# IMPLEMENTACIÓN DE CÓDIGO CFD PARALELO PARA FLUJO COMPRESIBLE

S. C. Chan Chang, G. Weht, M. Montes
Departamento Mecánica Aeronáutica, Facultad de Ingeniería,
Instituto Universitario Aeronáutico
Av. Fuerza Aérea 6500 (X5010JMX) Córdoba, Argentina.
**e-mail: gweht@iua.edu.ar**

## RESUMEN

*Se presentan implementaciones de códigos paralelos para la solución de las ecuaciones de Euler para flujo compresible aplicado a dominios bidimensionales. Las ecuaciones se discretizan espacialmente mediante el método de los elementos finitos. Para la discretización temporal se utilizan esquemas combinados de Adams-Bashforth y Runge-Kutta.*
*Se estudian implementaciones usando OpenMP y CUDA y se presentan las performances obtenidas y se exponen las dificultades encontradas.*

**Palabras clave:** Flujo compresible, Elementos Finitos, OpenMP, CUDA, GPGPU.

## INTRODUCCIÓN

Durante gran parte de la historia de las computadoras, los programadores se vieron beneficiados por el continuo crecimiento de la velocidad de los procesadores. La observación de que la cantidad de transistores que entran en un chip se duplica cada 18 meses dio lugar a la conjetura de la Ley empírica de Moore: *"la capacidad de cómputo se duplica aproximadamente cada dos años"*. Efectivamente, su validez se mantuvo durante más de medio siglo, influenciando las decisiones en la industria del software.
Sin embargo, este crecimiento se vio desacelerado desde el 2003. Los problemas de disipación de calor en los chips sumado al alto consumo de energía ponen un límite en las velocidades de los procesadores. Los fabricantes pronto entendieron que si querían seguir escalando de acuerdo a la ley de Moore, al menos en términos de FLOPS, debían mudar a otro modelo de hardware, de single core a multi-core. Esto hace que la programación paralela sea un estándar actual no solo en el ámbito del cómputo de altas prestaciones (HPC). Algunos autores se refieren a esto como la revolución de la concurrencia [2].
La mayor complejidad en la programación paralela se hace evidente por la inclusión de conceptos nuevos como dependencia de tareas, condiciones de carrera, sincronización de hilos; pero más que nada por la incapacidad de los compiladores habituales de detectar errores relacionados a estos.
Mientras que los fabricantes de procesadores convencionales como Intel han tomado el camino de *multi-core* (pocas cantidades de poderosos núcleos diseñados cada uno para maximizar la velocidad), otros han tomado la trayectoria de *many-thread* o *many-core* (centenas o miles de núcleos muy simples), como son las placas gráficas (Graphical Processing Unit o GPU), diseñadas para procesar millones de operaciones independientes muy simples, como ser el rendering de cada píxel en una pantalla. Resulta que estas últimas encajan muy bien con las necesidades de muchos programas de aplicaciones científicas.
Los primeros intentos de aprovechar las GPUs para su uso en aplicaciones científicas han sido tediosos y estaban reservados para los expertos en programación con librerías gráficas. Esto cambió con la aparición de CUDA (Compute Unified Device Architecture) y OpenCL (Open Computing Language) que permiten total abstracción con las instrucciones gráficas de la GPU, y junto con la aparición de GPUs exclusivamente dedicadas a la industria HPC, hoy se habla de procesadores gráficos de propósito general (General Purpose Graphical Processing Unit o GPGPU).
El objetivo de este trabajo es implementar un código paralelo para resolver las ecuaciones de Euler con OpenMP y CUDA, comparar y explicar las performances obtenidas, y exponer las dificultades encontradas durante la implementación.

## METODOLOGÍA

Las ecuaciones de Euler en su forma conservativa se discretizan espacialmente mediante el método de elementos finitos. En particular, se utiliza una formulación Streamline-Upwind/Petrov-Galerkin (SUPG) [3, 4] junto con un método de alto orden para el caso de inestabilidades por discontinuidades en la solución:

$$\frac{\partial U}{\partial t} + A_i \frac{\partial U}{\partial x_i} = 0 \tag{1}$$

La forma semi-discreta resulta:

$$\begin{aligned} &\int_\Omega N^T \left(\frac{\partial U^h}{\partial t} + A_i^h \frac{\partial U^h}{\partial x_i}\right) d\Omega \\ &+ \sum_{e=1}^{n_{el}} \int_{\Omega_e} \tau \frac{\partial N^T}{\partial x_j} A_j^h \left(A_i^h \frac{\partial U^h}{\partial x_i} - \vartheta^h\right) d\Omega_e \\ &+ \sum_{e=1}^{n_{el}} \int_{\Omega_e} \nu \frac{\partial N^T}{\partial x_i} \frac{\partial U^h}{\partial x_i} d\Omega_e = 0 \end{aligned} \tag{2}$$

Aquí, $\tau$ y $\vartheta$ son parámetros de estabilización de SUPG [3, 4] y $\nu$ es el parámetro de captura de choque [3]. $N^T$ es la matriz de las funciones de forma.
Para la discretización temporal, se decide utilizar un esquema explícito Runge-Kutta de cuarto orden de mínimo almacenamiento (minimal storage RK scheme [1]) e intercalar los pasos con un esquema de Adams-Bashforth:

*Runge-Kutta:*

$$\Delta U^{n+i} = \alpha_i \Delta t \, \boldsymbol{r}\left(U^n + \Delta U^{n+i-1}\right), \quad i = 1, \ldots, 4, \quad \Delta U^0 = 0, \quad \alpha_i = 1/i \tag{3}$$

*Adams-Bashforth:*

$$U^{n+4} = U^{n+3} + \Delta t \left(\frac{55}{24}\boldsymbol{r}\left(U^{n+3}\right) - \frac{59}{24}\boldsymbol{r}\left(U^{n+2}\right) + \frac{37}{24}\boldsymbol{r}\left(U^{n+1}\right) - \frac{3}{8}\boldsymbol{r}\left(U^n\right)\right) \tag{4}$$

La ventaja de esta combinación se debe a la mayor rapidez del esquema de Adams-Bashforth, ya que luego de las primeras iteraciones, solo necesita una evaluación del miembro derecho $\boldsymbol{r}(U)$ por cada paso. La parte pesada del algoritmo se centra en las sucesivas evaluaciones del vector $\boldsymbol{r}(U)$ :

```
foreach Ω_e in Ω :
  foreach node k in Ω_e :
      rhs[map[e,k]] += ∫_{Ω_e} N_e^k A_i^h ∂U^h/∂x_i + τ ∂N_e^k/∂x_j A_j^h (A_i^h ∂U^h/∂x_i − ϑ^h) + ν ∂N_e^k/∂x_i ∂U^h/∂x_i dΩ_e
  end
end
rhs := −M^{-1} *rhs
```

donde $M$ es la matriz de masa condensada, y `map[e,k]` es un arreglo de las conectividades de los nodos de la malla, que indexa el nodo de numeración local $k$ del elemento $\Omega_e$ con la correspondiente numeración global en la malla.

