incluso comparándola con una versión optimizada a mano.

1.2. Trabajo Relacionado

1.2.1. Modelo Poliédrico

Trabajos sobre la importancia del *scheduling* y *allocation* tienen comienzo a finales de la década del 60 en publicaciones como la de R. Karp et al [22] en el cual se muestra el impacto y necesidad de tener en cuenta estos aspectos para la paralelización y localidad de datos en almacenamientos temporarios y de alta velocidad en las llamadas *ecuaciones de recurrencia uniforme*. Estas aparecen generalmente en el cómputo de aproximaciones de diferencia finita de *Ecuaciones Diferenciales Parciales*.

Muchos trabajos fundacionales en el análisis de dependencias fueron realizados a fines de la década del 80 y en la década del 90. Trabajos como los realizados por Feautrier [12] o Pugh [29] presentan métodos de resolución de problemas de programación entera muy comunes en el análisis de programas y en el análisis de dependencias en bucles. Otros trabajos también hacen análisis de dependencias y utilizan *Parametric Integer Programming* como componente básica de sus cálculos [13,15]. También importantes contribuciones para abordar el problema de generación de código se tuvieron en cuenta en aquel momento [17,23]. También fueron hechas en este período otras contribuciones en el estudio de transformaciones de programa para operaciones que resultaron clave en un futuro como *Loop Tiling* [14, 20, 40, 41].

Luego de que se definiera el área, esta se expandió en distintos aspectos, como por ejemplo en la búsqueda de soluciones a los problemas de escalabilidad que todavía se mantenían en el análisis de dependencias [37] y en especial en generación de código [2,30,36]. Estudios de la composicionalidad de transformaciones de programas [11, 16] aplicados a código representativo de aplicaciones reales como los benchmarks SPEC CPU 2000 fp [4], hicieron el modelo útil para algunas optimizaciones en compiladores de producción [27]. Luego se diseñó y desarrolló Pluto [5,6], una herramienta que integra y realiza de forma automática todas las fases de la compilación poliédrica, es escalable y se enfoca en optimizaciones para la paralelización y localidad de datos. *Loop Tiling* es su transformación clave. Otras herramientas automáticas para la transformación poliédrica no poseían una función de estimación de costo realista para las transformaciones y no trabajaban los aspectos de paralelización y localidad de datos en forma conjunta.

Actualmente, el estado del arte en técnicas de compilación para la optimización y paralelización está basado fuertemente en el modelo poliédrico para manipular *kernels* de computación intensiva y reestructurarlos agresivamente a nivel de iteraciones [3, 6, 28] y también, para realizar transformaciones en programas de forma dinámica para los casos donde no es posible detectar código optimizable en tiempo de compilación [21]. Sin embargo, las técnicas actuales hacen análisis exactos de dependencias y aseguran por

[4] `https://www.spec.org/cpu2000/CFP2000/`

construcción que la optimización preserva la semántica original del programa. Aunque existan semánticas mas relajadas, por ejemplo, para soportar conmutatividad y permitir vectorización [24], todas las iteraciones aún deben ser ejecutadas.

1.2.2. Técnicas de Compilación para Aproximaciones

La idea de relajar dependencias y omitir cálculos para mejorar performance a cambio de precisión fue estudiada en el pasado desde varios enfoques. *Loop perforation* [19,33] es una técnica estática que elimina iteraciones completas de un bucle, seleccionadas de acuerdo a una fase de entrenamiento. Distintamente a *loop perforation*, ACR es un enfoque dinámico que puede eliminar solo las partes elegidas de la ejecución de un bucle y está diseñado para producir resultados más precisos con la ayuda del usuario final. Nuestro trabajo comparte con Green [1] el concepto de información de alto nivel, incluyendo implementaciones alternativas suministradas a través de pragmas. Sin embargo, Green está basado en un entrenamiento previo para elegir el código final mientras que ACR está continuamente recalculando el mejor código. EnerJ [32] usa el sistema de tipos para especificar variables aproximadas y así ahorrar energía. SAGE [31] es una técnica orientada a GPU que omite o simplifica procesamiento con respecto a la performance mientras que nuestra técnica se enfoca principalmente en la precisión. HELIX-UP [9] ignora dependencias para permitir paralelismo, que es complementario a nuestro enfoque.

1.2.3. Técnicas de Cálculo Aproximado

ACR estuvo inspirado en técnicas de análisis numérico, en particular *Adaptive Mesh Refinement* [4] que puede mantener la consistencia de una solución para un error acotado, en el menor tiempo posible, en problemas de simulación. Se pueden nombrar algunos ejemplos de variaciones de estas mallas adaptivas. Muller [26] extiende el uso de análisis multiescala para adaptación de cuadrículas para incorporar una variación local del *timestep*. En el caso de Brix [8], utiliza *space filling curves* para transformar datos multidimensionales de la cuadrícula a datos unidimensionales fáciles de manipular y distribuir para una mejor paralelización de los métodos de adaptación multiescala.

Nuestra técnica ACR, aunque inspirada en AMR, posee poco más en común con ella que ser basada en cuadrículas. La manipulación del espacio de iteración es distinto ya que en AMR éste es modificado continuamente: la estructura de iteración subyacente es una cuadrícula jerárquica que incorpora y pierde ramas de su árbol durante la ejecución. Por el contrario, ACR mantiene la resolución del espacio original de iteración, incluso cuando omite cálculos, mientras que el código generado cambia de forma dinámica.

Capítulo 2

Modelo Poliédrico

Nuestra técnica, desde el análisis del código original hasta la generación de versiones optimizadas, está fuertemente basada en la compilación poliédrica. Es la herramienta que nos permite obtener código altamente optimizado partiendo de una cuadrícula que describe distintas estrategias de cálculo en cada una de sus celdas. Su trabajo engloba el análisis de código original, el cómputo de las nuevas regiones de interés, la modificación del modelo en base a estas regiones y la generación de código optimizado. Hace que el código generado recorra el espacio de iteración manteniendo el orden de las dependencias lo más posible y sin switchs en los bucles internos para determinar los límites de cada región, que son costosos.

El modelo poliédrico es un modelo matemático-computacional para la optimización y paralelización automática de programas. Afirma que un statement que se encuentra dentro de un bucle anidado no puede ser analizado como un solo objeto sino como un conjunto de instancias tan grande como el número de iteraciones en las que se ejecuta ese statement. Por esta razón, una de sus características clave es que describe el conjunto de valores que toman los iteradores en bucles anidados, es decir el espacio que de iteración, de forma geométrica a través de poliedros como se puede ver en la Figura 2.1. Las ecuaciones de estos poliedros están dadas por las cotas de los bucles. A la par, se modela el ordenamiento y lugar de estas instancias para su ejecución, a través de *fechas lógicas*. También los accesos a datos son modelados matemáticamente. Lo que tienen en común estos elementos de representación de programas es que para todos ellos se utilizan *relaciones poliédricas* para expresarlos matemáticamente. Estas se verán en forma general al final de este capítulo.

Una vez que el programa está en la representación poliédrica, puede alterarse el conjunto de instancias ejecutadas, su orden y lugar, a partir de transformaciones afines, como por ejemplo la de la Figura 2.2 hasta cambiar radicalmente el programa.

Como consecuencia de esta representación, el dominio de aplicación del

modelo poliédrico corresponde a códigos tales que los posibles valores de cada iterador de bucle puede ser modelados por un conjunto de restricciones afines en función de ese mismo iterador, iteradores de bucles externos y parámetros fijos. Gran parte de las aplicaciones se encuentran en este dominio, ya que generalmente los *kernels* pueden ser expresados fácilmente con funciones afines.

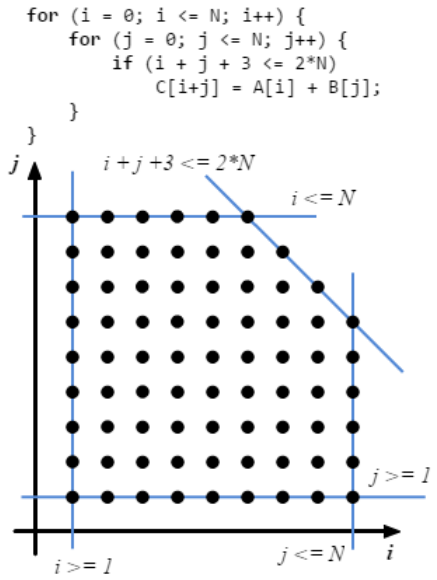


Figura 2.1: Dominio de Iteración (abajo) representado por el modelo poliédrico del ejemplo de bucle anidado (arriba).

Como lo explicamos en la Sección 1.2.1, el problema de generación código fue muy estudiado. Éste consiste en encontrar un código (preferentemente eficiente) que visite cada punto entero de cada poliedro una sola vez, respetando el orden lexicográfico de algunas dimensiones. En la actualidad existen técnicas de generación de código eficiente con buen balance entre sobrecarga en control de flujo y tamaño del código [30]. Existen herramientas de análisis, que nos permiten traducir el código a un modelo poliédrico, también para transformar distintas representaciones con la posibilidad de reestructurarlas de forma agresiva, y también otras que nos permiten la generación de código optimizado que recorra todos los puntos del poliedro respetando de forma lexicográfica algunas dimensiones. En la Sección 2.4 nombramos las herramientas más conocidas disponibles para la compilación poliédrica.

2.1. Representación Poliédrica de Código

En esta sección explicamos qué elementos matemáticos se utilizan para la representación de un programa. Primero explicamos los Dominios de

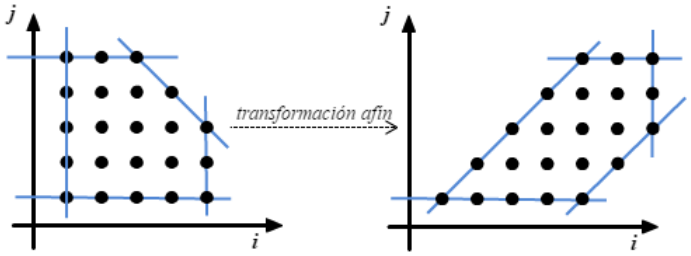


Figura 2.2: Un ejemplo de una posible transformación afín de un programa.

Iteración, que describen el conjunto de ejecuciones de un statement en un bucle. Luego, los mapeos de *scheduling* y *posicionamiento*, que describen el orden y qué procesador (para ejecuciones en paralelo) se le asigna a cada una de estas instancias y finalmente, sus *relaciones de acceso* a los datos.

2.1.1. Dominios de Iteración

En el modelo poliédrico, a cada ejecución de un statement que se encuentra dentro de un bucle se la trata como una entidad individual y se la denomina instancia de statement. Estas instancias quedan determinadas por el valor de los iteradores en el momento de su ejecución. Así, si tenemos un statement S dentro de un bucle anidado en donde iteran las variables i y j, la instancia del statement para el valor (i', j') la podemos representar como $S(i', j')$. Ahora S es una función cuyo dominio es el conjunto de vectores $\binom{i}{j}$ que recorran todos los valores de iteración. Un ejemplo de esto es el statement S_1 de la Figura 2.3.

```
1       for (i = 0; i < 2 * N - 1; i++)
2  S1:          z[i] = 0;
3
4       for (i = 0; i < N; i++)
5            for (j = 0; j < N; j++)
6  S2:          z[i+j] = x[i] + y[j];
```

Figura 2.3: Kernel de Multiplicación de Polinomios.

Para describir el conjunto de estos valores en un código, una forma muy conveniente de hacerlo es como un conjunto de puntos enteros en un espacio de tantas dimensiones como componentes tiene el vector de iteración, y cuyas cotas están dadas en base a las restricciones de los bucles en el recorrido de sus iteradores. Si los iteradores y sus restricciones son funciones afines, esto se puede representar como una lista de inecuaciones afines sobre los iteradores, es decir un poliedro (específicamente, un politopo) de tantas dimensiones

como el vector de iteración. Este poliedro puede ser representado como una unión de inecuaciones matriciales de la forma $A\vec{p} \geq 0$, donde A es la matriz de coeficientes, $\vec{p}$ es un vector que contiene los iteradores, parámetros fijos y un 1 para constantes. Para expresar un poliedro correspondiente a un dominio de iteración, se utilizan uniones del siguiente tipo de relaciones afines:

$$\mathcal{D}_S(\vec{p}) = \left\{ () \to \vec{l_S} \in \mathbb{Z}^{dim(\vec{l_S})} \middle| [D_s] \begin{pmatrix} \vec{l_S} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\}.$$

Por ejemplo, las relaciones poliédricas que definen los dominios de S_1 y S_2 para el kernel de multiplicación de polinomios son:

$$\mathcal{D}_{S_1}(N) = \left\{ () \to (i) \in \mathbb{Z} \middle| \begin{bmatrix} 1 & 0 & 0 \\ -1 & 2 & -2 \end{bmatrix} \begin{pmatrix} i \\ N \\ 1 \end{pmatrix} \geq \vec{0} \right\}$$

y

$$\mathcal{D}_{S_2}(N) = \left\{ () \to \begin{pmatrix} i \\ j \end{pmatrix} \in \mathbb{Z}^2 \middle| \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq \vec{0} \right\}.$$

En la Figura 2.4 se puede ver el conjunto de instancias de S_2.

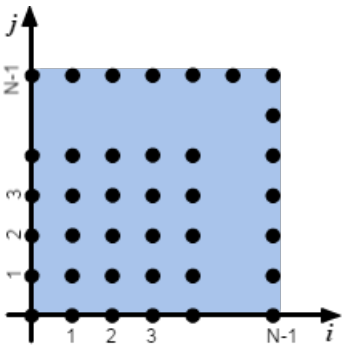


Figura 2.4: Vista geométrica de las instancias de S_2.

2.1.2. Ordenamiento Temporal y Espacial

Los dominios de iteración no explican en qué orden ni en qué lugar (en este caso, procesador) va a ejecutarse una instancia de statement con respecto al resto de ellas. Para ello, utilizamos mapeos de tiempo y espacio. Estos mapeos son funciones que asignan a cada instancia una *fecha lógica*,

que es un vector de valores escalares. Los mapeos de tiempo se denominan *scheduling*s y los de espacio mapeos de *posicionamiento*.

En los mapeos de tiempo, dos fechas lógicas pueden compararse mirando los valores de la primer componente: si son distintas, el menor corresponde a la instancia que se va a ejecutar primero, pero si son iguales, esa comparación pasa a la segunda componente y así hasta que se encuentren componentes distintas. En el caso de que las fechas lógicas sean iguales, no importa el orden en que estas instancias serán ejecutadas, por lo que pueden ser ejecutadas en paralelo. Este tipo de ordenamiento es un orden lexicográfico en función de las componentes de la fecha lógica.

Dada la cantidad de instancias que puede tener un statement en un bucle, los mapeos de fechas lógicas no se realizan por enumeración, pero pueden ser convenientemente expresados como relaciones afines que llamaremos relaciones de *scheduling*. Utilizamos la siguiente relación para expresarlo:

$$\theta_S(\vec{p}) = \left\{ \vec{l_S} \to \vec{t_S} \in \mathbb{Z}^{dim(\vec{l_S})} \times \mathbb{Z}^{dim(\vec{t_S})} \middle| [T_S] \begin{pmatrix} \vec{t_S} \\ \vec{l_S} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\},$$

donde $\vec{l_S}$ es el vector de iteración multidimensional, $\vec{t_S}$ es el vector de la relación de mapeo, y $T_S \in \mathbb{Z}^{m_{\theta_S} \times (dim(\vec{l_S}) + dim(\vec{t_S}) + dim(\vec{p}) + 1)}$ es una matriz de enteros donde m_{θ_S} es el número de restricciones.

Por ejemplo, supongamos que queremos hacer el *scheduling* del programa original de multiplicación de polinomios. Tenemos que asignar fechas lógicas para dejar claro que todas las instancias de S_1 van antes que las de S_2 y ordenar las instancias en S_1 y S_2 en base al orden de sus iteradores. En casos donde hay más de un statement dentro de un bucle, y/o bucles anidados al mismo nivel que otros statements, la fecha lógica requiere algunas componentes más, pero a modo de una explicación simple, en este caso tenemos:

$$\theta_{S_1}(N) = \left\{ (i) \to \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \in \mathbb{Z} \times \mathbb{Z}^2 \middle| \begin{bmatrix} -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 \end{bmatrix} \begin{pmatrix} t_1 \\ t_2 \\ i \\ N \\ 1 \end{pmatrix} = \vec{0} \right\},$$

que corresponde a la función $\theta_{S_1}(N) = \begin{pmatrix} 0 \\ i \end{pmatrix}$ y

$$\theta_{S_2}(N) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} \in \mathbb{Z}^2 \times \mathbb{Z}^3 \middle| \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ i \\ j \\ N \\ 1 \end{pmatrix} = \vec{0} \right\},$$

que corresponde a la función $\theta_{S_2}(N) = \begin{pmatrix} 1 \\ i \\ j \end{pmatrix}$.

