PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Reporte Técnico RT 10-06

Estudio preliminar de estrategias híbridas de cómputo CPU-GPU para acelerar algoritmos evolutivos

José Blengio Nicolás Soca Martín Pedemonte
Pablo Ezzatti

2010

Estudio preliminar de estrategias híbridas de cómputo CPU-GPU para acelerar algoritmos evolutivos
Blengio, José; Soca, Nicolás; Pedemonte, Martín; Ezzatti, Pablo
ISSN 0797-6410
Reporte Técnico RT 10-06
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, 2010

Estudio preliminar de estrategias híbridas de cómputo CPU-GPU para acelerar algoritmos evolutivos

Jose Blengio, Nicolas Soca, Martín Pedemonte[1], and Pablo Ezzatti[1]

Centro de Cálculo–Instituto de la Computación, Universidad de la República, 11.300–Montevideo, Uruguay, {mpedemont,pezzatti}@fing.edu.uy

Resumen Debido a su gran capacidad para encontrar buenas soluciones en tiempos de ejecución razonables las metaheurísticas son ampliamente utilizadas para la resolución de problemas de optimización. Dentro de las metaheurísticas se destacan los Algoritmos Genéticos (GAs). Sin embargo, resolver problemas con instancias de gran tamaño puede ser difícil incluso para este tipo de estrategias. Por esta razón, la paralelización de metaheurísticas es una alternativa interesante para disminuir los tiempos de ejecución de estos algoritmos.
En los últimos años, las GPUs han sufrido una evolución explosiva. Originalmente eran dispositivos diseñados para un único propósito específico, el procesamiento gráfico, pero en pocos años se transformaron en verdaderos multiprocesadores de memoria compartida. En base a esto, las GPUs se presentan como una plataforma poderosa para implementar algoritmos paralelos.
En este reporte, presentamos un estudio preliminar de paralelización de un algoritmo genético simple incluyendo estrategias híbridas de cómputo CPU-GPU. La propuesta presentada se basa en el esquema de paralelismo de GAs Maestro-Esclavo. Se presentan los resultados obtenidos utilizando una GPU de bajo rango (NVidia 9800 GTX+), alcanzando valores de speedup de 9x.

1. Introducción

Los algoritmos exactos no siempre son una buena opción para la resolución de instancias grandes de problemas de optimización, debido al alto costo computacional que tienen asociado. En este contexto, las metaheurísticas han emergido como una herramienta flexible y robusta para la resolución de este tipo de problemas. Las metaheurísitcas no garantizan la obtención de la solución óptima, pero son capaces de encontrar soluciones cercanas a las óptimas de manera eficiente. Una de las familias de estos métodos que ha crecido en popularidad en los últimos años son los Algoritmos Genéticos (GAs).

A pesar de la eficiencia de las metaheurísticas, su uso para la resolución de instancias grandes provoca importantes incrementos en los tiempos de ejecución. Por esta razón, y basados en el aumento de posibilidades ofrecido por

las arquitecturas de hardware modernas, la aplicación de estrategias de computación de alto desempeño y en particular la paralelización de metaheurísticas se presentan como una opción interesante para reducir los tiempos de ejecución.

En los últimos años, las unidades de procesamiento gráfico (GPUs) han sufrido un avance explosivo, pasando de ser dispositivos con una funcionalidad específica a ser verdaderos multiprocesadores de memoria compartida. En una primera etapa, el desarrollo del hardware gráfico no fue acompañado de un desarrollo equivalente de software. Si bien, se desarrollaron diversas herramientas, eran generalmente incompatibles entre ellas y estaban diseñadas en base a la arquitectura de cada modelo de tarjetas. Con la introducción de Compute Unified Device Architecture (CUDA) por NVIDIA se dió un paso importante en la simplificación de desarrollo de aplicaciones, ya que no es necesario utilizar las APIs gráficas o los lenguajes ensamblador para programar las GPUs. Ahora, las GPUs pueden ser vistas como un procesador multi-core de memoria compartida y de arquitectura SIMD para la resolución de problemas de propósito general.

En este reporte se presenta estudio de diferentes implementaciones de GAs paralelos, que utilizan la GPU para acelerar el tiempo de cómputo. La estrategia de paralelismo utilizada es el modelo Maestro-Esclavo.

Lo que resta de este documento se estructura de la forma que se describe a continuación. Primero se expone una introducción a los GAs y las opciones de aplicación de técnicas de paralelismo. Luego, en la Sección 3, se ofrece un resumen de las características de las GPUs modernas, así como un relevamiento de trabajos en el área de GAs con GPUs. La Sección 4, detalla las distintas versiones de GAs implementadas. Posteriormente, se exponen los experimentos realizados y los resultados obtenidos. Por último, se presentan las conclusiones arribadas durante el trabajo y los trabajos futuros planteados.