El cálculo de las integrales en cada elemento se puede realizar de forma independiente, pero el resultado local se debe ensamblar en el vector global `rhs` de forma atómica, ya que elementos vecinos `e1` y `e2` que comparten nodos tendrán `map[e1,k2]=map[e2,k2]` para algún par `(k1,k2)`, teniéndose una condición de carrera cuando dos hilos se encuentran procesando dos elementos vecinos. El uso de operaciones atómicas será necesario para serializar los accesos a la memoria cuando más de un hilo intenta acceder a la misma dirección. El efecto es una evidente disminución del paralelismo. Aun así, la mayor parte del trabajo que calcula la integral en cada elemento es efectivamente paralelizada. Además, el buen mallado promueve pocos elementos vecinos por nodo (evitando elementos "astilla") con lo que se considera apropiado el uso de operaciones atómicas.

## RESULTADOS Y DISCUSIÓN

Para las pruebas, se utilizaron las máquinas del grupo "Mendieta" de FaMAF, que cuenta con $2\times$ Intel Xeon e5-2680-v2 por cada nodo ( $2\times10$ núcleos), de 2.7 – 3.5 ghz. en cuanto a la GPU, se utilizó la tarjeta Nvidia tesla M2090 con 512 streaming processors de 1.30 Ghz.

## OPENMP

La implementación con OpenMP se resume en los siguientes pasos:

1. Definición de las regiones paralelas mediante las cláusulas `!$OMP PARALLEL` y `!$OMP END PARALLEL` y definición de la división de tareas.
2. Definición de las partes que necesitan sincronización o restricción mediante locks. De acuerdo a la necesidad, se utilizarán cláusulas `!$OMP ATOMIC`, `!$OMP CRITICAL`, `!$OMP BARRIER`, etc.
3. Definición de las variables privadas y compartidas.

Una distribución lógica es repartir equitativamente el bucle sobre los elementos en los hilos. Esta es la configuración por defecto si se usa `!$OMP DO`.
La mayoría de los problemas que se tuvieron durante la implementación son típicos en el uso de OpenMP, y se deben a variables mal declaradas como privadas o compartidas. El problema suele agravarse con la cantidad de variables y la extensión de la región a paralelizar.
En las **Figuras 1** y **2** se exponen los rendimientos obtenidos en la implementación para diferentes cantidades de núcleos. La figura ayuda a establecer la escalabilidad del código. Se traza una curva realizando un fitting según la ley de Amdahl $1/(\alpha+(1-\alpha)/P)$ .
En las **Figuras 3** y **4** se muestran los rendimientos para diferentes tamaños del problema utilizando 20 núcleos. Se traza una curva sobre los speedups logrados según $g/(1+\nu/w_g)$ donde $g$ es el máximo speedup, y $\nu$ y $w_g$ son la sobrecarga del manejo de los hilos y la cantidad de trabajo, respectivamente.