La primer componente especifica que todas las iteraciones de S_1 serán ejecutadas antes que las de S_2 y el resto de las componentes en cada fecha lógica ordena las instancias de cada uno de los statements de acuerdo a cómo está ordenado en el bucle.

Muchas transformaciones de programas como *loop skewing*, *interchange*, *reversal*, *shifting tiling*, etc., están definidas en base a la aplicación de distintos *schedulings* de los programas. Los tipos de transformaciones van más allá del alcance de este trabajo y no los explicaremos aquí.

Los mapeos de posicionamiento son iguales a los de *scheduling*, pero con semántica diferente. En vez de asignar fechas lógicas, se asignan procesadores. Un mapeo de espacio-tiempo σ_S devuelve un vector donde se indica información de la fecha lógica y del procesador utilizado.

2.1.3. Funciones de Acceso

Nuevamente, para modelar los accesos a memoria y en el caso de que estos estén sólo en función de bucles externos y parámetros fijos, los modelaremos con las llamadas *relaciones de acceso* a partir de una referencia a un array. Su forma general es:

$$\mathcal{A}_{S,r}(\vec{p}) = \left\{ \vec{l_S} \rightarrow \vec{a}_{S,r} \in \mathbb{Z}^{dim(\vec{l_S})} \times \mathbb{Z}^{dim(\vec{a}_{S,r})} \middle| [A_{S,r}] \begin{pmatrix} \vec{a_{S,r}} \\ \vec{l_S} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\},$$

donde $\vec{l_S}$ es el vector de iteración, $\vec{a}_{S,r}$ es el vector de accesos donde el elemento i-ésimo corresponde al índice de la i-ésima dimensión del array. $A_{S,r} \in \mathbb{Z}^{m_{A_{S,r}} \times (dim(\vec{l_S}) + dim(\vec{a}_{S,r}) + dim(\vec{p}) + 1)}$ siendo $m_{A_{S,r}}$ el número de restricciones, y r el número de referencia del array en el statement. Para el código de la Figura 2.3 la relación de acceso sería:

$$A_{S_1,1}(N) = \left\{ (i) \to (a_{S_1,1}) \in \mathbb{Z} \times \mathbb{Z} \middle| \begin{bmatrix} -1 & 1 & 0 & 0 \end{bmatrix} \begin{pmatrix} a_{S_1,1} \\ i \\ N \\ 1 \end{pmatrix} = \vec{0} \right\},$$

$$A_{S_2,1}(N) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to (a_{S_2,1}) \in \mathbb{Z}^2 \times \mathbb{Z} \middle| \begin{bmatrix} -1 & 1 & 1 & 0 & 0 \end{bmatrix} \begin{pmatrix} a_{S_2,1} \\ i \\ j \\ N \\ 1 \end{pmatrix} = \vec{0} \right\},$$

$$A_{S_2,2}(N) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to (a_{S_2,2}) \in \mathbb{Z}^2 \times \mathbb{Z} \middle| \begin{bmatrix} -1 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} a_{S_2,2} \\ i \\ j \\ N \\ 1 \end{pmatrix} = \vec{0} \right\},$$

$$A_{S_2,3}(N) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to (a_{S_2,3}) \in \mathbb{Z}^2 \times \mathbb{Z} \middle| \begin{bmatrix} -1 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{pmatrix} a_{S_2,3} \\ i \\ j \\ N \\ 1 \end{pmatrix} = \vec{0} \right\}.$$

Las herramientas que trabajan con el modelo poliédrico guardan información extra sobre cada statement y el programa en general, manteniendo asociados los números de referencia a array con sus variables.

2.2. Aplicando Mapeos

2.2.1. Transformaciones

Se necesita una forma de combinar la información de los dominios de interés y de los mapeos de tiempo y espacio para tener una representación del código que va a ser generado. Existen dos maneras de realizar esto, una es transformación inversa y la otra el cambio de base generalizado.

La transformación inversa utiliza las ecuaciones de representación de dominio originales pero en términos de los nuevos índices, luego de haber sido aplicada una transformación. Esta es una forma simple de expresarlo, pero necesita que la matriz de transformación T sea invertible, e incluso cuando lo es, puede generar puntos enteros que no correspondan al dominio de iteración si la matriz de transformación no es unimodular [1]. Por otro lado, el cambio de base generalizado agrega a las ecuaciones de dominio de iteración índices correspondientes al nuevo ordenamiento de instancias, asociándolos

[1] una matriz cuadrada de enteros con determinante 1 o -1 es una matriz unimodular

con la representación de dominio. Si bien esta técnica no requiere ninguna propiedad en la matriz de transformación, agrega complejidad ya que aumenta las dimensiones de la matriz generada. Ambos métodos son utilizados en las herramientas de compilación dependiendo de lo que se necesite hacer.

Las transformaciones realizadas en este trabajo consisten solamente en la eliminación y modificación de instancias de statement del dominio de iteración. El orden original de las instancias no eliminadas se mantienen y también lo hacen las dimensiones del espacio de iteración. Por esta razón, no se necesitan hacer cambios de base ni renombrar índices. El código puede ser generado directamente luego del recómputo de dominios.

2.2.2. Dependencia de Datos

No todas las transformaciones preservan la semántica del código original. Esto ocurre en los casos más comunes, donde el orden de acceso a los datos por parte de las instancias es importante. Dos instancias que acceden a la misma referencia de datos deben mantener su orden de ejecución si al menos una de las dos escribe en la referencia a memoria. Los tipos de dependencia son: *Read-After-Write(RAW)* o *dependencia de flujo*, *Write-After-Read(WAR)* o *anti-dependencia* y *Write-After-Write(WAW)* o *dependencia de salida*. En el caso de dos instancias que hacen lecturas, existe también el tipo *Read-After-Read(RAR)* que técnicamente no es una dependencia, ya que cambiar el orden de statements en tal caso no cambia la semántica, pero mantenerlo es importante para la localidad de los datos.

El modelo poliédrico modela la información necesaria sobre dependencias. Existen muchas abstracciones, desde las simples de dependencia sólo entre bucles, hasta dependencia entre iteraciones, que es la más precisa. ACR preserva las dependencias lo máximo posible de las instancias que no son eliminadas ni modificadas.

2.3. Aplicabilidad

Para que un bucle anidado pueda ser expresado mediante el modelo poliédrico, deben cumplirse las siguientes condiciones:

- las cotas de sus ciclos son funciones afines de las cotas de los ciclos envolventes
- cada predicado es una función afín de los ciclos envolventes y de parámetros globales
- las estructuras de datos presentes en las sentencias de los bucles son arreglos multidimensionales y escalares de cualquier tipo

- los índices de dichos arreglos son funciones afines de los ciclos envolventes y de parámetros globales.

Estas restricciones se dan por la forma de codificar las estructuras de representación poliédrica a partir de *relaciones poliédricas* como las vistas en las secciones anteriores, que solamente poseen restricciones afines y cuya forma general es la siguiente:

$$\mathcal{R}_i(\vec{p}) = \left\{ \vec{x}_{in} \to \vec{x}_{out} \in \mathbb{Z}^{dim(\vec{x}_{in})} \times \mathbb{Z}^{dim(\vec{x}_{out})} \middle| \exists \vec{l_i} \in \mathbb{Z}^{dim(\vec{l_i})} : \begin{bmatrix} A_{out,i} & A_{in,i} & L_i & P_i & \vec{c_i} \end{bmatrix} \begin{pmatrix} \vec{x}_{out} \\ \vec{x}_{in} \\ \vec{l_i} \\ \vec{p} \\ 1 \end{pmatrix} \geq 0 \right\}$$

donde $\vec{x}_{in}$ es una coordenada de entrada, $\vec{x}_{out}$ una de salida, $\vec{l_i}$ un vector de variables locales, $\vec{p}$ el vector de parámetros, $\vec{c_i}$ un vector constante, y $A_{out,i}$, $A_{in,i}$, L_iy P_i matrices enteras. Sin pérdida de generalidad, no incluimos variables locales, por simplicidad. El poder representativo del modelo poliédrico está en su simplicidad y ligado a estas relaciones con restricciones afines.

2.4. Herramientas de Compilación Poliédrica

Existe un amplio rango de herramientas y entornos de trabajo para la compilación poliédrica. Algunas de las destacadas son las siguientes:

- OpenScop: es un formalismo para la representación computacional del modelo poliédrico y también representa un conjunto de librerías para manipular esta representación.
  `http://icps.u-strasbg.fr/people/bastoul/public_html/development/openscop`
- Clan permite transformar programas escritos en C, C++, C#, o Java a la representación poliédrica.
  `http://icps.u-strasbg.fr/~bastoul/development/clan`
- ClooG [2] es una herramienta para el escaneo de poliedros y la generación de código (por ejemplo, C).
  `http://www.cloog.org`
- isl [38] es una librería muy popular para manipular conjuntos de enteros expresados a través de relaciones, y muy eficiente para calcular operaciones entre ellos como unión, intersección, resta, cápsula convexa, chequeo de conjuntos vacíos, entre otras. `http://isl.gforge.inria.fr`

En nuestro trabajo, las herrramientas que utilizamos son OpenScop para la representación poliédrica, Clan para la transformación de código C a OpenScop, isl para el cómputo eficiente de los nuevos dominios y Cloog para la generación de código. Éstas integran el primer aporte de nuestro trabajo, una herramienta llamada Spot, que será descripta en el capítulo siguiente. Nuestro segundo aporte, ACR, las utiliza de forma indirecta a través de Spot. El rol de cada una de ellas se ve más detalladamente en la Sección 3.2.

Capítulo 3

Compilación con Spot

Como primer paso de implementación, y para realizar las tareas de análisis, generación de código y cómputo de regiones de interés, desarrollamos una herramienta experimental llamada Spot. Ésta permite al usuario expresar matemáticamente áreas del espacio de iteración con un formalismo simple y fácil de leer a través de notación de la librería isl. La misma facilita la descripción de conjuntos de enteros de manera muy intuitiva y también la aplicación de operaciones de unión, disjunción, diferencia y otras de manera muy eficiente. Se expresan las diferentes regiones del espacio de iteración en función de iteradores de bucles externos y parámetros fijos, para cambiar el cálculo por otro alternativo o directamente omitirlo del espacio de iteración, es decir, especificar zonas que no van a ser recorridas por los bucles. Para la definición de las diferentes zonas, Spot utiliza un tipo de pragmas específicos. Luego, con ayuda de Clan, ClooG, isl y OpenScop, el código se analiza y se lo traduce al modelo poliédrico, se lo transforma y se genera nuevo código de acuerdo con los pragmas. En la Sección 3.1 se describe el formalismo para indicar diferentes zonas del espacio de iteración y cómo esto impacta en el código. La forma en la que Spot utiliza las herramientas poliédricas para llevar esto a cabo se explica en la Sección 3.2.

3.1. Descripción de Regiones

Spot es una herramienta de compilación *source-to-source*, esto quiere decir que recibe como entrada un programa en un lenguaje de alto nivel y su salida es otro programa de alto nivel. Básicamente, el código de entrada es un bucle perfectamente anidado [1] que tiene statements de cálculo. Los pragmas de optimización de computaciones están definidos por el usuario y tienen el siguiente formato:

- Un *centinela* denotando una región de interés: `#pragma spot`,

[1] tiene statements solamente en el último nivel de anidamiento

- Un *nivel de interés* que indica la prioridad con un entero mayor o igual a 0 (0 corresponde a la menor prioridad).
- Un *dominio de interés* expresando un sub-dominio del dominio de computación original, expresado en notación isl.
- Si el nivel de interés es mayor a 0, un *bloque de statement* que reemplaza el statement de cálculo original en el dominio de interés.

Estos pragmas definen partes del espacio de computación donde el cálculo original se elimina y la zona no es recorrida por el bucle, cuando el nivel de interés es 0, o la zona se reemplaza con otro bloque de statements, cuando el nivel de interés es mayor a 0. Por ejemplo, consideremos el *kernel* de computación de la Figura 3.1. El código original cambia los valores de la matriz a `3.14`, correspondiendo a la computación compleja. Dos pragmas Spot especifican que no se debe realizar ningún cálculo en el dominio descripto por el primero y que las entradas de la matriz deben ser seteadas a `1.11` en el dominio descripto por el segundo, que corresponde a una computación simple. Los dominios de interés están expresados en notación de conjuntos isl.

```
#pragma spot 0 [N]->{[i, j] | 0<=i<=3 and 0<=j<=3}
#pragma spot 1 [N]->{[i, j] | 5<=i<=8 and 2<=j<=9} {A[i][j] = 1.11;}
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    A[i][j] = 3.14;
  }
}
```

Figura 3.1: Kernel Computacional con Pragmas Spot

La ejecución del código generado a partir del código de entrada en la Figura 3.1, con N = 10 y suponiendo que las entradas de la matriz `A` están inicializadas a cero, daría como resultado el siguiente contenido en `A`:

i/j	0	1	2	3	4	5	6	7	8	9
0	0.00	0.00	0.00	0.00	3.14	3.14	3.14	3.14	3.14	3.14
1	0.00	0.00	0.00	0.00	3.14	3.14	3.14	3.14	3.14	3.14
2	0.00	0.00	0.00	0.00	3.14	3.14	3.14	3.14	3.14	3.14
3	0.00	0.00	0.00	0.00	3.14	3.14	3.14	3.14	3.14	3.14
4	3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14
5	3.14	3.14	1.11	1.11	1.11	1.11	1.11	1.11	1.11	1.11
6	3.14	3.14	1.11	1.11	1.11	1.11	1.11	1.11	1.11	1.11
7	3.14	3.14	1.11	1.11	1.11	1.11	1.11	1.11	1.11	1.11
8	3.14	3.14	1.11	1.11	1.11	1.11	1.11	1.11	1.11	1.11
9	3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14	3.14

Otro ejemplo más concreto es la modificación de filtros de imágenes, ya que éstos se aplican punto por punto en todos los píxeles de la imagen. Por ejemplo, un kernel de computación aplica un filtro de enfoque a una imagen cargada de un archivo. Queremos poder codificar fácilmente algunas zonas específicas de la imagen para aplicarles filtros diferentes. Para esto podemos utilizar pragmas Spot para no aplicar ningún filtro en algunas partes del espacio o aplicar otros filtros distintos en otras. En el código de la Figura 3.2 podemos ver la codificación de filtros [2] para algunos sectores de la imagen de la Figura 3.3, donde se le aplica un filtro de enfoque a la imagen original (arriba a la izquierda). La salida del programa original ignorando los pragmas se puede ver arriba a la derecha. Los pragmas codifican el uso de un filtro de afilado más profundo para la margarita, y uno que no realiza enfoque, sino resaltado de bordes para la flor roja de adelante.

```
#pragma spot 1  [H, W]-> {x,y | x > flor_lateral_izq
                               or y > flor_base
                               or x < flor_lateral_der
                               or y < flor_tope }
                           {enfoque_drastico(x, y, img);}
#pragma spot 2  [H, W]-> {x,y | x > rosa_lateral_izq
                               or y > rosa_base
                               or x < rosa_lateral_der
                               or y < rosa_tope }
                           {bordes(x, y, img);}
for (x = 0; x < H; x++)
   for (y = 0; y < W; y++)
      enfoque(x, y, img);
```

Figura 3.2: Usando Spot para cambiar filtros fácilmente en el procesamiento de imágenes.

3.1.1. Conjuntos en notación isl

La notación isl para expresar dominios de iteración es compacta e intuitiva. Por ejemplo, para expresar el espacio de iteración del código de multiplicación de polinomios de la Figura 2.3 podemos escribir:

```
S1: [N] -> { [i] : 0 <= i < 2 * N-1 }
S2: [N] -> { [i, j] : 0 <= i < N and 0 <= j < N}
```

[2] Los filtros, así como las imágenes utilizadas fueron obtenidos del tutorial de filtros de Lode Vandevenne: `http://lodev.org/cgtutor/filtering.html`.