2. Algoritmo Genéticos

Dentro de las metaheurísticas y en particular entre las estrategias que emulan los procesos evolutivos de la naturaleza, se destacan los GAs [8,11,12]. Su popularidad ha crecido en forma notoria en los últimos años como consecuencia de su versatilidad para resolver problemas. Los algoritmos genéticos se basan en el esquema genérico presentado en la Figura 1.

Los GAs no trabajan directamente sobre soluciones del problema en cuestión, sino que lo hacen sobre una abstracción de los objetos solución, usualmente denominadas cromosomas por analogía con los conceptos biológicos. Los cromosomas a su vez se componen de genes, los cuales tienen un valor, conocido como alelo. A la representación de la solución se le conoce como genotipo, mientras que la solución al problema es el fenotipo. Por tanto se debe tener un mecanismo de codificación (y de decodificación) que permita pasar del fenotipo al genotipo y viceversa. Todos los individuos de la población tienen asociado un valor de fitness, que permite cuantificar la calidad de la solución representada por el individuo. Para obtener este valor se utiliza la función de fitness. La función de fitness se aplica al fenotipo correspondiente a un cromosoma, siendo esto de-

```
InicializarAleatoriamente(P(0));
generacion = 0;
mientras (no CriterioParada) hacer
     Evaluar(P(generacion));
     Padres = Seleccionar(P(generacion));
     Hijos = Aplicar_Recombinacion(Padres);
     Hijos = Aplicar_Mutacion(Hijos);
     NuevaPoblacion = Reemplazar(Hijos,P(generacion));
     generacion++;
     P(generacion) = NuevaPoblacion;
Fin
Retornar Mejor Solucion Encontrada
```

Figura 1. Esquema de Algoritmos Genéticos.

pendiente del problema que se está intentando resolver. Puede verse como una caja negra que guiará el proceso de búsqueda a través del operador de selección. La función de fitness debe contemplar el criterio del problema de optimización (minimización o maximización de un objetivo) y las restricciones presentes en el problema. En caso de trabajar con soluciones no factibles, la función de fitness deberá asignarle valores adecuados que garanticen que tales individuos no se perpetúen durante el proceso evolutivo.

Los operadores evolutivos determinan el modo en que el algoritmo explora el espacio de soluciones del problema. Una gran diversidad de propuestas de operadores evolutivos han surgido a lo largo de la historia de los GAs. A continuación, se examinarán los operadores más utilizados que son los de: selección, cruzamiento y mutación.

Selección La selección es el proceso mediante el cual los individuos son seleccionados para reproducirse, y crear individuos para la siguiente generación. Las estrategias de selección que garantizan que los mejores individuos, es decir aquellos con mayor valor de fitness, sobrevivan de una generación a otra, se conocen como estrategias de elitismo.

La estrategia de selección determina fuertemente la operativa del mecanismo de búsqueda (dirige al algoritmo hacia la exploración de secciones prometedoras del espacio de búsqueda). En la naturaleza, la presión de selección (o presión selectiva) determina la intensidad con la que el entorno tiende a dar ventajas adaptativas a organismos o a eliminar su información genética. La presión de selección es crítica para el funcionamiento de los GAs. Una presión de selección alta puede ocasionar que la búsqueda termine prematuramente (en un óptimo local). La situación se denomina como convergencia prematura. Una presión de selección baja puede conducir a que la búsqueda avance mucho más lento de lo necesario. Lo adecuado es intentar mantener un compromiso entre la exploración del espacio de búsqueda y la explotación de buenas soluciones. Algunos de los mecanismos de selección más comunes son la selección proporcional (o rueda de ruleta) y la selección por torneo.

Cruzamiento Con el operador de cruzamiento se intenta combinar soluciones existentes, con el fin de crear una nueva solución que preserve alguna de las características de cada padre. Es un operador probabilístico, es decir, su aplicación depende de una cierta probabilidad denominada probabilidad de cruzamiento. El nuevo individuo se forma combinando partes de cada uno de los dos padres. Una forma de hacerlo es seleccionar un punto de cruzamiento, dividir los cromosomas de los padres en el punto seleccionado, y luego unir las partes de los diferentes cromosomas para componer uno nuevo. Este operador es conocido como cruzamiento de un punto, y se puede extender la idea a dos, tres o más puntos.

Otra variante de operadores de cruzamiento, corresponde a la familia de operadores de cruzamiento uniforme parametrizados (UPX). Esta familia de operadores propone el intercambio aleatorio de alelos, correspondientes a los mismos genes, entre los individuos padre dependiendo de una probabilidad fija p. Las variantes paramétricas del operador se obtienen considerando p entre 0 y 0.5. El operador UPX tradicional corresponde a utilizar un valor de $p = 0.5$.