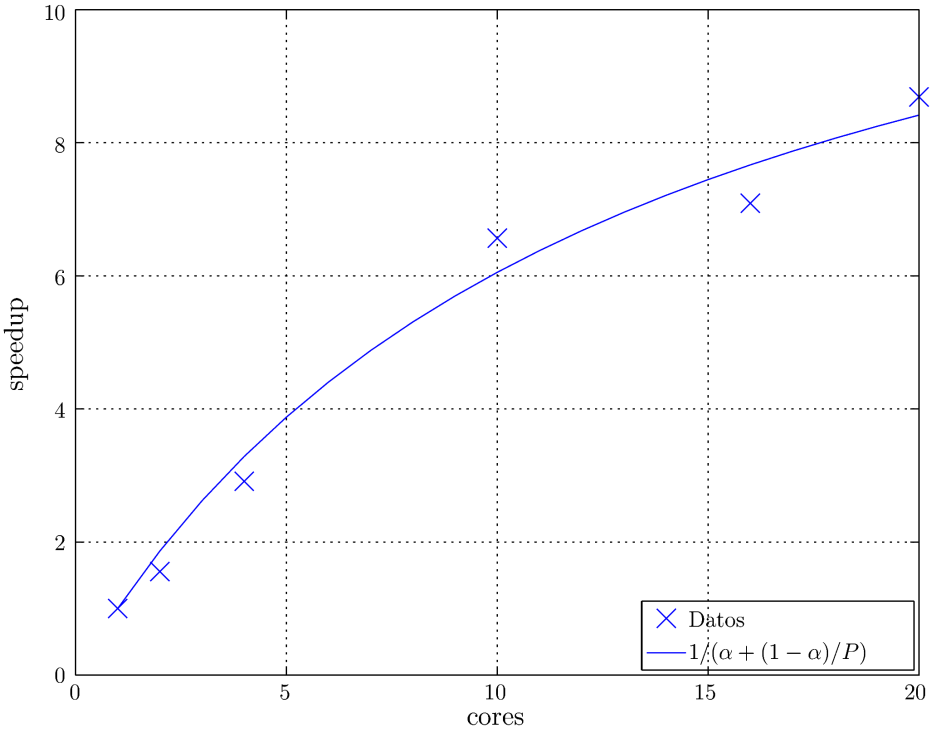



**Figura 1.** Speedup para diferentes cantidades de procesadores.

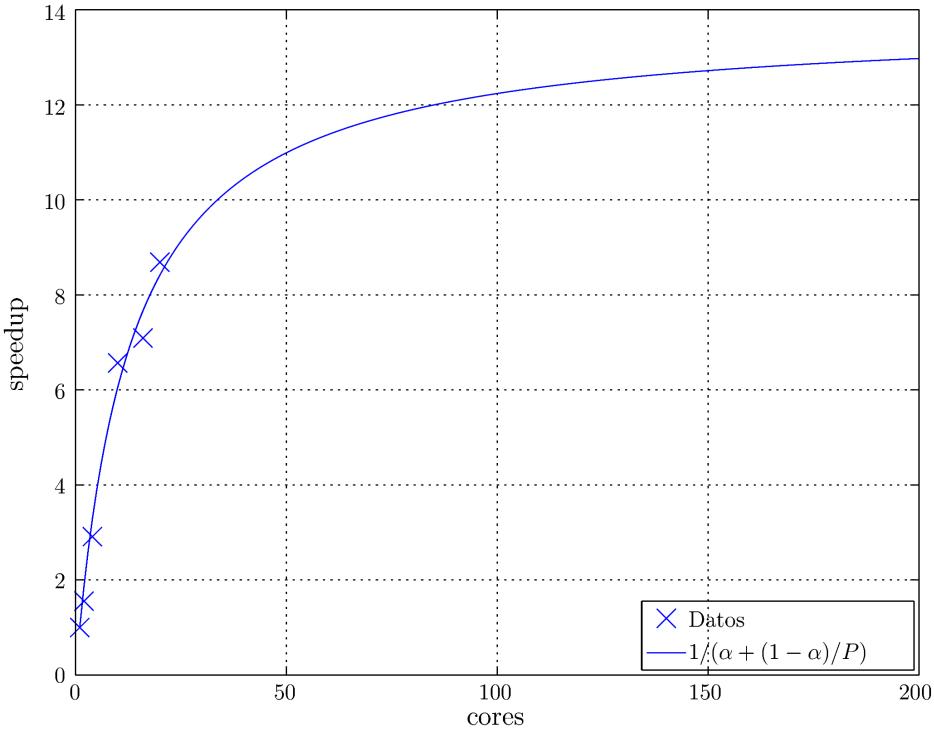



**Figura 2.** Speedup para diferentes cantidades de procesadores.

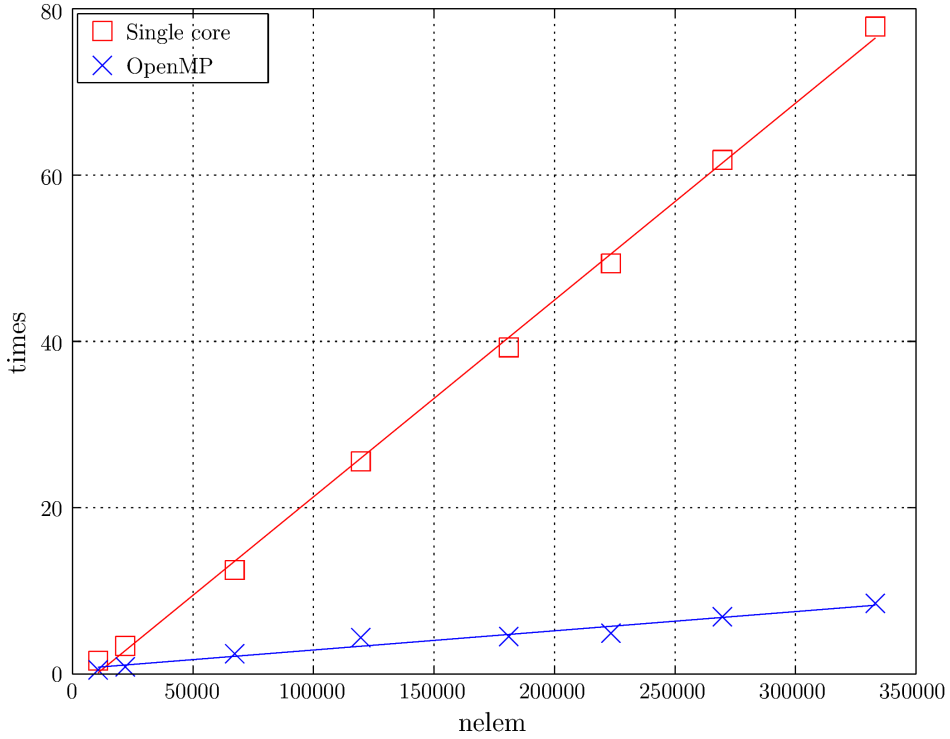



**Figura 3.** Tiempos para diferentes cantidades de elementos.

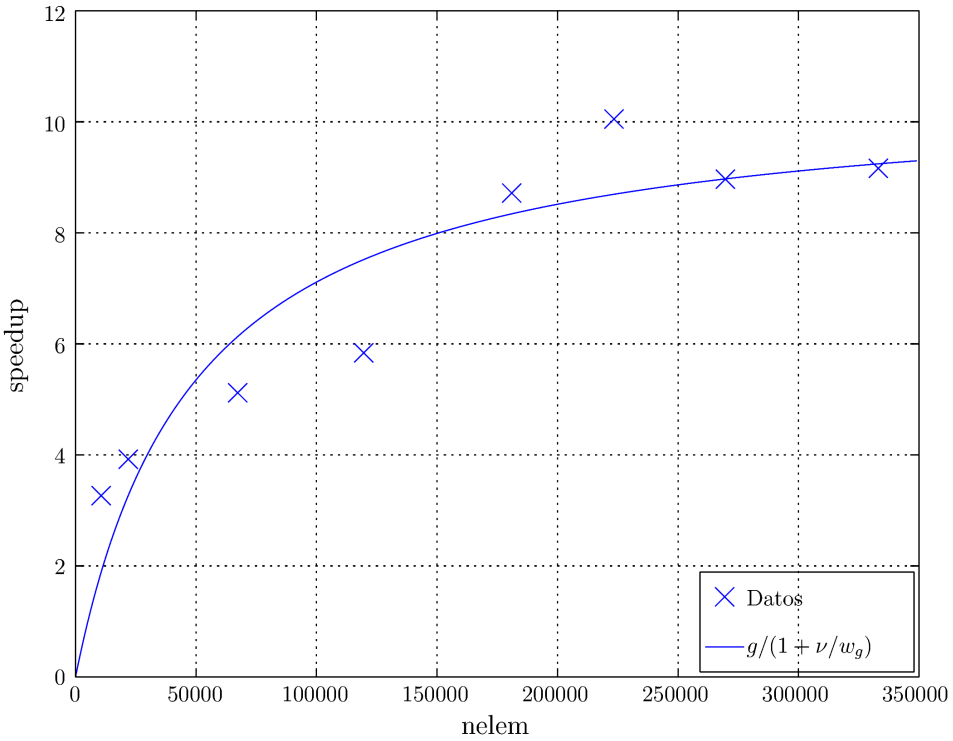



**Figura 4.** Speedup para diferentes cantidades de elementos.

**CUDA**

Para las pruebas, se utilizó la tarjeta Nvidia Tesla M2090 con 512 streaming processors de 1.30 Ghz. A diferencia de la CPU en donde es común hacer corresponder la cantidad de hilos de ejecución a la cantidad de procesadores físicos o núcleos, en la GPU es normal lanzar tantos hilos como subtareas se tengan. Esta distinción se debe a diseños de hardware fundamentalmente distintos. La GPU esta optimizada para el procesamiento masivo de hilos de ejecución simples, mientras que la CPU esta diseñada para una amplia variedad de tareas, desde hilos de ejecución simples hasta hilos de ejecución con flujos de control complejos.
La implementación con CUDA se resume en lo siguiente:

1. Escribir el *kernel* o función que será ejecutada por la GPU. Esta es una función precedida por el especificador `__device__` y define la tarea que realizará cada hilo ejecutado en la GPU. La cantidad de hilos que se lanzarán se especifica en la llamada del kernel, en este caso `nelem` hilos (`nelem` = cantidad de elementos), donde cada hilo se encarga de procesar un único elemento de la malla.
2. Declarar punteros y reservar espacios en la memoria de la GPU, que serán dedicados a albergar los datos de entrada para el kernel. Este paso se realiza una sola vez.
3. El bucle del cálculo consiste básicamente en:
   a) transferir los datos de entrada a la GPU,
   b) ejecutar el kernel, y
   c) transferir los datos de salida a la CPU para su posterior uso.