Figura 3.3: Aplicación de diferentes filtros con Spot. Arriba a la izquierda: imagen original de entrada. Arriba a la derecha: imagen de salida con filtro de enfocado. Abajo: imagen de salida con un filtro de enfocado común para toda la imagen excepto para la región de la margarita, que tiene un filtro más profundo de enfoque, y para la zona de la flor roja donde se aplica un filtro de resaltado de bordes.

El espacio de iteración en el que se ejecuta el statement `A[i][j] = 3.14` en el código salida de Spot para el kernel de la Figura 3.1 puede ser expresado como

```
[N] -> { [i, j] : i >= 0 and j >= 0
               and ((9 <= i < N and j < N)
                 or (5 <= i <= 8 and 10 <= j < N)
                 or (i <= 3 and 4 <= j < N)
                 or (5 <= i <= 8 and i < N and j <= 1));
         [4, j] : N >= 5 and 0 <= j < N }
```

que es el resultado de restarle al dominio de iteración original los dominios de interés de los pragmas y teniendo en cuenta diferentes valores de `N`. La operación también se puede expresar en isl como

```
[N]-> {[i, j] : 0 <= i < N and 0 <= j < N}
        - {[i, j] : 0<=i<=3 and 0<=j<=3 }
        - {[i, j] : 5<=i<=8 and 2<=j<=9 }
```

Spot elimina los sub-dominios especificados en los pragmas al dominio de iteración con el statement original. Luego, a cada dominio especificado en un pragma lo asocia a un statement alternativo si el nivel de interés es mayor a 0. Finalmente se rearma el bucle con estas computaciones alternativas, el cual no recorre las partes del espacio con nivel de interés igual a 0.

3.2. Funcionamiento

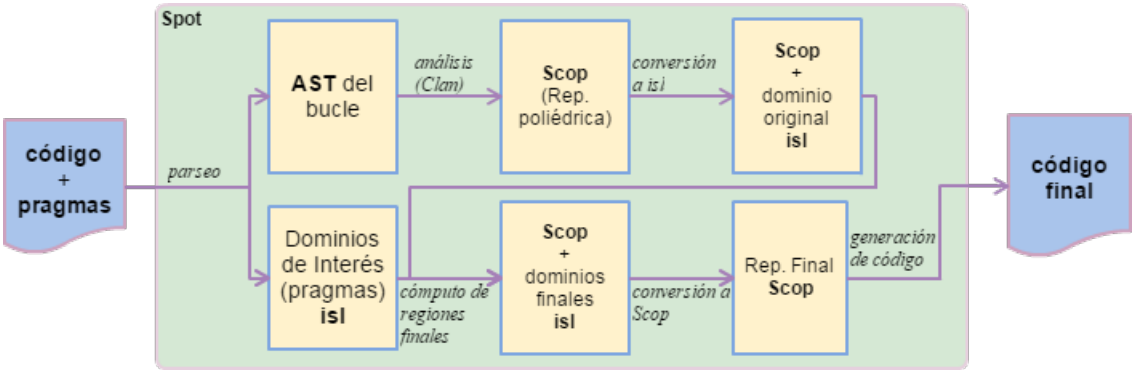


Figura 3.4: Flujo de información y operaciones de transformación de código de Spot.

Spot está desarrollado en C, al igual que todas las herramientas poliédricas que utiliza. Aunque este fue programado de forma procedural, se planea reestructurarlo para que sea código orientado a objetos, adoptando los mismos lineamientos que éstas. Esto incluiría también utilizar la Convención para C++ de Google [3] en los aspectos que fuese aplicable a C. Su código se encuentra disponible en la web [4].

En primera instancia, necesitamos una herramienta que implemente la representación poliédrica. Para ello, elegimos la representación de OpenScop para el código completo y luego isl para las transformaciones de los dominios de interés. OpenScop es una representación con la que el resto de las herramientas poliédricas pueden interactuar e isl es muy eficiente para el cómputo de nuevas regiones.

[3] https://google.github.io/styleguide/cppguide.html
[4] https://github.com/cesarsabater/spot

La entrada de Spot es código con pragmas embebidos en él, arriba de del bucle perfectamente anidado que vamos a transformar. Utilizamos Clan para analizar el código y obtener la representación del kernel original. En esta fase de análisis también se detectan los pragmas y sus conjuntos isl, cada uno asociado a un statement o indicando que en ese lugar no se realiza ningún cálculo, dependiendo del nivel de interés. Al dominio de iteración del kernel original en OpenScop, lo traducimos a isl para hacer eficiente el cómputo de nuevos dominios de iteración.

Ahora, para representar los dominios, tenemos una lista de tuplas de la forma `(isl_set, *statement, prioridad)` que asocia cada región con un puntero a su cómputo y su prioridad, que es proporcional a su nivel de interés. Para el caso en donde no se realiza ningún cómputo, el valor de `*statement` es `null`. Sin embargo, estos conjuntos se superponen ya que el conjunto del dominio original lo hace con todos los demás y es posible que los otros también se superpongan entre ellos. Igualmente hay que tener en cuenta que el código que realmente se va a ejecutar en un subconjunto compartido es el que tiene mayor `prioridad`. Entonces, para armar una partición con los dominios reales hacemos lo siguiente: para cada tupla cuyo `*statement` no es nulo, tomamos su conjunto y le restamos los conjuntos de las tuplas cuya prioridad es mayor. Luego, descartamos de la lista las tuplas cuyo `*statement` es `null`, ya que son regiones donde no vamos a necesitar generar código. De esta manera, obtenemos nuevas tuplas `(isl_set, *statement)` donde cada región queda correctamente delimitada y se ignoran los cálculos deseados.

Una vez procesados estos dominios, se los vuelve a traducir a una estructura unificada en OpenSCop para la generación de código en Cloog.

Capítulo 4

Refinamiento Adaptivo de Código

ACR es una técnica para la transformación automática y dinámica de programas que computan aproximaciones. Para ello, se basa en la compilación poliédrica y conocimiento específico del dominio del programa a optimizar. ACR toma como entrada un kernel computacional dedicado al cálculo aproximado de soluciones y un conjunto de pragmas especialmente diseñados para que el usuario pueda suministrar condiciones y estrategias de optimización, como por ejemplo el de la Figura 4.1. La técnica ahorra cálculos en tiempo de ejecución realizando transformaciones al kernel sin preservar la semántica original del código, pero manteniendo una precisión aceptable de acuerdo al conocimiento de dominio codificado en los pragmas. Para realizar estas transformaciones, ACR monitorea el conjunto de estados del programa, tomando valores a través de una cuadrícula embebida en el espacio de cómputo. Con los valores cargados en esta cuadrícula, identificamos las zonas donde el código puede ser transformado ya que la complejidad de los cálculos necesaria para asegurar un cierto nivel de precisión es menor que la del código original. Habiendo identificado diferentes áreas de complejidad, ACR realiza las transformaciones continuamente utilizando Spot. Para asegurar precisión, el código generado es verificado a medida que el estado evoluciona en la ejecución, realizando nuevos cambios si es necesario.

El dominio de aplicación de ACR corresponde a kernels computacionales en los cuales los posibles valores de cada iterador pueden ser modelados por un conjunto de restricciones afines en ese iterador, iteradores exteriores y parámetros fijos. Un caso simple y útil corresponde a bucles cuyo paso es una constante conocida y sus cotas son funciones lineales de variables asociadas a los bucles exteriores y parámetros fijos. Esta limitación está relacionada a la representación poliédrica de programas [14] pero en nuestro caso son menores las restricciones porque se tiene acceso a cualquier tipo de datos y está permitida la omisión o modificación de dependencias, según

```
// Bucle iterando sobre las celdas de una simulación de fluido
while(true) {
  ...
  // lin_solve kernel
  #pragma ACR grid(10)
  #pragma ACR monitor(density[i][j], max, filtro)
  #pragma ACR alternative bajo(parameter,    MAX = 1)
  #pragma ACR alternative medio(parameter, MAX = 3)
  #pragma ACR alternative alto(parameter,   MAX = 4)
  #pragma ACR alternative fuente(code, lin_solve_complex(k, i, j))
  #pragma ACR strategy direct(1, bajo)
  #pragma ACR strategy direct(2, medio)
  #pragma ACR strategy direct(3, alto)
  #pragma ACR strategy \
        zone("|i-N/2| < N/8 and |j-N/4| < N/8", fuente)
  for (k = 0; k < MAX; k++) {
    for (i = 1; i <= N; i++) {
      for (j = 1; j <= N; j++) {
        lin_solve_computation(k, i, j);
      }
    }
  }
  ...
}
```

Figura 4.1: **Pseudo-código que muestra una posible forma de aplicar ACR al kernel `lin_solve` del caso de estudio de simulación de fluidos del Capítulo 5**. El estado monitoreado es la matriz bidimensional `density` de `N` × `N`, que almacena valores de densidad. La función `filtro` clasifica distintos valores de densidad: cercana a cero (1), media (2), o alta (3). El valor `monitor` para una celda corresponde al máximo de todos el rango de valores en la celda. Este rango es usado para la aplicación de diferentes cálculos alternativos para cada celda. En este ejemplo, el bucle `k` va a iterar más o menos dependiendo de ese rango. A la porción del estado donde ingresa el fluido (`fuente`) se le aplica el cálculo alternativo `lin_solve_complex`.

lo especificado en los pragmas de usuario. De cualquier manera, ACR está fuertemente basado en técnicas de compilación poliédrica. Particularmente, ACR representa el espacio de computación y sus celdas como poliedros de la misma manera que los compiladores poliédricos representan los dominios de iteración [6, 14, 28]. Por ejemplo, el espacio de computación del bucle

`lin_solve` en la Figura 4.1 es:

$$\mathcal{S}_{lin\_solve} = \left\{ \begin{pmatrix} k \\ i \\ j \end{pmatrix} \middle| \begin{array}{lll} 0 \leq & k & < MAX \\ 1 \leq & i & \leq N \\ 1 \leq & j & \leq N \end{array} \right\}$$

Para que se pueda aplicar ACR, el código, debe poder transformarse utilizando el modelo poliédrico de tal manera que se puedan realizar cómputos alternativos en regiones específicas. La clase de aplicaciones objetivo de ACR debe tener las siguientes propiedades:

1. **Representatividad Poliédrica**: el kernel, y el conjunto de todas las transformaciones que pueda realizarle ACR de acuerdo a información suministrada por el programador, deben poder ser representadas con el modelo poliédrico como está descripto en la Sección 2.3.

2. **Existencia de Regiones de Complejidad**

   a) Un conjunto conexo de estados típico de una ejecución, es decir, que sea representativo de los posibles valores que pueda tomar la ejecución dada la naturaleza del programa optimizado, debe exhibir regiones calculables con cómputos de menor poder computacional.

   b) Estos cómputos deben poder integrarse con el resto de los cómputos afectando la calidad de la solución en valores bajos y aceptables por el programador.

3. **Cuadrícula de Monitoreo Embebible**:

   a) Una cuadrícula debe poder embeberse en el espacio de cómputo, particionándolo en conjuntos de elementos asignados a celdas de la misma. Los elementos de cada celda deben tener mayor probabilidad tener la misma complejidad de cálculo y necesitar el mismo tipo de cómputos para calcular su solución. A partir de un conjunto de celdas se debe poder contener los elementos pertenecientes a regiones de complejidad, describiéndolas con una buena precisión.

   b) La función que asigna elementos a una celda, debe ser fácil de calcular y de orden constante.

   c) Cualquier conjunto de cómputos que modifica una porción del estado que compone una celda solamente debe modificar elementos de esa celda, y debe poder representarse de forma individual por el modelo poliédrico.

4. **Ventaja de la Optimización Dinámica**: Si se desearía realizar estáticamente una optimización del mismo tipo para la aplicación,

significaría un alto costo de control no paralelizable, comparado con realizarlo dinámicamente con ACR.

La propiedad 4 no es completamente necesaria, ya que si un programa que no la cumple, este igualmente puede ser optimizado con ACR y el programador se beneficiaría ya que la optimización se realiza automáticamente, solamente ingresando algunos pragmas. De todos modos, para aplicaciones donde realizar estáticamente una optimización que adapte la complejidad de cómputos a regiones cambiantes en tiempo de ejecución requiere muchas comprobaciones, ACR sería una muy buena alternativa.

El código donde se realice el cálculo aproximado deberá tener la forma de bucles computacionales con pragmas ACR suministrados por el usuario. Los pragmas brindan información de alto nivel para transformar el espacio de iteración de los bucles y los cálculos que se realizan en ellos, mayormente en base a información monitoreada durante la ejecución. En esos bucles, el rígido modelo de dependencias usado por los compiladores es explícitamente relajado y se pueden usar cálculos alternativos para que el cómputo de los resultados sea más rápido, de acuerdo con estrategias definidas por el usuario. Estas estrategias pueden ser estáticas, es decir independientes de los valores computados en tiempo de ejecución para áreas específicas del espacio de computación, o dinámicas –es decir dependientes de estos valores– donde no se pueden aplicar estrategias estáticas. Los pragmas se describen detalladamente en la Sección 4.1.

El algoritmo de optimización de ACR está basado en tres componentes principales. Primero, se evalúa continuamente una cuadrícula de estado en base a estrategias definidas por el usuario como se detalla en la Sección 4.2. Luego, un thread dedicado genera código optimizado de acuerdo a la cuadrícula. Este proceso está explicado en la Sección 4.3. Finalmente, un thread de ejecución asegura que sea ejecutada la mejor versión del código teniendo en cuenta performance y precisión, como se explicará en la Sección 4.4.

La técnica fue implementada en un prototipo y evaluada con un código de simulación de fluidos hecha en C para visualización de Gráficos de Computadora. El caso de estudio y la evaluación de ACR será presentada en el Capítulo 5. El prototipo tuvo como fin obtener resultados preliminares de ACR en un código de producción, por lo que no se tomaron más decisiones de diseño que realizar una implementación rápida en C con paradigma procedural. El código esta disponible en la web [1].

4.1. Información de alto nivel: Pragmas ACR

ACR suministra al usuario un conjunto de pragmas de alto nivel que permiten proveer información sobre conocimiento específico de dominio pa-

[1] `https://github.com/cesarsabater/Floppy`

ra realizar la aproximación. El algoritmo de optimización aprovechará los pragmas para computar resultados aproximados con buena precisión de forma automática. Estos pragmas son extensiones del lenguaje de programación que deben ser insertados antes de un bucle con el objetivo de calcular aproximaciones usando implementaciones alternativas a las suministradas dentro del bucle o reducir los cálculos en algunas áreas del espacio de computación. El conjunto de pragmas está compuesto por cuatro palabras clave:

1. La palabra clave `grid` define la resolución de la cuadrícula que monitorea el espacio de cálculo para la estrategia dinámica de aproximación: el espacio de cálculo está descompuesto en la cantidad de celdas definida por el valor de `grid`. El monitoreo así como también la estrategia de aproximación serán hechas al nivel de granularidad del tamaño de celda. Cuanto más resolución tenga la cuadrícula, más precisa será la descripción de las regiones. Su formato es:
   `#pragma ACR grid(`*tamaño*`)`
   donde *tamaño* es un número constante: si el espacio de cálculo es de dos dimensiones, el tamaño de la celda sería *tamaño* × *tamaño*.

2. La palabra clave `monitor` especifica qué datos tienen que ser monitoreados para las estrategias de optimización dinámica y de qué forma el monitoreo se va a resumir para las celdas. Su formato es:
   `#pragma ACR monitor(`*datos*`,` *síntesis*`[,` *filtro*`])`
   donde *datos* una referencia conjunto de valores de estado a observar (comúnmente dependiente de las coordenadas del espacio de cálculo), *síntesis* especifica cómo resumir los datos de todos puntos de iteración que corresponden a una celda en un solo valor (usando políticas predefinidas, e.g, `media`, `max`, `min`, etc.) y *filtro* es una función opcional que puede ser usada para preprocesar los valores en bruto monitoreados (por ejemplo, para clasificarlos en categorías): el valor usado para la estrategia de optimización será *filtro(datos)*. Por ejemplo, si el bucle itera sobre `i` y `j`, *datos* puede ser `valor[i][j]` y *preprocesador* una función que asigna categorías a diferentes valores para tener una representación más simple.

3. La palabra clave `alternative` define un cálculo alternativo al original en el código, es decir el bloque de statements que se encuentra dentro del bucle, la porción de código que se repite. Su formato es:
   `#pragma ACR alternative` *nombre*`(`*tipo*`,` *efecto*`)`
   donde *nombre* es el nombre de una estrategia y *tipo* especifica el tipo de estrategia: sólo `parameter` para mantener los statements de cálculo originales pero con otros parámetros a definir en el campo *efecto*, o `code` para suministrar un bloque de código alternativo en el campo *efecto*.

4. La palabra clave `strategy` especifica, para cada celda, en qué condiciones se debe tomar cada alternativa definida. Estas pueden ser estáticas, especificando áreas del espacio de cálculo para cierta alternativa, o dinámicas, especificando qué alternativa usar dependiendo de valores monitoreados o su evolución en esa celda. En el caso de estudio presentado en el capítulo siguiente se establece una vinculación directa entre un valor dinámico de monitor y cálculos alternativos. La forma del pragma es:
   `#pragma ACR strategy direct(`*valor*`,` *alternativa*`)`
   donde *valor* es un valor que puede ser capturado por un monitor para una celda de la cuadrícula y *alternativa* es la alternativa a aplicar a la celda si el monitor reporta ese valor. Una estrategia estática tiene la siguiente forma:
   `#pragma ACR strategy zone(`*área*`,` *alternativa*`)`
   donde *área* es una parte del espacio de cálculo expresado usando notación de conjuntos similar a la de Spot, (por ejemplo, {`i | 0<=i<=3`} para expresar parte del espacio donde i está entre 0 y 3) y *alternativa* es el nombre de una alternativa definida en el aspecto `alternative`.

La Figura 4.1 muestra cómo este conjunto de pragmas puede ser utilizado para especificar computaciones aproximadas en una versión similar a la de nuestro caso de estudio detallado en el Capítulo 5. Luego de que el código haya sido modificado con compilación poliédrica, éste difiere radicalmente de su forma original. En el Apéndice A.2 se muestra una versión del kernel transformado para calcular una buena aproximación cuando se realizaron 982 pasos de simulación. La utilización de condicionales en los bucles internos para controlar el flujo son una opción que mantiene la forma original del kernel, pero el código es menos eficiente.

4.2. Cuadrícula de Estado

ACR necesita una manera de obtener regularmente información sobre el estado de la ejecución para poder identificar diferentes tipos de regiones para aproximar, es decir, las partes del espacio de iteración donde la estrategia de aproximación debería ser diferente. Esta tarea debe ser lo suficientemente liviana como para no agregar una sobrecarga significativa de cálculo con respecto al tiempo ahorrado por el código optimizado. Para lograrlo, una cuadrícula uniforme es embebida en la porción del espacio de estados que se quiere monitorear para representar diferentes zonas de cálculo, con tamaño acorde al pragma `grid`. Cada celda en la cuadrícula representa una porción (hiper-)cúbica del espacio y resume información descriptiva teniendo en cuenta valores relevantes de esa zona según lo indicado por el usuario en el pragma `monitor`. La información almacenada en la cuadrícula y su evolución durante la ejecución permiten decidir qué nivel de complejidad es

el adecuado para el conjunto de cálculos cuyo resultado impactará sobre la porción de solución de cada celda. Por un lado, la cuadrícula permite resumir información relevante con pocos valores, y por otro permite representar de forma regular valores que potencialmente estarán dispuestos de forma irregular, permitiendo luego trabajar fácilmente con su representación poliédrica. Una vez que la cuadrícula posee toda la información necesaria, las regiones se construyen uniendo las celdas con estrategias similares de aproximación. El proceso completo se puede ver en la Figura 4.2. A nivel técnico, cada celda de la cuadrícula es representada como un poliedro y cada región es construida añadiendo celdas a partir de uniones poliédricas con la ayuda de `isl`.

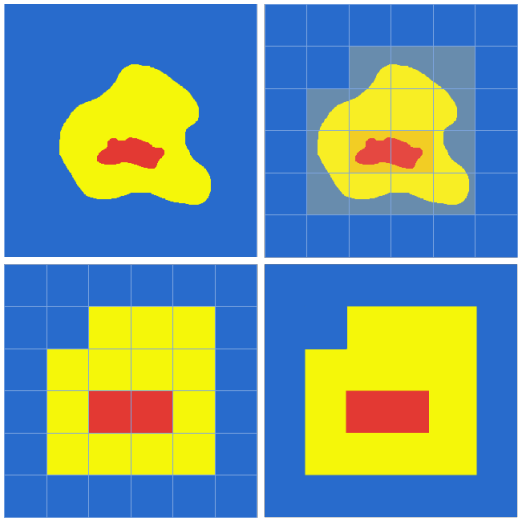


Figura 4.2: Proceso de obtención de regiones de complejidad codificadas como conjuntos poliédricos, a partir de conjuntos de valores con formas irregulares en la ejecución del código. De izquierda a derecha y de arriba hacia abajo, las imágenes muestran: (1) un estado de ejecución con regiones de cálculo altamente irregulares representadas por los colores azul, amarillo y rojo, (2) la división del espacio de cálculo en celdas, (3) los valores monitoreados resumidos en uno solo por celda, según la *síntesis* y el *filtro* del pragma `monitor` y (4) las regiones, producto de la unión de celdas con clasificaciones similares según su pertenencia al dominio de aplicación de cada estrategia de aproximación.

La relación de tamaño entre la resolución de la cuadrícula con respecto a la resolución del conjunto de estados monitoreado debe ser grande ya que de otro modo el cómputo de cada nueva aproximación requerirá una cantidad alta de operaciones para generar regiones geométricamente complejas y en base a muchas celdas. Esto haría que los cálculos sean demasiado complejos y haya una sobrecarga de transformación del código. Cada celda de la

cuadrícula debe representar a una cantidad de valores del estado de ejecución que justifique resumirse en uno solo. Por otro lado, si la resolución es demasiado baja, es decir, si hay demasiados valores heterogéneos en cada celda, se perderá la capacidad de describir la forma de las regiones con buena fidelidad. En este caso se pierde la capacidad del código de adaptarse a diferentes zonas del espacio de estados, que es lo que estamos buscando. Lo aconsejable es una cuadrícula de resolución lo más baja posible para que sea eficiente en el momento de computarla pero que permita generar código sin perder poder representativo de las regiones relevantes. Como por ejemplo, la resolución de la cuadrícula para un fluido con movimiento turbulento de la Figura 4.3, que si bien no da información de la forma de las curvas con nivel de detalle, captura las zonas donde predominan diferentes colores, que se corresponden con distintos niveles de complejidad de cálculo.

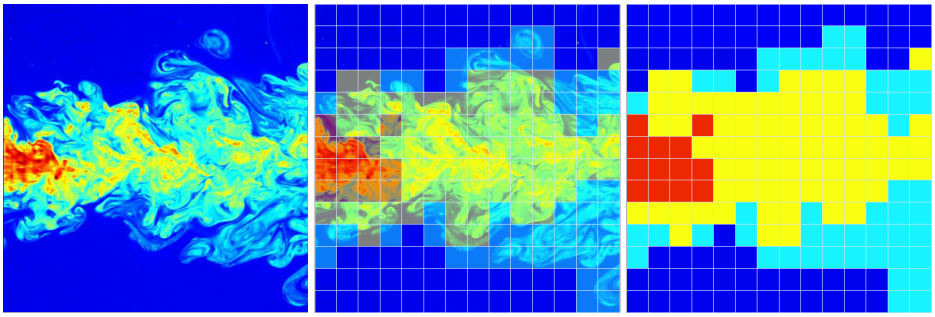


Figura 4.3: Proceso de obtención de regiones para un momento de la ejecución de una simulación de fluido en zonas turbulentas. En complejidad de cálculo ascendente, los colores azul, celeste, amarillo y rojo representan las distintas zonas.

En base a a la información de las regiones de complejidad, se transformará el kernel a la medida de los valores de la ejecución. Éste itera sobre el estado de la solución, pero no necesariamente su dominio de iteración es el mismo espacio que el del estado, ya que el espacio de iteración representa el espacio de los iteradores de los bucles del kernel, mientras que el espacio de estado, el de los valores de la solución. Entonces, ambas representaciones poliédricas puede diferir entre sí. Por ejemplo, el caso de estudio tratado en el siguiente capítulo simula un fluido cuya porción de estado monitoreado se representa con una matriz bidimensional. Por otro lado, el kernel que computa cada paso de la simulación es un algoritmo que itera varias veces sobre los mismos puntos de esta matriz, y su espacio de cálculo es tridimensional, como se puede ver en la Figura 4.4. En este caso, el conjunto de valores del estado es una proyección del dominio de iteración sobre el hiperplano correspondiente al espacio de estados.

Cuando el espacio de cálculo difiere del espacio de estados de ejecución, la porción del dominio de iteración del kernel que se transformará según

los valores de una celda cualquiera dada, es el conjunto de instancias de statement que realizan escrituras sobre ella.

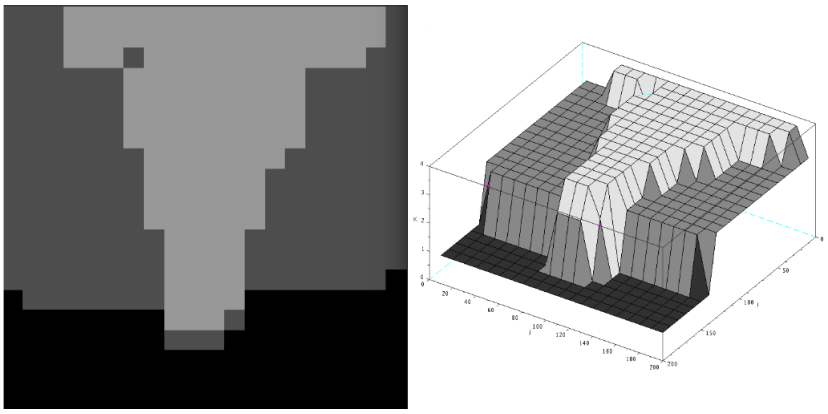


Figura 4.4: A la izquierda valores de la cuadrícula que monitorea una simulación de fluido, en donde las regiones en negro son regiones de muy baja complejidad, las regiones en gris oscuro son regiones de complejidad media, mientras que las regiones en gris claro son regiones que requieren cálculos precisos. A la derecha, el volumen de cálculo que tiene el espacio final de iteración del kernel generado con la cuadrícula.

4.3. Generación dinámica de Código

En el momento de la ejecución que se termina de monitorear el estado, se deben asignar estrategias de aproximación adecuadas a cada región según los pragmas `strategy`. Luego, se genera código optimizado sobre la marcha para reemplazar el que está computando soluciones en ese momento. Esta generación y reemplazo del código actual se realizará solamente si este ya no es adecuado para el estado de la simulación, es decir, cuando las regiones de la cuadrícula cambian. La generación de código se realiza con herramientas de compilación poliédrica. Aprovechamos librerías de Spot para hacer solicitudes simples de generación de código eficiente en base a una representación poliédrica como vimos en el Capítulo 3, pero con la diferencia de que las regiones no se ingresan a través de pragmas Spot, sino que las computa ACR.

Para agrupar las celdas de la cuadrícula en zonas con estrategias de aproximación similares, creamos conjuntos poliédricos usando operaciones de unión de los conjuntos de puntos correspondientes a cada celda. Estos conjuntos son una partición del estado, y cada uno está asociado al cálculo correspondiente a su estrategia de aproximación, que puede ser un bloque de código alternativo o restricciones a los parámetros de la ejecución, según el pragma `alternative`. Estas uniones forman los dominios de iteración del

problema de generación de código. Luego, para asegurar que los cálculos aproximados preservan el orden de las computaciones originales lo más posible (incluso para los casos en donde haya cálculos ignorados o que éstos cambien), reforzamos el orden lexicográfico utilizado en el espacio de cálculo original, como relaciones de *scheduling* de entrada para la generación de código.

Teniendo estas regiones armadas, utilizamos ClooG (a través de Spot) para generar código con un costo de control altamente optimizado y, para casos de transformación dinámica, evitando pruebas costosas en bucles internos para elegir la estrategia de aproximación indicada, a diferencia de una optimización estática. Un ejemplo del código generado a partir de la cuadrícula de la Figura 4.2 lo podemos ver en la Figura 4.5.

El bajo costo de control se logra ya que a diferencia de un código compilado estáticamente, el código que generamos en tiempo de ejecución es aplicado únicamente para el estado actual de la ejecución y cuando ésta cambie lo suficiente, se generará una nueva versión. Una versión compilada para un estado específico no tiene que gastar tiempo de cómputo en controlar qué tipo de cálculos realizar ni qué cantidad de ellos para cada celda de la cuadrícula, porque eso se resuelve en la generación de código. De esta manera, se evitan los condicionales que pueden aparecer en un gran porcentaje de las iteraciones totales o para algunos casos en su totalidad, que se necesitarían en un enfoque estático correspondiente a una optimización manual que realice los mismos ahorros y mantenga el orden de las iteraciones. Un ejemplo de este tipo de códigos se verá en el caso de estudio del Capítulo 5. Además, como se verá en la Sección 4.4 para aprovechar arquitecturas multinúcleo, la generación de código se realiza en un thread aparte, por lo que no afecta la ejecución.

Como vimos en el Capítulo 3, Spot utiliza Clan para hacer el análisis de código. Esta herramienta de compilación poliédrica sirve para transformar código de alto nivel como C a su representación poliédrica, transformando bucles en dominios de iteración, relaciones de acceso y de *scheduling* descriptas en la Sección 2.1. Como el único cambio principal en el código es el espacio de iteración y los bloques de statement para cada región, el análisis del código solamente se realiza una vez, al principio de la ejecución del código que queremos optimizar. Cuando tenemos una representación poliédrica base del kernel, parametrizamos su generación de acuerdo a cambios en las regiones dadas por las alternativas de la cuadrícula. Al hacerlo de esta manera evitamos costo adicional de análisis cada vez que necesitamos nuevo código.

Una vez que Spot haya generado una versión código de aproximación, éste es compilado y cargado de forma dinámica al proceso de computación. Para el caso de estudio presentado en el siguiente capítulo la Figura 4.4 muestra una vista del volumen de cálculo que termina siendo ejecutado por el código generado de acuerdo al estado de la cuadrícula y información de

usuario vista en la Figura 4.1. Si tuviéramos que representar el volumen de cálculos realizados por el código original, en el ejemplo, éste sería un prisma rectangular de dimensiones iguales a los topes de sus tres iteradores. El código generado automáticamente tiene las siguientes propiedades: (1) no tiene pruebas internas costosas para decidir la estrategia de optimización, (2) hace cálculos complejos sólo cuando es necesario y (3) los cálculos que se realizan, se realizan respetando con la mayor fidelidad posible el orden original para mantener la semántica.

Como el código está generado solamente en base a los valores de la cuadrícula y estrategias definidas estáticamente, hay una relación biunívoca entre una cuadrícula y una versión de código. Es decir, una cuadrícula es una representación válida del código que se generó en base a ella. Como veremos en la siguiente sección, esto es útil para comparar distintos códigos.

4.4. Ejecución

La ejecución de ACR está desacoplada en dos threads que aprovechan arquitecturas multinúcleo para reducir sobrecarga de la técnica. El primer thread es responsable de los cálculos en sí mismos mientras que el segundo está dedicado a la generación de código y responderá solicitudes de nuevas versiones del kernel hechas por thread de cálculo.

Los threads operan de la siguiente manera: al principio de la ejecución, no hay código optimizado disponible. Por lo que el thread de cálculo ejecuta el código original para la primera iteración y monitorea nuevos valores del estado de ejecución para actualizar la cuadrícula. Luego solicita código optimizado al thread de generación de código en base a esta cuadrícula. El thread de cálculo continúa su ejecución con la implementación anterior y cuando el thread de generación de código posee código optimizado listo para usarse, el thread de cálculo comprueba primero si el código disponible todavía sirve para el estado actual de la cuadrícula o no. Si es así entonces cambia la ejecución a éste, sino lo ignora y realiza una nueva solicitud con el nuevo estado de la cuadrícula pero continúa haciendo cálculos con el código original.