El cuello de botella en las aplicaciones de CUDA suele ser la transferencia de datos entre CPU y GPU. En este trabajo, no se considerarán los tiempos de transferencia y se enfocará solamente en el tiempo de cómputo del kernel.
La primera implementación se realizó de forma directa, sin cambios significativos. Las dificultades encontradas fueron propias de traspasos de códigos de Fortran a C:

- Fortran utiliza la disposición de arreglos tipo *column-major-layout,* mientras que C utiliza la disposición *row-major-layout*.
- Fortran utiliza la convención *call-by-reference*, mientras que C utiliza la convención *call-by-value*.
- Fortran utiliza la indexación comenzando en 1, mientras que C utiliza la indexación comenzando en 0. Los índices globales y locales de los nodos en la malla deben ser reenumerados acordemente.
- C no posee la flexibilidad de Fortran en el manejo de arreglos multidimensionales gracias a los *array descriptors*. Las variables alocadas en C deben ser tratadas unidimensionalmente. El uso de macros como `#define IDX(i,j,n) ((i)*(n)+j)` puede simplificar esta tarea.

Otra dificultad que se encontró es la falta de soporte en algunas operaciones de doble precisión, en particular el `atomicAdd` para doble precisión que es utilizado para acumular y ensamblar los resultados de la integral al vector `rhs` atómicamente. Una solución es utilizar el siguiente fragmento de código, que se muestra a continuación:

```
__device __
double atomicAdd ( double* address, double val )
{
unsigned long long int* address_as_ull =
( unsigned long long int* ) address;
unsigned long long int old = * address_as_ull , assumed;
do {
assumed = old;
old = atomicCAS ( address_as_ull, assumed,
_double_as_longlong ( val + _longlong_as_double ( assumed ) ) );
} while ( assumed != old );
return _longlong_as_double ( old );
}
```