Un caso frecuente es el de llegar a un estado de la ejecución en el que estamos utilizando ya una versión optimizada del kernel generada una cantidad relativamente cercana de iteraciones atrás. La ejecución continuará hasta que el estado cambie lo suficiente y la optimización no sea adecuada. Esto puede suceder porque ya no respeta las estrategias que deben ser aplicadas o porque existen optimizaciones más adecuadas y eficientes para para el nuevo estado. En este caso, la ejecución debe cambiar rápidamente a una nueva versión del código ya que la optimización actual podría no ser segura y violar dependencias al no aplicar las estrategias de los pragmas correctamente. Por otro lado, existen mejores optimizaciones para calcular los

Sea $kern_{orig}$ la función que aplica una iteración del kernel original ;
Sea $kern_{optm}$ la función que aplica una iteración del kernel optimizado ;
Sea $Grid_{optim}$ la cuadrícula utilizada para generar $kern_{optim}$;
Sea $State$ el estado de la solución actual ;
while *la ejecución no terminó* **do**
  Sea $kern = kern_{orig}$ el kernel que va a ser aplicado ;
  Actualizar $Grid$ basándose en $State$;
  if $Grid_{optim} \geq Grid$ **then**
    $kern = kern_{optim}$;
  end
  $kern(State)$;
end

Algorithm 1: Thread de Cálculo

Sea $Grid$ la cuadrícula del estado de la solución actual ;
Sea $GenCode(grid)$ una función que compila y carga código en base a $grid$;
while *la ejecución no terminó* **do**
  if $Grid_{optim} \neq Grid$ **then**
    Sea $Grid_{gen} = Grid$ un *snapshot* ;
    Generar nuevo kernel basándose en $Grid_{gen}$;
    Compilar y Cargar el kernel generado en $kern_{optim}$;
    $Grid_{optim} = Grid_{gen}$
  end
end

Algorithm 2: Thread de Generación de Código

resultados para el estado actual que con el código que se está utilizando. Si la versión de código solicitada todavía no está lista, cambiamos a la original solamente en el caso de que el código actual no sea seguro. De otro modo, lo seguimos utilizando hasta que la optimización solicitada esté lista, porque es más eficiente que la original. En los Algoritmos 1 y 2 muestran como trabaja cada thread, y la Figura 4.6 los muestra trabajando en conjunto.

Verificar que una versión de código sea segura se traduce a comprobar que ésta realice cálculos tanto o más precisos que los necesarios para el instante actual de la ejecución. Como vimos en la sección anterior, una cuadrícula es suficiente para representar el código que se generó. Debido a esto, una manera fácil de verificar seguridad es comparar la cuadrícula de código generado con la cuadricula actual del estado de ejecución, y si la primera posee estrategias tanto o más precisas que la segunda en cada una de sus celdas, el código optimizado satisface lo especificado en pragmas por el usuario, aunque pueda no ser el más eficiente que podríamos generar. En

el Algoritmo 1, esto lo notamos con $Grid_{optim} \geq Grid$.

La implementación en C de ambos threads y el control del acceso a las regiones críticas por parte de éstos se realizó con *POSIX Threads*. Los threads se comunican a través dos estructuras compartidas: la cuadrícula de monitoreo y un apuntador de función que hace referencia al kernel. Existen cuatro regiones críticas en el código, dos accedidas por el thread de cálculo y otras dos por el thread de generación de código:

1. En el thread de cálculo:

   a) La escritura de la cuadrícula durante el monitoreo.

   b) El referenciamiento del apuntador al kernel optimizado para realizar cálculos.

2. En el thread de generación de código:

   a) La lectura de la cuadrícula para generar una nueva versión de kernel.

   b) La actualización del apuntador al kernel, cuando está lista una nueva versión del mismo.

Las regiones críticas $1a$ y $2a$ tienen la cuadrícula en común. Si ésta se lee mientras está siendo actualizada por el thread de cálculo, la lectura puede ser inconsistente, generando un kernel inadecuado. El acceso a estas regiones es controlado por un mutex. Las regiones críticas $1b$ y $2b$ tienen el apuntador al kernel en común. Si el thread de cálculo referencia este apuntador mientras el thread de generación de código está modificándolo, puede se referencie un lugar equivocado, y que probablemente se produzca una excepción. Para evitar esta situación, también se controla el acceso a estas regiones con un otro mutex.

Si bien las regiones críticas correspondientes a la cuadrícula pueden generar kernels inconsistentes, esto no es siempre un problema. Supongamos el caso en que no tenemos control por mutex y la cuadrícula es leída por el thread de generación de código mientras es modificada por le thread de cálculo. Esta cuadrícula puede contener celdas con valores de dos o más estados diferentes pero consecutivos. Si bien es una cuadrícula híbrida, no generaría problemas. En primer lugar, el thread de cálculo comprueba que todo kernel sea seguro antes de ser ejecutado, esto impediría que un kernel generado con una cuadricula híbrida haga menos cálculos que los necesarios para una solución aceptable. En segundo lugar, un kernel no es generado para solo un estado, sino que para un conjunto, y esta cuadrícula sigue teniendo información actualizada del conjunto de estados relevantes. Por esta razón, región crítica no es necesaria, pero puede resultar útil para ciertas aplicaciones. Por ejemplo, aplicaciones en las cuales el estado tiene poca

variación por segmentos de tiempo, y cambios bruscos en ciertos momentos de la ejecución. En estos casos quizás sea conveniente generar un kernel ajustado a un estado en particular, ejecutarlo en un segmento de tiempo y luego generar otro para un cambio grande. En la implementación de ACR para el caso de estudio del capítulo siguiente, el mutex para la cuadrícula no es utilizada porque el estado de la aplicación tiene una evolución gradual y entonces esto no genera problemas.

```
#define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
#define ceild(n,d)  (((n)<0) ? -((-(n))/(d)) : ((n)+(d)-1)/(d))
#define max(x,y)    ((x) > (y) ? (x) : (y))
#define min(x,y)    ((x) < (y) ? (x) : (y))

float procesar_matriz(float A[N][N]) {
    int i, j;
    for (i=0;i<=floord(N-1,6);i++) {
      for (j=0;j<=N-1;j++)
        calculo_simple(i,j)
    }
    for (i=ceild(N,6);i<=floord(N-1,3);i++) {
      for (j=0;j<=floord(N-1,6);j++)
        calculo_simple(i,j)
      for (j=ceild(N,6);j<=floord(5*N-6,6);j++)
        calculo_balanceado(i,j)
      for (j=ceild(5*N-5,6);j<=N-1;j++)
        calculo_simple(i,j)
    }
    for (i=ceild(N,3);i<=floord(N-2,2);i++) {
      for (j=0;j<=floord(N-1,6);j++)
        calculo_simple(i,j)
      for (j=ceild(N,6);j<=floord(N-1,3);j++)
        calculo_balanceado(i,j)
      for (j=ceild(N,3);j<=floord(2*N-3,3);j++)
        calculo_preciso(i,j)
      for (j=ceild(2*N-2,3);j<=floord(5*N-6,6);j++)
        calculo_balanceado(i,j)
      for (j=ceild(5*N-5,6);j<=N-1;j++)
        calculo_simple(i,j)
    }
    for (i=ceild(N-1,2);i<=floord(2*N-3,3);i++) {
      for (j=0;j<=floord(N-1,6);j++)
        calculo_simple(i,j)
      for (j=ceild(N,6);j<=floord(5*N-6,6);j++)
        calculo_balanceado(i,j)
      for (j=ceild(5*N-5,6);j<=N-1;j++)
        calculo_simple(i,j)
    }
    for (i=ceild(2*N-2,3);i<=floord(5*N-6,6);i++) {
      for (j=0;j<=floord(N-1,3);j++)
        calculo_simple(i,j)
      for (j=ceild(N,3);j<=floord(5*N-6,6);j++)
        calculo_balanceado(i,j)
      for (j=ceild(5*N-5,6);j<=N-1;j++)
        calculo_simple(i,j)
    }
    for (i=max(ceild(N,6),ceild(5*N-5,6));i<=N-1;i++) {
      for (j=0;j<=N-1;j++)
        calculo_simple(i,j)
    }
}
```

Figura 4.5: Código generado por ACR a partir de una cuadrícula. `ceild(a,b)` y `floord(a,b)` son respectivamente el techo y piso de la división `a/b`.

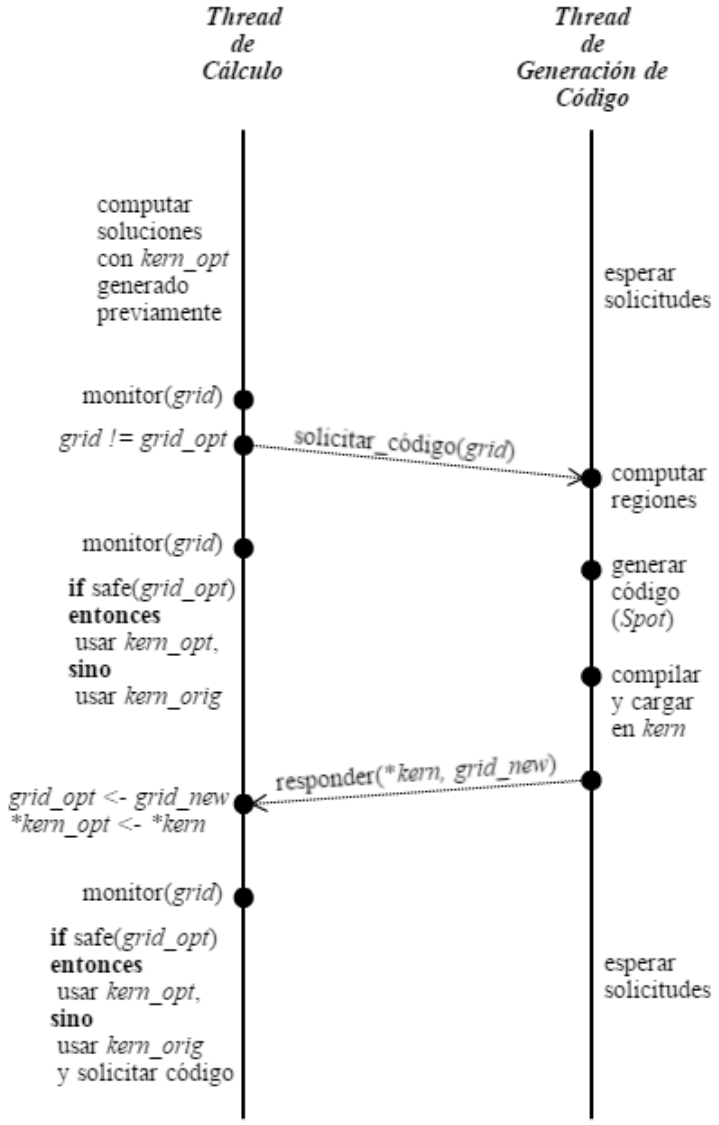


Figura 4.6: Paso de regeneración de código multi-thread.

Capítulo 5

Caso de Estudio: Simulación de Fluidos

La técnica de Refinamiento Adaptivo de Código fue implementada en un prototipo y evaluada con un código de simulación de fluidos hecho por Stam [35] para visualización de Gráficos de Computadora. Este caso de estudio es adecuado para evaluar ACR ya que es una aproximación de un fenómeno físico. El enfoque de simulación es comúnmente llamado enfoque *Euleriano* [7] y está basado en una cuadrícula con partículas fijas que describen la velocidad y la densidad del fluido en cada punto computacional del espacio. Dentro de los enfoques más populares, este conjunto de códigos nos pareció conveniente para hacer pruebas preliminares de ACR ya que también implementa una cuadrícula que, aunque es de resolución mucho mayor, tiene una correspondencia directa con nuestra cuadrícula de monitoreo. También el procesamiento está hecho en un espacio de cálculo altamente regular, y se realizan los mismos cálculos en cada punto del espacio.

En la Sección 5.1 se explicará el código simulación elegido. La simulación mantiene el estado de la cantidad de fluido y su velocidad en cada punto del espacio. Como se explica en la Sección 5.2 aplicamos la técnica en base a la cantidad de fluido en diferentes regiones del espacio: realizamos diferentes estrategias haciendo la mayor cantidad de cálculos en donde hay mayor cantidad de fluido. En la Sección 5.3 comparamos los resultados de la ejecución de un fluido en dos dimensiones optimizada por ACR y los resultados del código original. Evaluamos aspectos de mejoras en tiempo total de ejecución, cantidad cálculos realizados y precisión. También comparamos resultados de tiempo con una optimización manual y estática de la versión original que realiza la misma cantidad de cálculos y con la misma precisión que ACR, pero tiene mayor costo de control en la ejecución.

5.1. Conocimiento de Dominio de Simulación

Las leyes que gobiernan el comportamiento físico de los fluidos son las leyes de conservación, especialmente las de conservación de masa, conservación de momento lineal y conservación de energía. Las ecuaciones que describen el movimiento de los fluidos afectados por gradientes de velocidad y presión son las ecuaciones diferenciales no lineales de la Figura 5.1, llamadas ecuaciones de *Navier-Stokes* [10]. Su forma general no tiene una solución de forma cerrada, es decir no es calculable con un número finito de operaciones, por lo que para calcular resultados computacionales de un fluido en movimiento son realizadas distintas aproximaciones, dependiendo del propósito de la implementación.

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

$$\frac{\partial \rho}{\partial t} = -(\mathbf{u} \cdot \nabla)\rho + \kappa \nabla^2 \rho + S$$

Figura 5.1: Ecuaciones de Navier-Stokes, que describen el vector velocidad notado como $\mathbf{u}$ (arriba) y la densidad notada como ρ moviéndose respondiendo a la velocidad (abajo). ν es la viscosidad del fluido y κ su razón de difusión. $\mathbf{f}$ es un vector de fuerzas externas y S un conjunto de fuentes externas de densidad.

El código de simulación de fluidos de Stam utilizado para evaluar ACR está orientado a simular un fluido para aplicaciones en gráficos computacionales, cuyo principal objetivo es que la simulación posea un comportamiento visual realista y elegante. Por ello buena precisión para lograr fidelidad visual es necesaria, pero no tanta como la que se necesitaría en una simulación para propósitos científicos o de ingeniería, como por ejemplo el cálculo de las fuerzas de los gases atmosféricos al rozar con el ala de un avión, así como cálculos industriales en seguridad, performance, etc. El código de simulación utiliza un algoritmo basado en partículas fijas dispuestas regularmente en el centro de las celdas de una cuadrícula que cubre el espacio, esta forma de tratar el fluido se conoce también como enfoque *Euleriano*. Para este enfoque, la implementación de Stam posee un método de cálculo numérico estable que no diverge para pasos de tiempo largos, y genera curvas de fluido muy suaves [34]. Por otro lado, sufre en algunos casos de un fenómeno conocido en física computacional como *disipación numérica*, y se refiere a ciertos efectos secundarios que pueden producirse al utilizar soluciones numéricas para resolver ecuaciones diferenciales, y produce un efecto de difusión reduciendo la energía donde no es adecuado. Pero estos efectos no son significativos para simulaciones visualmente realistas.

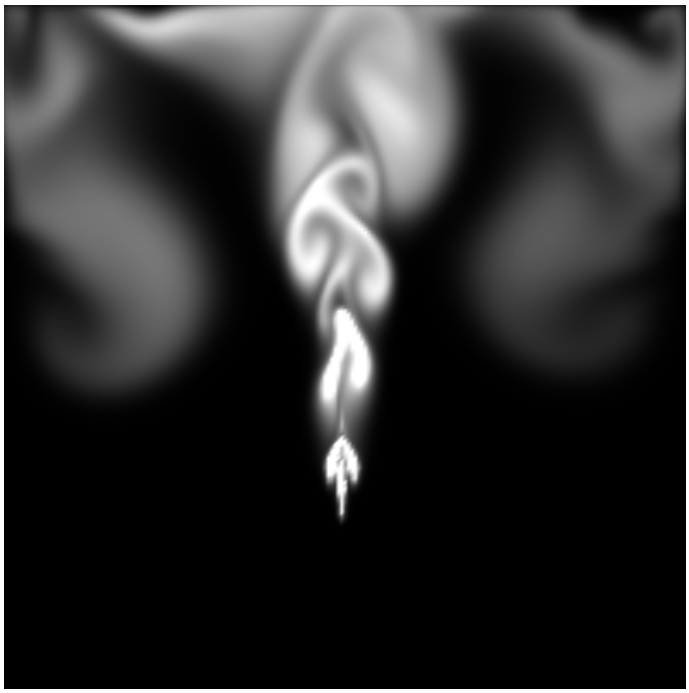


Figura 5.2: Captura de un estado de la simulación para los valores de densidad en cada punto.