**Codigo 1.** Fragmento de código para sumas atómicas de doble precisión en CUDA.

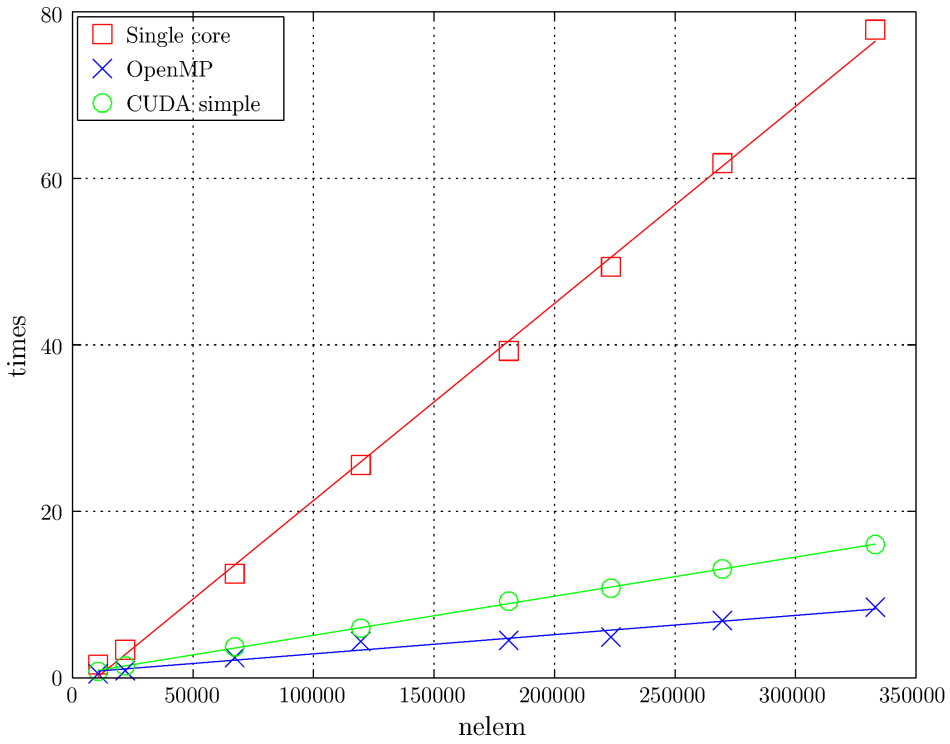



**Figura 5.** Tiempos para diferentes cantidades de elementos.

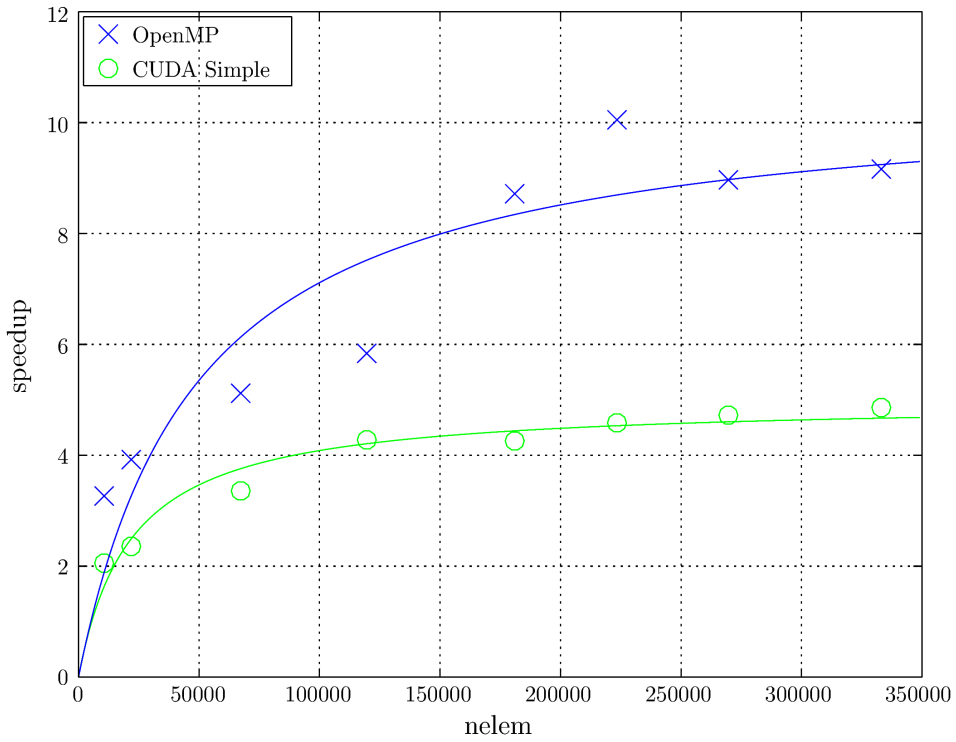



**Figura 6.** Speedup para diferentes cantidades de elementos.

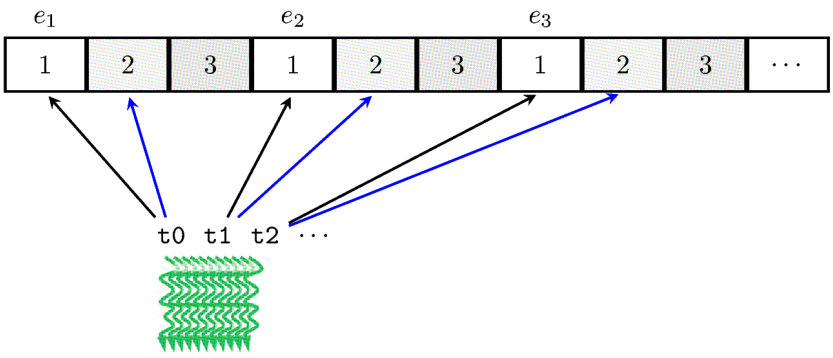
En las **Figuras 5** y **6** se comparan los rendimientos encontrados en CUDA y OpenMP. Considerando que las características teóricas de la GPU (bandwidth y FLOPS) son superiores a las de la CPU utilizada, los rendimientos de la implementación con CUDA fueron poco satisfactorios. Algunas de las razones de este bajo rendimiento son:

1. Presión del registro: El cálculo de la integral en cada elemento requiere del uso de muchas variables intermedias. Esto sumado al empleo de doble precisión para las variables hace que haya mucha presión del registro. Esto provoca un *register spilling*, que sumado al poco cache de las GPUs, hace que algunas variables del kernel deban ser alojadas en la memoria global, de velocidades de acceso mucho más lentas (dos órdenes de magnitud).
2. Baja ocupación de los Stream Multiprocessors: Es otra consecuencia del punto anterior, ya que la cantidad de registros utilizados limita la máxima cantidad de hilos que pueden ser alojados al mismo tiempo en el Stream Multiprocessor. La herramienta de diagnóstico `nvprof` muestra una ocupación de menos del 40%. El resultado de esto es que los multiprocesadores probablemente se encuentren trabajando por debajo de su capacidad máxima.
3. Disposición no favorable de los arreglos: Una disposición de arreglos multidimensionales favorable para su uso en la CPU suele ser poco favorable para la GPU. Desde el punto de vista de un hilo de ejecución de la CPU, la forma óptima de disponer un arreglo multidimensional es de forma tal de aprovechar la localidad espacial, es decir, que las variables necesarias para la ejecución del hilo se encuentren próximos en la memoria. En la GPU, esto es algo diferente ya que los hilos son agrupados en *warps* y ejecutados de manera SIMD (*single instruction, multiple data*). En este caso, la forma óptima es disponer los arreglos de forma que hilos contiguos accedan a posiciones contiguas de memoria. En CUDA, esto se llama *coalesced accesses*.
4. Uso de `atomicAdd` de doble precisión presentado arriba: Una causa obvia es el uso del artificio para poder realizar sumas atómicas de doble precisión, que no cuenta con soporte de

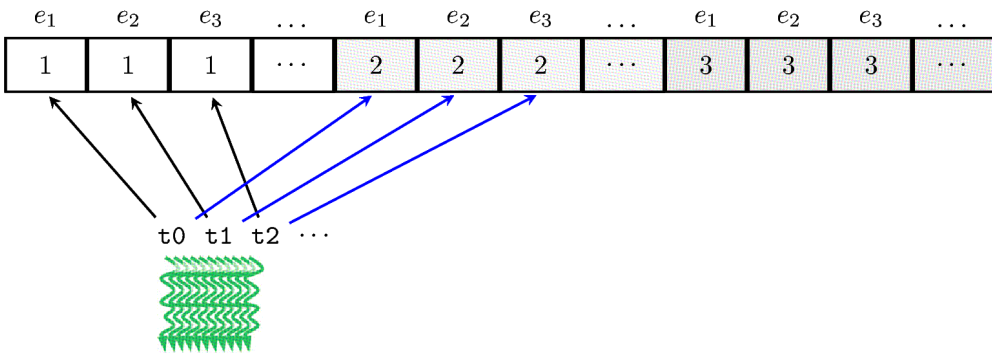
hardware como otras operaciones atómicas como `atomicAdd` de simple precisión, `atomicCAS`, `atomicMax`, etc.

Una alternativa común que se hace para solucionar los puntos 1 y 2 es dividir el kernel en kernels más pequeños y simples. En el presente código, sin embargo, no se ha encontrado una mejora, debido principalmente a la operaciones redundantes que resultan de aplicar esta alternativa.
Con respecto al punto 3, en la mayoría de los casos el problema se resuelve fácilmente transponiendo las matrices. Este ha sido el caso para muchas de las matrices del código. Para otros arreglos, fue necesario un poco más de trabajo. Por ejemplo, los arreglos `rhs[ipoin,ivar]` y `U[ipoin,ivar]` guardan los valores del right hand side y de las variables conservativas en el nodo de índice global `ipoin`, respectivamente. Como el índice global de los nodos de un elemento pueden ser aleatorios, el acceso a los arreglos mencionados es disperso.
Una solución que se opto es el uso de arreglos 3D que almacenan las contribuciones locales de cada elemento, de forma separada. Por ejemplo, `U2[ivertex,ielem,ivar]` almacena las variables conservativas del vértice `ivertex` del elemento `ielem`. De esta forma, los hilos contiguos accederán a posiciones contiguas (ver **Figura 7**). La otra gran ventaja de esta opción es que se evita el uso del costoso `atomicAdd` de doble precisión. Para recuperar los arreglos `rhs[ipoin,ivar]` y `U[ipoin,ivar]`, se reducen los arreglos grandes, que puede ser realizado en la CPU, o mediante otro kernel:

```
for each ivertex:
  for each ielem:
    U[map[ielem,ivertex],ivar] += U2[ivertex,ielem,ivar]
    rhs[map[ielem,ivertex],ivar] += rhs2[ivertex,ielem,ivar]
  end
end
```




a) Accesos dispersos.




b) Accesos contiguos.

**Figura 7.** Ejemplo de accesos coalesced y non-coalesced en CUDA.

La clara desventaja de este enfoque es el mayor uso de memoria (aproximadamente un factor 6 por cada arreglo, para casos 2D). Las **Figuras 8** y **9** muestran los rendimientos encontrados para la nueva implementación. Se puede apreciar que los cambios realizados produjeron una mejora significativa, con un rendimiento más coherente con las capacidades especificadas de la GPU.