En el enfoque Euleriano, el estado de un fluido es generalmente representado por su vector de velocidad y escalar de densidad en cada punto del espacio. La densidad en un punto dado representa la cantidad de fluido, es decir, cuanta materia está concentrada. Esta es la parte del fluido que podemos visualizar, como en la Figura 5.2. La velocidad indica la dirección e intensidad del movimiento del fluido en ese punto, como se muestra en la Figura 5.3. Por ejemplo, para representar los estados de un fluido en dos dimensiones, dos matrices bidimensionales (con las dimensiones del espacio) son necesarias para representar densidad y velocidad en cada punto del espacio. Una matriz de elementos escalares, que representa la densidad, y otra matriz cuyos elementos son vectores de dos componentes. Un ejemplo de la representación de la velocidad en una simulación Euleriana se puede ver en la Figura 5.3. La simulación evoluciona en un paso de tiempo constante y, en cada paso, todos los valores son actualizados.

Las dos ecuaciones de Navier-Stokes para velocidad y densidad son muy similares entre sí y poseen tres términos cada una que determinan los tres pasos fundamentales para calcular la evolución de la simulación. Ambas ecuaciones se componen de

- el *agregado de fuentes* locales o globales de fuerzas tales como la gravedad, un ventilador moviendo aire, el ala de un avión atravesando el

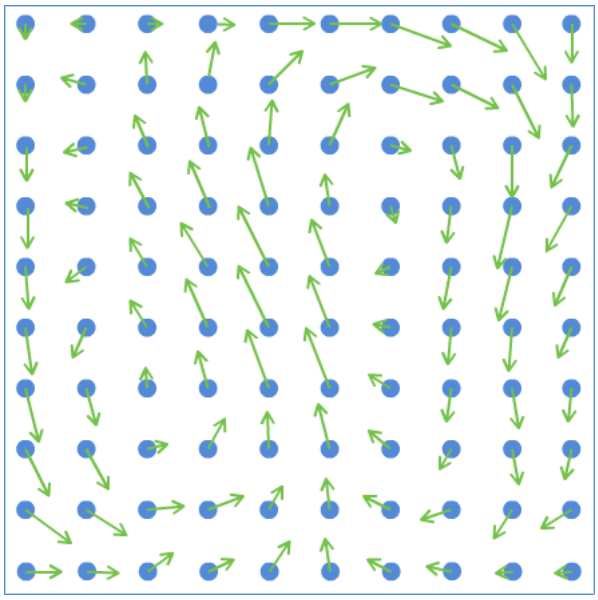


Figura 5.3: Espacio de simulación compuesto de partículas fijas. Las flechas corresponden al vector de velocidad.

espacio, etc. en el caso de la velocidad, y fuentes de densidad como humo de algún objeto quemándose, o de un líquido ingresando en el espacio, etc. para la densidad.

- la *difusión* de la velocidad de acuerdo a su viscosidad, y de la masa de acuerdo a su razón de difusión.
- y la *advección*, que es el movimiento generado por la velocidad, y afecta a la materia del fluido como también a la velocidad en sí misma. Este último efecto puede interpretarse como el arrastre del campo de velocidad y densidad en dirección de la corriente.

El agregado de fuentes, la difusión y la advección son hechos en cada paso de la simulación, interviniendo en las dos matrices de estado. La viscosidad, la razón de difusión y la longitud del paso de tiempo entre dos iteraciones contiguas de la simulación son parámetros determinantes de la simulación que queremos lograr y del tipo de fluido simulado.

El kernel numérico para el cálculo de soluciones, en la parte de la advección y de la difusión, realiza cálculos que requieren mayormente de la resolución de sistemas de ecuaciones lineales. La solución a estos sistemas lineales se realiza por medio de una relajación del algoritmo de *Gauss-Seidel*, un algoritmo de aproximación de convergencia iterativa, es decir, que mejora su solución cuantas más iteraciones realiza. En la Figura 5.4 podemos ver cómo, para cada paso, el algoritmo itera sobre el espacio una determinada cantidad de veces, cada vez obteniendo mejores aproximaciones. Una

```
void lin_solve(int N, float **x, float **x0)
{
  int i, j, k;
  for ( k=0 ; k<20 ; k++ ) {
    for ( i=1 ; i<=N ; i++ ) {
      for ( j=1 ; j<=N ; j++ ) {
        x[i][j] = nueva_iteracion(x0[i][j],
                            x[i-1][j], x[i+1][j],
                            x[i][j-1], x[i][j+1]);
      }
    }
  }
}
```

Figura 5.4: Versión simplificada del kernel original `lin_solve` que se abstrae de los cálculos numéricos y muestra las dependencias y la forma en la que itera. La matriz `x` posee la última solución en cada punto, ya sea para calcular densidad o componentes de la velocidad, y `x0` mantiene los valores de la última solución antes de empezar a iterar en un nuevo valor de `k`.

nueva aproximación se calcula partiendo de aproximaciones anteriores. En este caso, cada vez que se calcula una nueva solución en un punto se toman resultados de las soluciones previas en ese punto y en los puntos vecinos a este, como se ve en la Figura 5.5. Por esta razón, el algoritmo no puede realizar más de una iteración para mejorar la solución en un mismo punto sin iterar también sobre sus vecinos, por lo que el bucle de la Figura 5.4 que itera sobre `k` no podría intercambiarse con los que iteran sobre `i` o `j` para que sea interno. De esta manera, se calculan soluciones a lo ancho para todos los puntos y se las va mejorando de forma pareja, manteniendo las dependencias.

5.2. Aplicando ACR

De los elementos que componen el estado de la simulación, los observables y los que definen la dificultad de los cálculos en diferentes regiones del espacio son la velocidad y la densidad. Cada vez que `lin_solve` calcula soluciones para una iteración, los pasos de difusión y advección requieren mejoras iterativas de las aproximaciones. Antes de comenzar la simulación, no se sabe qué valores de densidad y velocidad va a tomar un punto del espacio en un momento dado. El kernel está diseñado para realizar un número suficientemente alto de iteraciones para asegurarse de que una aproximación de buena calidad se realice para cualquier valor de cada punto. Sin embargo,

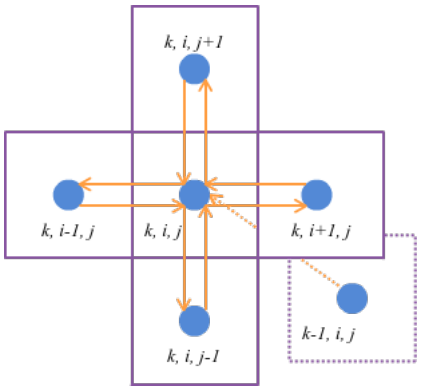


Figura 5.5: Dependencias directas de `lin_solve` para un punto que no está en un borde en el dominio de iteración.

la convergencia de la solución en cada punto es dependiente, y para algunas operaciones altamente dependiente, de los valores de velocidad y densidad en ese punto y sus vecinos directos. Esto ocurre en el sub-paso de difusión. Los cálculos en este sub-paso representan más de la mitad de los cálculos originales de de la simulación, haciendo que la optimización efectiva de esta etapa tenga un alto impacto en el tiempo total. Nuestra optimización dinámica se basa en un hecho comprobado empíricamente del comportamiento de la simulación: el kernel numérico para difusión converge con mayor rapidez donde menos densidad tiene el fluido.

A primera vista, optimizar advección no fue una opción prometedora ya que la reducción de cálculos, tanto en base a la densidad como a la velocidad, tuvo un gran impacto en la calidad de aproximación, haciendo diverger la simulación con respecto de la original en pocas iteraciones de kernel. Optimizar difusión en base a la velocidad tampoco dio buenos resultados. La alternativa con la que se lograron mejores resultados fue modificar el código de forma dinámica de acuerdo a los valores de densidad en diferentes regiones de la simulación. De esta manera, un código estático preparado para cualquier valor en todas las regiones no tiene que ser utilizado, sino que puede utilizarse código específico para cada momento de la simulación.

Cada celda de la cuadrícula monitorea una región cúbica o cuadrada de puntos del espacio de simulación. El pragma `monitor` lo utilizamos sobre la matriz de densidad. Resumimos los valores de la región tomando el valor del punto con mayor densidad, y utilizamos un `filtro` cuya clasificación divide los valores en tres categorías: cantidad alta, baja, o casi nula de fluido. Por eso nuestro pragma `monitor` será:

```
#pragma monitor density[i][j](max, filtro)
```

Usamos ACR para hacer que el algoritmo numérico realice menos iteraciones mientras se mantienen las dependencias lo mejor posible con las

iteraciones que se realizan en áreas con poco o nada de fluido. Elegimos tener tres niveles de complejidad para las regiones: en un paso de la simulación, el algoritmo optimizado hará las primeras iteraciones básicas en todo el espacio de cálculo. Luego, más iteraciones sobre una región mas restringida donde hay una mayor cantidad de fluido, y concluirá haciendo un último conjunto de iteraciones en lugares donde hay una cantidad considerable de fluido que necesita cómputos adicionales para obtener la solución con precisión suficientemente buena. Por esto, el `filtro` clasificador toma valores de punto flotante y los clasifica en tres categorías (`1`, `2` y `3`), para cada cantidad. El conjunto de valores para cada categoría así como también la cantidad de iteraciones de aproximación realizadas por `lin_solve` fue ajustado empíricamente buscando valores en los que la simulación no pierda calidad de aproximación. En la Figura 5.6 vemos la recopilación de información de densidad en la cuadrícula para tres momentos de la simulación distintos. Se utiliza gris claro y oscuro para cantidades alta y baja de fluido respectivamente. El color negro para cantidades de fluido muy bajas o nulas.

Las regiones finales computadas en la cuadrícula recubren zonas de alta o media precisión y además un margen de una celda en sus alrededores, cuando la zona contigua es de menor precisión. Esto se realiza para darle tiempo al thread generador de código para tener lista una nueva versión del kernel en el caso de que una porción del fluido se traslade rápidamente, modificando las áreas de complejidad. De no prever este caso, muchos pasajes al código original se generan por no tener listas optimizaciones adecuadas a tiempo, ocasionando gran cantidad de tramos de baja eficiencia. Cuanto más rápidos sean los tiempos de generación de código optimizado, mas ajustadas podrán ser las regiones.

Nuestra estrategia no utiliza bloques de código alternativos, sino que omite iteraciones en zonas de convergencia media o rápida, por esta razón los pragmas de alternativa son solamente del tipo `parametro` y ajustan la cota de `k`. A diferencia del ejemplo de la Figura 4.1, no contemplamos transformaciones en zonas estáticas, por lo que nuestros pragmas de estrategia son exclusivamente pragmas `direct`. Los pragmas `alternative` son

```
#pragma ACR alternative bajo(parametro, MAX=precision_baja)
#pragma ACR alternative medio(parametro, MAX=precision_media)
#pragma ACR alternative alto(parametro, MAX=precision_alta)
```

y cada uno cambia el `parametro` `MAX`, que es el límite de iteración del bucle que itera sobre `k`. Los pragmas `strategy` son

```
#pragma ACR strategy direct(1, bajo)
#pragma ACR strategy direct(2, medio)
#pragma ACR strategy direct(3, bajo)
```

que asignan a cada zona clasificada por `filtro` una alternativa. El pragma `grid` también está ajustado empíricamente para que la cuadrícula tenga la resolución que mejores resultados tiene para nuestro caso de estudio. Una versión simplificada del código con los pragmas ACR es similar a la de la Figura 4.1, con la diferencia de que el ejemplo de la Figura utiliza una estrategia `zone` y una alternativa `code`, que en este caso de estudio no utilizamos.

Dado que el agregado de fuentes como fuerzas o materia es un cálculo mucho menos complejo que el resto, el kernel original se enfoca en computar soluciones para difusión y advección en simulaciones de todas las posibles características de viscosidad, razón de difusión, diferencia de tiempo por iteración, etc. Una característica importante de las optimizaciones adaptivas y dinámicas generadas con compilación poliédrica, es que el kernel compilado estáticamente debe estar preparado para cualquier valor de densidad y velocidad en cada punto. Modificar el código del kernel de forma estática para ahorrar cálculos según la densidad en un punto y respetando el orden original de los cálculos que no se evitan tiene mayor costo de control.

5.3. Resultados Experimentales

La simulación ejecutada está configurada para un fluido en un espacio bidimensional de 200 por 200 partículas. Se compararon los códigos que se basan en aproximaciones ACR con el código original y también con una versión optimizada a mano que imita la estrategia de ACR sin generar el código dinámicamente. Los aspectos que observamos fueron eficiencia, ahorro en cálculos y precisión en un rango de iteraciones de la simulación. Durante la simulación, el fluido es inyectado regularmente en el medio del espacio de simulación en conjunto con una fuerza direccional hacia arriba que posee siempre la misma intensidad para que el fluido tome velocidad y se mueva en el espacio. El caso de prueba tiene diferentes tipos de regiones que evolucionan en el tiempo.

El montaje del experimento se realizó sobre un sistema quad-core Intel Core i7-2630QM con 8GB de memoria. Todos los códigos, excepto el kernel `lin_solve` generado dinámicamente, fueron compilados usando GCC 4.8 con la opción `-O3`, que compila las mejores versiones del código original tanto como el optimizado a mano. `lin_solve` fue compilado usando Tiny C Compiler (TCC), 0.9.25 para minimizar el tiempo de compilación dinámica para ACR. La resolución de la cuadrícula fue fijada en 10 por 10 celdas como una buena resolución de acuerdo a las dimensiones del espacio de cálculo.

La comparación de tiempos totales de ejecución en diferentes momentos de la simulación para el código original, el código original optimizado usando ACR y la versión optimizada manualmente se muestra en la Figura 5.7.

El código manualmente optimizado se puede ver en el Apéndice A.3 y corresponde a una versión con la misma estrategia de aproximación que

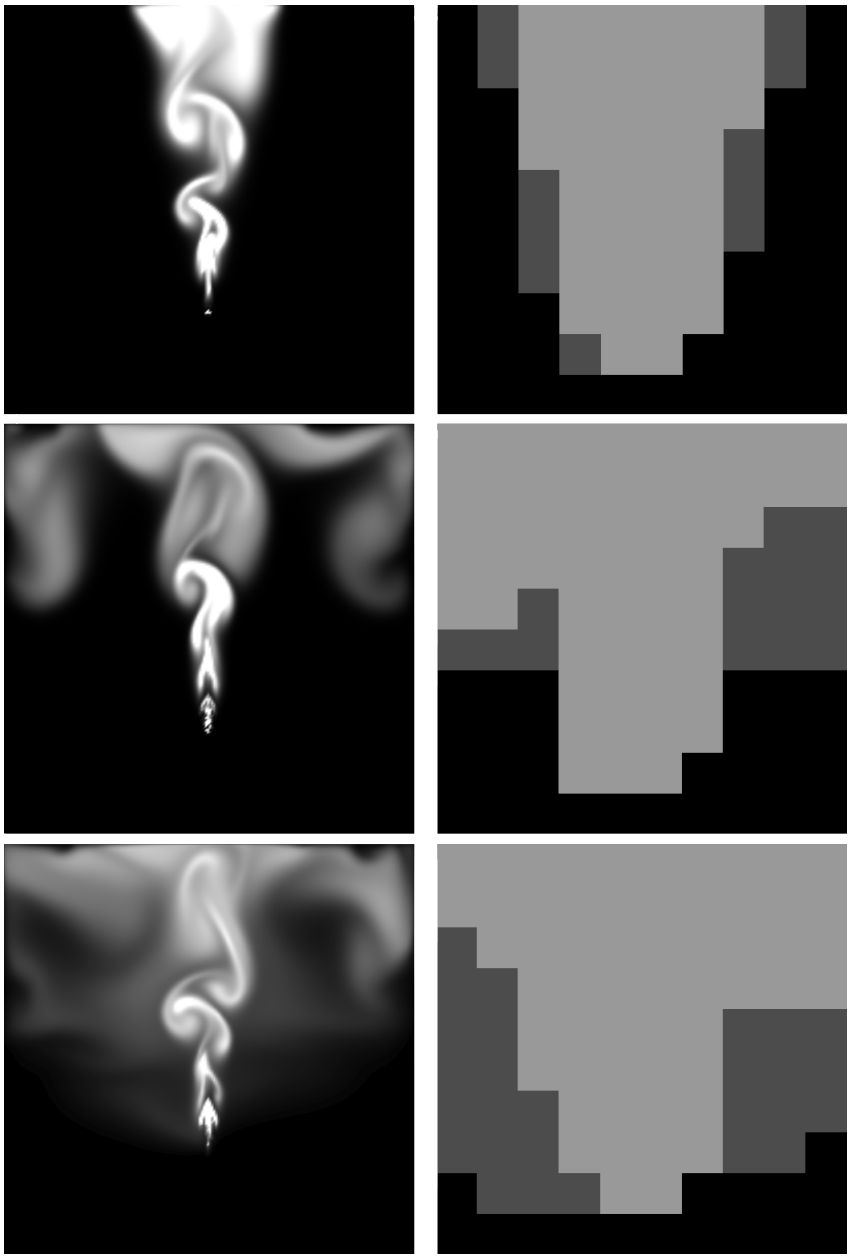
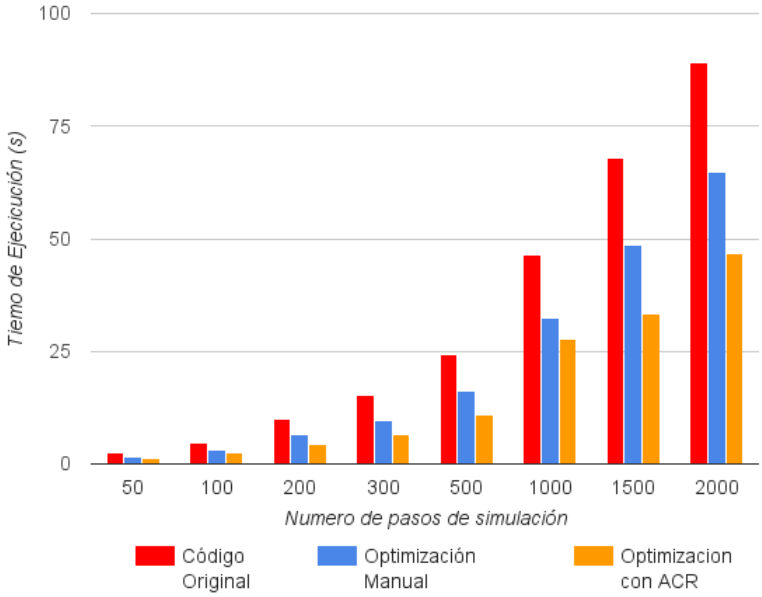


Figura 5.6: El estado de la densidad por punto y lo que recolecta la cuadrícula en las iteraciones 208, 982 y 1947.