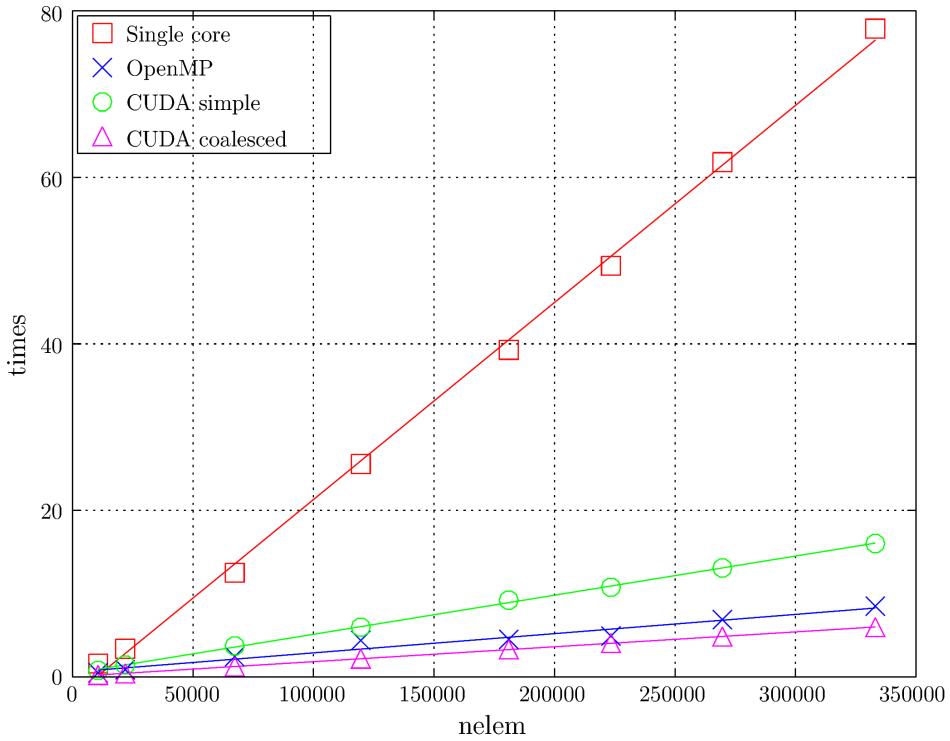



**Figura 8.** Tiempos para diferentes cantidades de elementos.

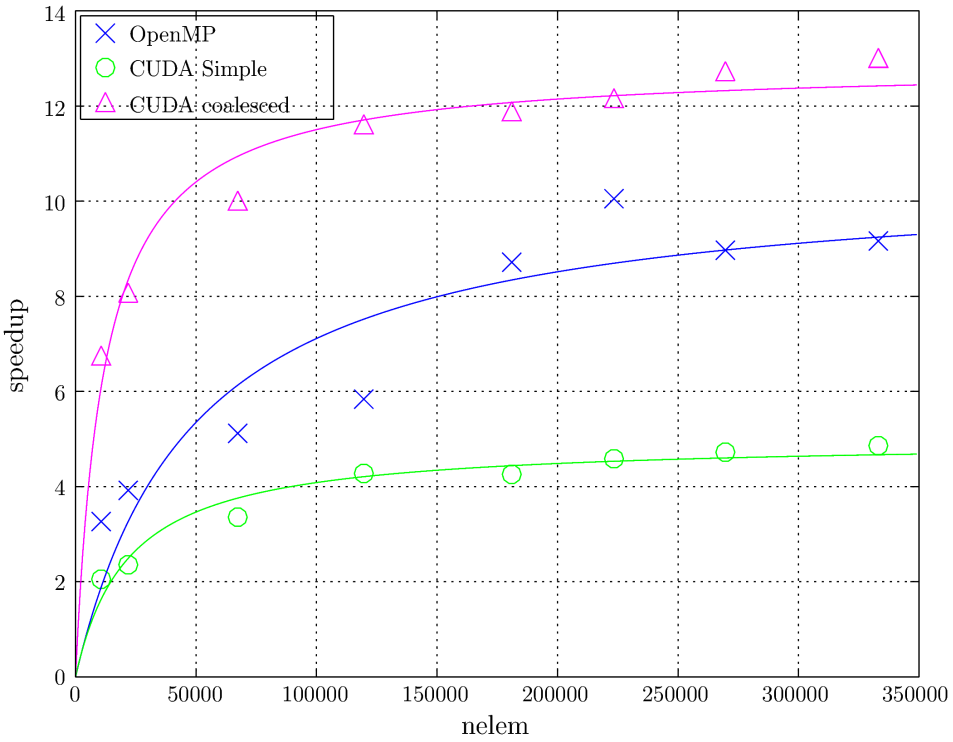



**Figura 9**. Speedup para diferentes cantidades de elementos.

## ENSAMBLAJE DE LA MATRIZ DE LOS LAPLACIANOS CON OPENMP Y CUDA

Una tarea común en CFD es el ensamblaje de propiedades de los nodos de cada elemento en una matriz global, por ejemplo, la matriz de masa. Estas matrices son en general tipo *sparse*. En esta sección se paraleliza un código para el ensamblaje de la matriz de los Laplacianos de las funciones de forma de la malla. Esta matriz se utiliza para suavizar los desplazamientos de la malla en una formulación ALE.
El código se describe en lo siguiente:

```
foreach point p :
  foreach Ω_e vecino de p :
    Encuentre el índice local i∈Ω_e de p
    foreach node j∈Ω_e :
      Encuentre la ubicación k∈S_laplace correspondiente a j
      S_laplace[k] ← S_laplace[k] + ∇N^i · ∇N^j
    end
  end
end
```

En este caso, no es necesario operaciones atómicas. En la **Figura 11** se muestra los rendimientos obtenidos para las implementaciones en CUDA y OpenMP. Se puede ver que los speedups obtenidos con OpenMP son similares a los obtenidos para el cálculo del right hand side, como era de esperarse. Sin embargo, la implementación en CUDA dieron resultados muy insatisfactorios, a pesar de que la rutina es más liviana (menos registros) y no necesita operaciones atómicas. La causa de este bajo rendimiento se debe, además de los accesos dispersos que presenta, a la *divergencia de warp* que ocasiona el algoritmo. Los algoritmos que involucran algún tipo de búsqueda presentan condicionales anidados, que no son óptimos para su empleo en las GPUs. Las rutinas con flujo de control simple y predecible son las que las mejores oportunidades tienen para ser aceleradas usando GPUs.

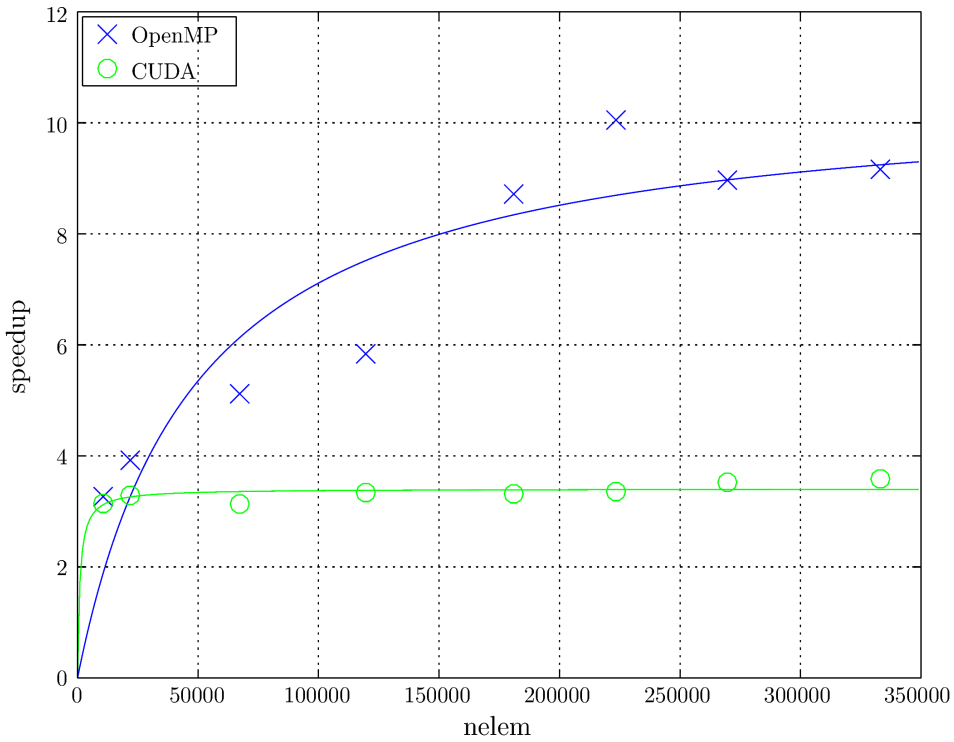



**Figura 10.** Ensamblaje de la matriz de los Laplacianos.

## CONCLUSIONES

Se han realizado y comparado paralelizaciones del código mediante OpenMP y CUDA. Las dificultades que se encontraron con OpenMP fueron típicas de las implementaciones del mismo: la facilidad y flexibilidad que ofrece el standard para su implementación rápida es la propensión de errores, la cual es menor cuando se programan explícitamente los hilos como en CUDA o Pthreads. Se vio que si bien la implementación en CUDA dio resultados esperados, su aplicación no fue directa como el caso con OpenMP, sino que fueron necesarias algunas modificaciones.
A continuación, se presenta una tabla comparativa de las especificaciones de los recursos empleados en este trabajo, y los speedups logrados para la subrutina:

**Tabla 1.** Especificaciones de los recursos empleados.

| | **2x Intel Xeon E5-2680 v2** | **Nvidia Tesla M2090** |
|---|---|---|
| **Procesadores** | 2 x 10 = 20 núcleos | 512 stream processors |
| **Bandwidth** | 2 x 59.7 = 119.4 GB/s | 177 GB/s |
| **Peak Performance (doble precisión)** | 2 x 216 = 432 GFLOPS | 666 GFLOPS |
| **Precio** | 2 x $1727 = $3454 (ark.intel.com) | $1570 (amazon), debut (2011): $2600 |
| **Speedup logrado** | ~10 | ~13 |

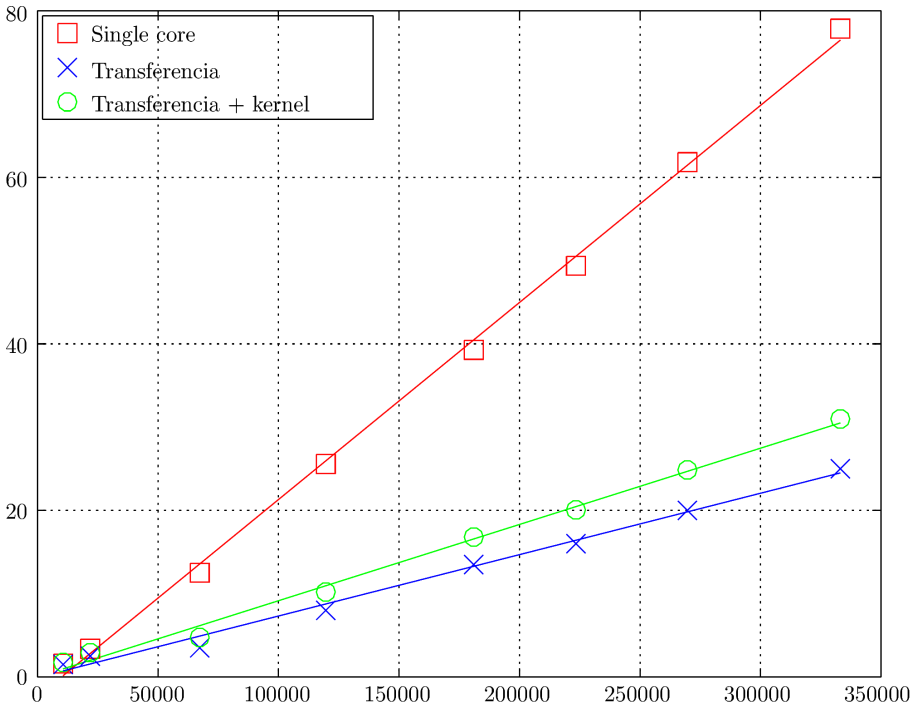



**Figura 11.** Tiempos de la implementación en CUDA, incluido tiempos de transferencia.

Por último, para ilustrar la importancia de los tiempos de transferencia entre CPU y GPU, en la **Figura 10** se comparan los tiempos del kernel + transferencia con los tiempos de la versión secuencial del código. Los tiempos de transferencia incluyen tanto la subida de los datos de entrada como la bajada de los datos resultados. Se ve inmediatamente que la implementación pierde toda su eficiencia si las transferencias se deben realizar por cada llamada de la subrutina, con speedups no mayores a 3.

## TRABAJOS FUTUROS

El bajo rendimiento de la rutina de ensamblaje de la matriz de los Laplacianos presentada anteriormente se debe a la divergencia de warp que genera el código. Un método sugerido en la referencia [6] para el ensamblaje de la matriz de masa puede ser útil, ya que el algoritmo en cuestión es similar. El método consiste en almacenar las contribuciones individuales de los elementos de forma separada, en un arreglo $\boldsymbol{S}^e$ expandido, y además utilizar una matriz $A$ de ceros y unos, denominada matriz de ensamblaje, que se construye una única vez para una malla de interconectividad fija (ver **Figura 12**). Entonces, la matriz sparse de los Laplacianos $\boldsymbol{S}$ se obtiene calculando:

$$\boldsymbol{S} = A^T \boldsymbol{S}^e A \tag{5}$$

En realidad, no es necesario calcular la matriz $\boldsymbol{S}$ de forma explicita, sino que cuando se necesite calcular $\boldsymbol{S}\boldsymbol{v}$ se hace:

$$\boldsymbol{S}\boldsymbol{v} = A^T \left( \boldsymbol{S}^e \left( A \boldsymbol{v} \right) \right) \tag{6}$$

O sea, 3 multiplicaciones *sparse-vector*, que es más barato que la multiplicación matriz-matriz. Operaciones algebraicas como estas son muy eficientes en la GPU, y la referencia sugiere que con este método se logran mejores rendimientos.

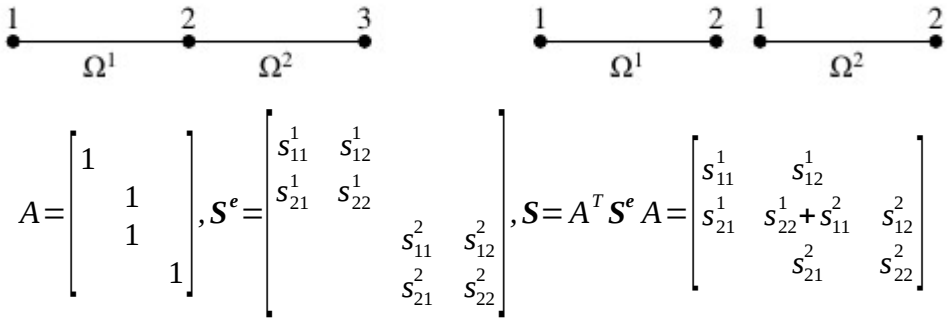



**Figura 12.** Ejemplo en 1D del método de ensamblaje.

## REFERENCIAS


1. R. Lohner (2008). Applied computational fluid dynamics techniques, 2nd Edition.
2. Sutter, H. & Larus, J. (2005). Software and the Concurrency Revolution. ACM Queue, 3(7), 54À62.
3. T. E. Tezduyar & M. Senga (2005). Stabilization and shock-capturing parameters in SUPG formulation of compressible flows
4. O. Soto, R. Lohner, J. Cebral & R. Codina (2001). A time-accurate implicit-monolithic finite element scheme for incompressible flow problems.
5. D. B. Kirk & W. Hwu (2012). Programming massively Parallel Processors, 2nd Edition.
6. G. Markall, A. Slemmer, D. Ham, P. Kelly, C. Cantwell, and S. Sherwin. Finite element assembly strategies on multi- and many-core architectures. Elsevier, 2012.