ACR, incluyendo la misma computación de la cuadrícula de estado y selección de la alternativa más apropiada de aproximación para cada celda de la cuadrícula en tiempo de ejecución. Sin embargo este código es completamente generado estáticamente y la selección de alternativas es hecha por un switch integrado en el espacio de iteración original, que es enteramente visi-



iteraciones	Original con -O3	Opt. Manual	Opt. con ACR
50	2,3	1,6	1,1
100	4,8	3,2	2,5
200	9,9	6,5	4,4
300	15,3	9,7	6,6
500	24,3	16,2	11,0
1000	46,5	32,4	27,6
1500	68,0	48,6	33,4
2000	89,1	64,8	46,7

Figura 5.7: Comparación de tiempos de ejecución en segundos de las 3 versiones probadas de la simulación.

tado, por lo que acarrea una alta sobrecarga en control. Como mencionamos anteriormente la Sección 5.2, esto ocurre porque el kernel itera a lo ancho para respetar las dependencias. Mejores aproximaciones estáticas y con menor costo de control pueden lograrse, por ejemplo, iterando sobre conjuntos de celdas, posiblemente implementados a través de listas enlazadas. Este enfoque permite iterar solamente sobre el espacio de iteración optimizado y no realizar pruebas costosas, pero tiene la gran desventaja de no respetar el orden original de los cálculos que no se evitan.

ACR genera código optimizado específicamente para el estado actual de la cuadrícula que recorre solamente el espacio de iteración necesario sin necesidad de pruebas costosas en bucles internos, como la versión generada del Apéndice A.2. Ésta tiene un costo de control insignificante, ya que los dominios de iteración ya fueron computados en la generación poliédrica de

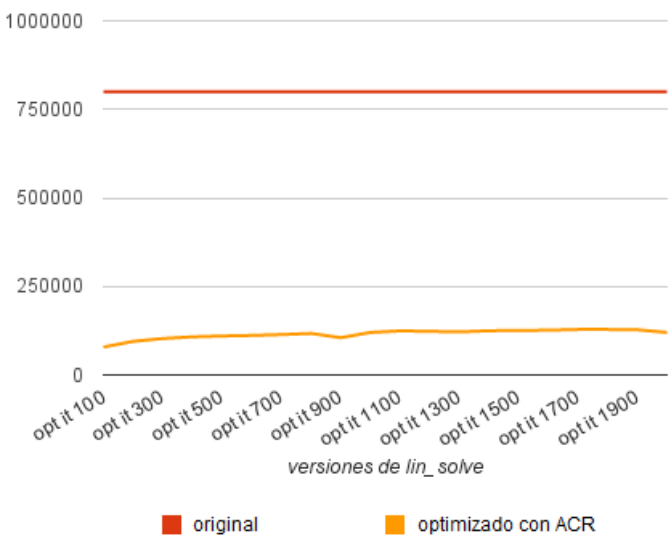
código. Los resultados de eficiencia sobre el total de cálculos muestran aumentos de velocidad que impactan en reducciones de tiempo en promedio del 52 % para el código original y 29 % para el código manualmente optimizado.

El código correspondiente a difusión se ejecuta tres veces: una para hacer difusión de materia y otras dos para cada componente de la velocidad. Los ahorros en cálculos de difusión para un paso de la simulación con respecto al código original se muestran en la Figura 5.8. Los resultados muestran ahorros en cálculos muy significativos comparados con el número de cálculos de la difusión original, desde 84 a 90 %. Los cálculos de difusión son más de la mitad de los cálculos totales. Debido a la heterogeneidad de operaciones realizadas en la simulación, es difícil determinar precisamente qué porcentaje de estas son de difusión, pero estimamos que están entre el 52 y el 60 % de los cálculos totales. La simulación optimizada tiene mejor eficiencia al principio porque hay menos cantidad de fluido y los cálculos hechos no son grandes en ese momento.

Los ahorros de cálculos se logran por la reducción de iteraciones para las zonas con muy poco fluido, una cantidad moderada de fluido e incluso donde las cantidades de fluidos son las más altas posibles para el fluido elegido. Este último ahorro se logra porque el código original no tiene en cuenta que la convergencia de `lin_solve` en el paso de difusión varía dependiendo de los parámetros de viscosidad, razón de difusión y paso de tiempo de la simulación. Para los valores de los parámetros elegidos el algoritmo converge mas rápido que para el caso general que tiene en cuenta el kernel, incluso donde hay gran cantidad de fluido. Es importante aclarar que no tenemos en cuenta los cálculos de generación de código, compilación y linkeo realizados paralelamente en el Thread de Generación de Código.

Resultados de precisión son mostrados en la Figura 5.9. Se midieron la diferencia de densidad por cada partícula en momentos fijos de ambas simulaciones, con el original como referencia. Los valores de densidad del fluido varían mayormente entre 0 y 1, pero en el punto de inyección de fluido y una pequeña porción que lo rodea, hay picos de hasta 20 en el momento de la inyección y se van estabilizando en algunas iteraciones posteriores hasta no ser mayores a 1. Observamos que luego de 2000 iteraciones, una instancia avanzada de la simulación y donde la forma del fluido se estabiliza por un período largo, la diferencia máxima está cerca del 7 % y el promedio de las diferencias es menor a 2 % teniendo en cuenta de que se eliminaron el 85 % de los cálculos en difusión.

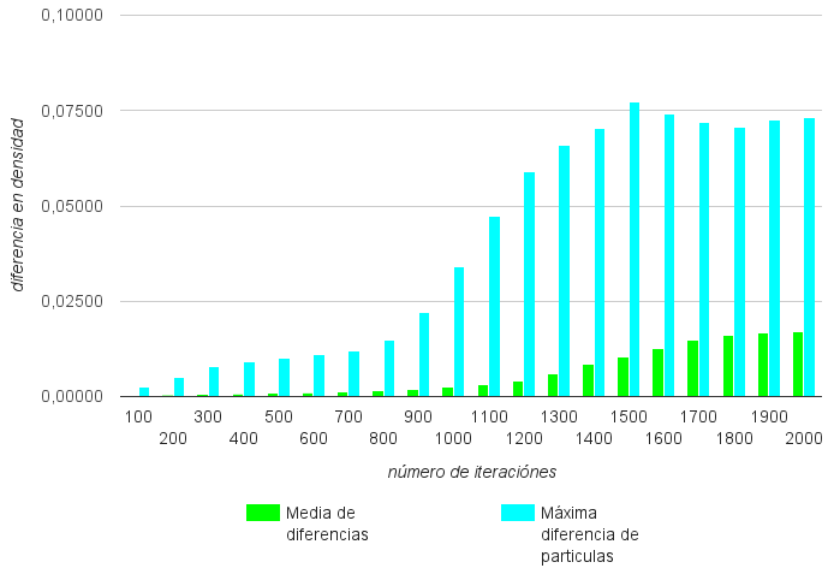
Los resultados muestran ahorros de cálculos considerables y mejora en la performance mientras se mantiene buena precisión. La mejora de tiempo de factor 2 obtenida es comparable con resultados obtenidos usando la técnica de *loop perforation* [33] que también elimina cálculos, pero para iteraciones completas. Sin embargo, construir conocimiento de dominio específico y con monitoreo dinámico en vez de estático hace a ACR más preciso y adaptable para tipos diferentes de simulación. De todas maneras, extensiones y me-



n. de iteraciones	n. de cálculos con ACR	cálculos ahorrados
100	78800	90.15 %
200	94800	88.15 %
300	102800	87.15 %
500	110000	86.25 %
1000	120000	85.00 %
1500	125600	84.30 %
2000	120000	85.00 %

Figura 5.8: Número de cálculos realizados por ACR y el ahorro con respecto los cálculos originales. El gráfico muestra la cantidad de cálculos de aproximación realizados (eje vertical) para distintas versiones optimizadas de `lin_solve` para una aplicación del paso de difusión. La linea roja es referencia de la cantidad de cálculos de la versión original.

joras técnicas son todavía necesarias para que ACR convierta el ahorro de cálculos en performance. El enfoque de generación poliédrica dinámica de código también supera en performance a la optimización manual, la cual es código modificado con respecto al bucle de simulación original, mientras ACR preserva el programa original.



iteraciones	media de las diferencias	máxima diferencia entre partículas
100	0.0001	0.0025
200	0.0003	0.0055
300	0.0005	0.0077
500	0.0009	0.0100
1000	0.0025	0.0340
1500	0.0105	0.0774
2000	0.0171	0.0732

Figura 5.9: Diferencia de densidades por partícula, en promedio, y la máxima diferencia entre dos partículas encontrada en todo el espacio.

Capítulo 6

Conclusiones y Trabajo a Futuro

6.1. Conclusiones

En este trabajo, se introduce el Refinamiento Adaptivo de Código, una nueva técnica de compilación para mejorar eficiencia a costo de precisión.

Comenzando por un programa de referencia e información de alto nivel relacionada con cálculos aproximados suministrada por el usuario, genera automáticamente un programa que adapta continuamente la estrategia de optimización a la más apropiada para diferentes regiones del espacio de cálculo. ACR suministra un conjunto único de características para ofrecer eficiencia, precisión y flexibilidad. La eficiencia es suministrada por basarnos en el estado del arte en técnicas de generación poliédrica de código para generar código optimizado y aprovechando la arquitectura multinúcleo con un thread dedicado a la generación de código para minimizar sobrecarga en tiempo de ejecución. La precisión es preservada lo más posible basándonos en una estrategia dinámica que realiza cálculos precisos solamente donde son necesarios y preservando el orden original de los cálculos. Finalmente, la flexibilidad es obtenida a través del diseño de un simple pero poderoso conjunto de pragmas para conducir la estrategia de aproximación, permitiendo al usuario enfocarse en un kernel simple e ideal mientras la aproximación es manejada por ACR.

Para evaluar ACR, construimos un prototipo y lo usamos en una aplicación existente de simulación de fluidos. Los resultados experimentales demuestran mejoras significativas en performance (una aceleración promedio de 2.06) con una baja diferencia en precisión (por cada partícula, menos de 2 % en promedio y 7 % como máximo). También mostramos que ACR supera en eficiencia a una aproximación manual que realizaría la misma aproximación dinámica pero que, por construcción, no puede tener un control de flujo optimizado que puede sólo ser logrado con generación de código en tiempo

de ejecución.

Este trabajo principalmente aporta una nueva forma de optimización que debe ser evaluada más ampliamente. Se basa en la transformación dinámica y automática de código, de forma transparente al usuario y está destinada a aplicaciones de cómputo por aproximación. Motivadores resultados preliminares abren la incógnita para seguir investigando hasta qué punto es efectivo para arquitecturas multinúcleo el invertir en costo computacional dedicado a threads paralelos en generación de código, inherente a la transformación dinámica, por sobre optimizaciones manuales y estáticas que por no conocer los estados de la ejecución en el momento de compilación, pueden llegar a tener mayor costo de control y no son automáticos.

Una cuestión interesante a notar en ACR es su capacidad de combinarse con optimizaciones estáticas, aplicándolas de la misma manera que lo hace la técnica con el código original cuando éste es más adecuado para ser ejecutado. De esta manera, para arquitecturas multinúcleo ACR no empeoraría sino que igualaría o mejoraría la eficiencia en tiempo de las optimizaciones estáticas.

6.2. Trabajo a Futuro

ACR se encuentra en una fase inicial y muchos estudios, mejoras y extensiones son posibles. Hay dos direcciones principales para esto. Una es la evaluación de la técnica con más de casos de estudio y comparación con optimizaciones en estos casos, para tener resultados representativos para su dominio de aplicación. La otra, que se encuentra muy relacionada con la primera, es el mejoramiento de los algoritmos que componen la técnica, principalmente en el monitoreo de estados, la generación de código y la calidad de las aproximaciones generadas.

En principio, se puede hacer una mejor evaluación del potencial de optimización de ACR para el caso de estudio tratado en el Capítulo 5. Se pueden hacer pruebas para incluir ahorros en advección y/o aprovechando el estado de la matriz de velocidad, como por ejemplo conocer si los cambios en los valores de velocidad afectan la rapidez de convergencia del kernel. También, se pueden estudiar más variantes de la implementación de Stam para diferentes disposiciones del fluido, como el ingreso de fuentes y fuerzas globales o locales, en otras direcciones e intensidades, o con diferentes parámetros como la viscosidad, el tiempo de cada paso de simulación, la razón de difusión o la cantidad de partículas utilizadas para la simulación. La implementación también permite la simulación de un fluido en tres dimensiones. Se puede evaluar la eficiencia de ACR en este tipo de simulación con una cuadrícula tridimensional.

El código de ACR, principalmente en el suministro de información de alto nivel, se encuentra todavía bastante acoplado a la implementación de

Stam. Hay que implementar el análisis de los pragmas para generalizar la técnica a una mayor variedad de aplicaciones.

Otros estudios para evaluar la aplicabilidad de la técnica técnica tienen que ser hechos para otras implementaciones en simulación de fluidos con el enfoque para partículas fijas o más comúnmente llamado "basado en cuadrícula". También evaluar qué tan aplicable es en otros enfoques, como simulación basada en partículas móviles o Lagrangiana [25], o fuera de la simulación, para otro tipo de aplicaciones, cuyo espacio de soluciones tenga diferentes zonas de complejidad, como podría ocurrir en el procesamiento de imágenes, video en tiempo real, aplicaciones de visión o de señales para sensores, etc. Para cada familia de códigos definida por un caso de estudio donde ACR sea viable, hay que realizar comparaciones con el estado del arte en técnicas de optimización automáticas y manuales existentes.

La evaluación realizada anteriormente debe ser hecha con versiones más y más eficientes de la técnica. Para para ello, varios cambios pueden realizarse. En este momento, la generación de código se realiza con TCC, un compilador de C liviano y rápido. Para reducir más el tiempo de generación, esta tarea puede realizarse mas rápido con compilación JIT. También puede aumentarse la cantidad de threads de generación de código, paralelizando las tareas de compilación poliédrica y la de compilación a código de máquina, por ejemplo, para tener más disponibilidad de threads esperando nuevas solicitudes de generación mientras otros trabajan en la compilación a código máquina. Esto puede ser particularmente útil en casos que se quieren generar varias versiones de kernel con distintas características para una solicitud: versiones más eficientes pero más ajustadas a las zonas reales, u otras menos eficientes pero más abarcativas el caso de que algunas región cambie mucho para no tener que pasar a utilizar el código original. Distintos niveles de compromiso pueden ser logrados.

La sobrecarga de tiempo de monitoreo se puede reducir dedicando un nuevo thread que realice esta tarea de forma paralela. Por otro lado, si deseamos mantener el monitoreo secuencial, otro enfoque posible sería estudiar la posibilidad de embeber el monitoreo en la realización de cálculos de kernel aprovechando la localidad de datos. Esto se podría realizar aprovechando el potencial de análisis de la compilación poliédrica, analizando en su representación y agregando sensores de valor en los statements que corresponden a las últimas escrituras de la matriz de puntos monitoreados. También se puede marcar una celda como *dirty* cuando los puntos de la celda cambian lo suficiente como para que ésta cambie de estrategia de aproximación, y de esta manera solo actualizar las celdas que están marcadas.

Para la flexibilización de la cuadrícula, mejoras podrían lograrse probando el uso de celdas que puedan adoptar otras formas geométricas regulares, como por ejemplo triángulos para lograr regiones mejor descriptas a un bajo costo computacional. Algo más ambicioso sería estudiar hasta qué punto la cuadrícula se puede refinar o engrosar para regiones que requieren ob-

servación más o menos detallada respectivamente. Como por ejemplo de la manera que lo hace AMR [4] para colisiones hidrodinámicas y otro tipo de simulaciones físicas, o mimetizando otros métodos adaptivos a la generación de código eficiente de forma dinámica.

Si bien uno de los aspectos a los que apunta el modelo poliédrico es la paralelización, nuestro trabajo solamente lo aborda dividiendo las tareas en dos threads. Como mencionamos arriba, existen maneras de paralelizar en mayor medida la generación de código incluyendo threads de monitoreo y asignando mas threads a tareas independientes del proceso de generación de código. Sin embargo, no se trata la paralelización ni la utilización de GPU en el cálculo principal del kernel a optimizar. Para ello, se puede optar por integrar herramientas de compilación poliédrica que de manera automática expongan y aprovechen el paralelismo luego de realizar la transformación de dominios y el cambio de cálculos alternativos. De esta manera, se puede generar código paralelo, para CPU y/o GPU. Por ejemplo, se podría utilizar PPCG [1] [39], una herramienta que genera código para arquitectura CUDA.

Otro enfoque sería el de utilizar información suministrada por el programador para explicitar paralelismo en el kernel. De esta manera, se podrían agrupar cómputos asociados al mismo statement y ejecutarlos como un kernel de CUDA. Los conjuntos de cómputos correspondientes a un mismo statement son los asociados a una misma región de complejidad, o a distintas regiones de complejidad pero que difieran solamente en ciertos parámetros. Para kernels en los que sea conveniente su total ejecución en GPU, el estado de la ejecución también debería ser alojado en la GPU para minimizar la latencia de acceso. Para mantener la localidad de acceso a memoria GPU, se debe tener en cuenta algún criterio para elegir cálculos que accedan a lugares próximos del estado. Por ejemplo, puede que a los cómputos asociados celdas de monitoreo contiguas sea conveniente situarlos en el mismo multiprocesador y los cómputos asociados a una misma celda en el mismo *warp*. Las tareas de generación de código se deben mantener en el CPU, ya que no son aptas para GPU. Suponiendo que el estado de ejecución se encuentra en memoria GPU, la información que debe transmitirse entre ésta y la memoria del CPU es la cuadrícula de estado y las versiones del kernel generadas en GPU. El monitoreo del estado e ingreso de valores en la cuadrícula también es muy paralelizable ya que pueden lanzarse muchos threads que resuman información de distintas zonas de la cuadrícula.

Por ejemplo, para nuestro caso de estudio del Capítulo 5, existen implementaciones para GPU que podrían adaptarse a ACR [18]. Se puede paralelizar cada iteración a lo ancho del kernel de convergencia iterativa, asignando un thread CUDA para el cálculo de cada punto del espacio. El estado de la simulación se mantiene en texturas para minimizar latencia de memoria. La implementación de Stam utiliza los últimos resultados disponi-

[1] `https://github.com/Meinersbur/ppcg`

bles de los vecinos para calcular la solución en un punto. Para que un punto no tenga dependencia con los vecinos en la iteración actual, deben utilizarse valores de la solución de la iteración anterior. Esto hace que se necesite una mayor cantidad de iteraciones para la convergencia del algoritmo, pero el alto grado de paralelización GPU hace que cada iteración se realice en mucho menos tiempo, por lo que en balance es conveniente.

En nuestra optimización con ACR, se puede crear un pragma que explicite que dos elementos con el mismo valor de `k` pueden calcularse de forma independiente. Luego, el código se genera para realizar las pasadas calculando cada solución puntual en paralelo, es decir, solamente respetando la secuencialidad de `k`. En valores de `k` bajos, esto se realiza para todos los puntos, mientras que en otros mas altos, en el subconjunto de puntos de regiones de mayor complejidad. En cuanto al mapeo de threads con los cálculos, se debería intentar mantener una buena cercanía (mismo *warp*, o mismo multiprocesador) de kernels que trabajan con datos próximos. Puede que los cálculos pertenecientes a una misma celda aumenten la localidad de los accesos, en ese caso, debería intentar ubicar elementos de las mismas celdas en threads en la misma disposición, y respetar lo más posible la posición de las celdas entre sí. Si tenemos regiones de interés con distintos bloques de cálculos, se deberían asignar distintos kernels para cada una. La paralelización de los cómputos rompería con la preservación del orden original del los cálculos, que es respetada por ACR, pero solamente donde el programador lo explicite.

Bibliografía

[1] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI'10 Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 198–209, Toronto, Canada, June 2010.

[2] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[3] Cédric Bastoul. *Contributions to High-Level Program Optimization.* Habilitation Thesis. Paris-Sud University, France, December 2012.

[4] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *J. Comput. Phys.*, 82(1):64–84, May 1989.

[5] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, pages 132–146. Springer, 2008.

[6] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI'08 ACM Conf. on Programming language design and implementation*, Tucson, USA, June 2008.

[7] Robert Bridson. *Fluid simulation for computer graphics.* CRC Press, 2015.

[8] Kolja Brix, Silvia Sorana Melian, Siegfried Müller, and Gero Schieffer. Parallelisation of multiscale-based grid adaptation using space-filling curves. *ESAIM*, 29:108–129, December 2009.

[9] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David M. Brooks. HELIX-UP: Relaxing program semantics to unleash parallelization. San Francisco, USA, February 2015.

[10] Alexandre Joel Chorin, Jerrold E Marsden, and Jerrold E Marsden. *A mathematical introduction to fluid mechanics*, volume 3. Springer, 1990.

[11] Albert Cohen, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache. Facilitating the search for compositions of program transformations. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 151–160. ACM, 2005.

[12] Paul Feautrier. Parametric integer programming. *Revue française d'automatique, d'informatique et de recherche opérationnelle. Recherche opérationnelle*, 22(3):243–268, 1988.

[13] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[14] Paul Feautrier. Some efficient solutions to the affine scheduling problem: one dimensional time. *Intl. Journal of Parallel Programming*, 21(5):313–348, october 1992.

[15] Paul Feautrier. Array expansion. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 99–111. ACM, 2014.

[16] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[17] Martin Griebl, Christian Lengauer, and Sabine Wetzel. Code generation in the polytope model. In *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, pages 106–111. IEEE, 1998.

[18] Mark J Harris. Fast fluid dynamics simulation on the gpu. In *SIGGRAPH Courses*, page 220, 2005.

[19] Henry Hoffmann, Sasa Misailovic, Stelios Sidiroglou, Anant Agarwal, and Martin Rinard. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2009-042, MIT, September 2009.

[20] François Irigoin and Remi Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–329. ACM, 1988.

[21] Alexandra Jimborean, Philippe Clauss, Jean-François Dollinger, Vincent Loechner, and Juan Manuel Martinez Caamaño. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *International Journal of Parallel Programming*, 42(4):529–545, 2014.

[22] Richard M Karp, Raymond E Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM (JACM)*, 14(3):563–590, 1967.

[23] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers' 95., Fifth Symposium on the*, pages 332–341. IEEE, 1995.

[24] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 127–138, Seattle, USA, June 2013.

[25] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 154–159. Eurographics Association, 2003.

[26] Siegfried Müller and Youssef Stiriba. Fully adaptive multiscale schemes for conservation laws employing locally varying time stepping. *J. of Scientific Computing*, 30(3), 2007.

[27] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. Graphite: Polyhedral analyses and optimizations for gcc. In *proceedings of the 2006 GCC developers summit*, page 2006. Citeseer, 2006.

[28] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press.

[29] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 4–13. ACM, 1991.

[30] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, 2000.

[31] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *MICRO'13 Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 13–24, Davis, California, December 2013.

[32] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *PLDI'11 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, San Jose, California, USA, June 2011.

[33] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *FCE'11 Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 124–134, Szeged, Hungary, September 2011.

[34] Jos Stam. Stable fluids. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 121–128. ACM Press/Addison-Wesley Publishing Co., 1999.

[35] Jos Stam. Real-time fluid dynamics for games. In *Proceedings of the game developer conference*, volume 18, page 25, 2003.

[36] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *International Conference on Compiler Construction*, pages 185–201. Springer, 2006.

[37] Nicolas Vasilache, Cédric Bastoul, Albert Cohen, and Sylvain Girbal. Violated dependence analysis. In *Proceedings of the 20th annual international conference on Supercomputing*, pages 335–344. ACM, 2006.

[38] Sven Verdoolaege. *isl*: An integer set library for the polyhedral model. In *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software*, pages 299–302, Kobe, Japan, September 2010.

[39] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

[40] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In *ACM Sigplan Notices*, volume 26, pages 30–44. ACM, 1991.

[41] Jingling Xue. *Loop Tiling for Parallelism*. Springer Science & Business Media, 2000.

Apéndice

Apéndice A

Kernels Originales y Optimizados

A.1. Kernel `lin_solve` original

```
void lin_solve_opt( int N, float **x, float **x0, float a, float c)
{
    int i, j, k;
    for ( k=0 ; k<20 ; k++ ) {
        for ( i=1 ; i<=N ; i++ ) {
            for ( j=1 ; j<=N ; j++ ) {
                  x[i][j] = (x0[i][j] + a * (x[i-1][j] + x[i+1][j] +
                                             x[i][j-1] + x[i][j+1] ))/c;
            }
        }
    }
}
```

A.2. Kernel `lin_solve` generado por ACR luego de 982 iteraciones

```
void lin_solve_opt( int N, float **x, float **x0, float a, float c)
{
 int k, i, j;
 if (N >= 1) {
   for (i=1;i<=N;i++) {
     for (j=1;j<=N;j++) {
       x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
     }
   }
   if (N >= 201) {
     for (k=1;k<=2;k++) {
       for (i=1;i<=60;i++) {
         for (j=82;j<=N;j++) {
           x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
```

```
      }
    }
    for (i=61;i<=120;i++) {
      for (j=22;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=121;i<=140;i++) {
      for (j=42;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=141;i<=200;i++) {
      for (j=82;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=201;i<=N;i++) {
      for (j=1;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
  }
}
if ((N >= 141) && (N <= 200)) {
  for (k=1;k<=2;k++) {
    for (i=1;i<=60;i++) {
      for (j=82;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=61;i<=120;i++) {
      for (j=22;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=121;i<=140;i++) {
      for (j=42;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=141;i<=N;i++) {
      for (j=82;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
  }
}
if ((N >= 121) && (N <= 140)) {
  for (k=1;k<=2;k++) {
    for (i=1;i<=60;i++) {
      for (j=82;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=61;i<=120;i++) {
      for (j=22;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=121;i<=N;i++) {
      for (j=42;j<=N;j++) {
```

```
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
  }
}
if ((N >= 82) && (N <= 120)) {
  for (k=1;k<=2;k++) {
    for (i=1;i<=60;i++) {
      for (j=82;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=61;i<=N;i++) {
      for (j=22;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
  }
}
if ((N >= 61) && (N <= 81)) {
  for (k=1;k<=2;k++) {
    for (i=61;i<=N;i++) {
      for (j=22;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
  }
}
if ((N >= 61) && (N <= 101)) {
  for (i=61;i<=N;i++) {
    for (j=22;j<=N;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
}
if (N >= 201) {
  for (i=1;i<=40;i++) {
    for (j=102;j<=N;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
  for (i=41;i<=60;i++) {
    for (j=122;j<=N;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
  for (i=61;i<=120;i++) {
    for (j=22;j<=N;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
  for (i=121;i<=140;i++) {
    for (j=42;j<=N;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
  for (i=141;i<=160;i++) {
    for (j=142;j<=N;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
  for (i=161;i<=200;i++) {
```

```
      for (j=162;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=201;i<=N;i++) {
      for (j=1;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
  }
  if ((N >= 162) && (N <= 200)) {
    for (i=1;i<=40;i++) {
      for (j=102;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=41;i<=60;i++) {
      for (j=122;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=61;i<=120;i++) {
      for (j=22;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=121;i<=140;i++) {
      for (j=42;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=141;i<=160;i++) {
      for (j=142;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=161;i<=N;i++) {
      for (j=162;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
  }
  if ((N >= 142) && (N <= 161)) {
    for (i=1;i<=40;i++) {
      for (j=102;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=41;i<=60;i++) {
      for (j=122;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=61;i<=120;i++) {
      for (j=22;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
    }
    for (i=121;i<=140;i++) {
      for (j=42;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
```

```
  }
  for (i=141;i<=min(160,N);i++) {
    for (j=142;j<=N;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
}
if ((N >= 122) && (N <= 141)) {
  for (i=1;i<=40;i++) {
    for (j=102;j<=N;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
  for (i=41;i<=60;i++) {
    for (j=122;j<=N;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
  for (i=61;i<=120;i++) {
    for (j=22;j<=N;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
  for (i=121;i<=min(140,N);i++) {
    for (j=42;j<=N;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
}
if (N == 121) {
  for (i=1;i<=40;i++) {
    for (j=102;j<=121;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
  for (i=61;i<=120;i++) {
    for (j=22;j<=121;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
  for (j=42;j<=121;j++) {
    x[121][j] = (x0[121][j] + a*(x[(121)-1][j]+x[121+1][j]+x[121][(j)-1]+x[121][j+1]))/c;
  }
}
if ((N >= 102) && (N <= 120)) {
  for (i=1;i<=40;i++) {
    for (j=102;j<=N;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
  for (i=61;i<=N;i++) {
    for (j=22;j<=N;j++) {
      x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
    }
  }
}
if (N >= 202) {
  for (k=4;k<=19;k++) {
    for (i=1;i<=200;i++) {
      for (j=202;j<=N;j++) {
        x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
      }
```

```
      }
      for (i=201;i<=N;i++) {
        for (j=1;j<=N;j++) {
          x[i][j] = (x0[i][j] + a*(x[(i)-1][j]+x[i+1][j]+x[i][(j)-1]+x[i][j+1]))/c;
        }
      }
    }
  }
  if (N == 201) {
    for (k=4;k<=19;k++) {
      for (j=1;j<=201;j++) {
        x[201][j] = (x0[201][j] + a*(x[(201)-1][j]+x[201+1][j]+x[201][(j)-1]+x[201][j+1]))/c;
      }
    }
  }
 }
}
```

A.3. Optimización manual de `lin_solve`

```
int iter_from_level(int lev) {
    switch (lev) {
        case 2:
            return 4;
            break;
        case 1:
            return 3;
            break;
        case 0:
            return 1;
            break;
        default:
            return 20;
            break;
    }
}

void lin_solve( int N, float **x, float **x0, float a, float c)
{
    int i, j, k;
    for ( k=0 ; k<20 ; k++ ) {
        for ( i=1 ; i<=N ; i++ ) {
            gi = (i-1)/slot_size;
            for ( j=1 ; j<=N ; j++ ) {
                gj = (j-1)/slot_size;
                if (k < iter_from_level(grid[gi][gj])) {
                        x[i][j] = (x0[i][j]
                                + a * (x[i-1][j] + x[i+1][j]
```

```
                                        + x[i][j-1] + x[i][j+1])) / c;
                }
            }
        }
    }
}
```