Mutación El operador tradicional de mutación introduce diversidad en el mecanismo evolutivo, simplemente modificando aleatoriamente uno de los alelos del cromosoma. Sobre un esquema de codificación binaria la modificación consiste en invertir el valor binario de un alelo, y por ello recibe el nombre de mutación de inversión del valor de un bit.

Al igual que el operador de cruzamiento, la mutación es también un operador probabilístico, en este caso la probabilidad con que se aplica el operador se denomina probabilidad de mutación.

2.1. GAs paralelos

La forma en que se puede categorizar la inclusión de estrategias de computación de alto desempeño (HPC) a los GAs depende de diversos elementos, entre otros:

- como se evalúa la función de fitness, lo cual puede realizarse de manera centralizada o distribuirse en varios procesos.
- si se utiliza una única población (también denominado, panmítica), o múltiples. En caso de utilizarse múltiples poblaciones, que políticas se utilizan para intercambiar individuos.
- el tamaño y organización de cada sub-población.
- como se aplica la selección, en forma global o local.
- el modo de aplicación del mecanismo de cruzamiento, que puede ser centralizado o distribuido.

A partir de los puntos descriptos anteriormente, surgen una variedad de algoritmos genéticos paralelos, a continuación se describen los principales características de cada variante presentadas siguiendo la taxonomía propuesta por Cantu-Paz [6].

Modelo maestro-esclavo Este modelo se basa en distribuir funcionalmente el algoritmo, asignando partes del proceso evolutivo a diferentes procesadores. Provee una población panmíctica, con evaluaciones realizadas en paralelo. La idea es tener un proceso maestro que maneje la población y asigna a un conjunto de procesos esclavos, individuos para ser evaluados (calcular la función de fitness para ese individuo). Luego de la evaluación, el maestro toma los resultados y aplica los operadores evolutivos para obtener la siguiente generación. El resultado es un algoritmo que obtiene los mismos resultados que un GA panmítico secuencial, pero con un tiempo de ejecución menor.

Este modelo puede ser implementado tanto en sistemas multiprocesadores de memoria compartida, como en sistemas de memoria distribuida.

Modelo de poblaciones distribuidas o de islas Este modelo, está orientado a la distribución de datos. Plantea la utilización de múltiples poblaciones, que trabajan en paralelo, pero intercambiando individuos (migración) con una cierta frecuencia. La razón fundamental para incluir este intercambio periódico es difundir buenas soluciones de una población a otra.

A esta clase de algoritmos evolutivos también se les conoce como algoritmos evolutivos distribuidos (dEA).

Varias políticas de reemplazo de individuos durante la migración han sido propuestas en la literatura. Una de las más comunes, es que si se migran k individuos, reemplaza los k peores individuos de la población de destino.

Es posible implementar este modelo sobre una arquitectura MIMD según la taxonomia de Flynn [7], mapeando cada isla en un procesador. Deben ser especificados nuevos parámetros como el número de individuos intercambiados, la frecuencia de la migración y la topología de conexión entre las islas.

Modelo Celular Un tercer modelo de GAs paralelo son los algoritmos genéticos celulares (cGA), también conocidos como algoritmos genéticos de grano fino o masivamente paralelos. En este modelo se define una estructura para la población (vecindarios) y la evaluación del fitness se hace en forma paralela para cada individuo. En cuanto a la selección y el cruzamiento estos se realizan de manera local en cada vecindario.

El desarrollo de los multiprocesadores de memoria compartida popularizó este modelo de algoritmos evolutivos paralelos en la década de 1990, dado que en muchos multiprocesadores los elementos de procesamiento se disponen en una grilla de dos dimensiones, al igual que los individuos de un cGA. En la actualidad este tipo de algoritmos evolutivos siguen siendo estudiados en la literatura, pero cuando se intenta implementar en forma paralela un cGA, aparecen dos dificultades características de estos algoritmos: en un cGA hay un gran número de sub-poblaciones; estas sub-poblaciones están estrechamente relacionadas.

Modelos híbridos Los modelos de PEAs anteriormente descriptos han progresado y se han diversificado a lo largo del tiempo, e incluso se han llegado a

combinar para obtener algoritmos híbridos, en los que se utilizan dos (o más) niveles de paralelismo. Por ejemplo, combinando el modelo de islas con el celular, haciendo que en cada una de las islas se ejecute un cGA, o contando con varias sub-poblaciones de paralelización distribuidas que ejecutan procesos migratorios entre sí, logrando hacer evaluaciones rápidas (utilizando un modelo maestro-esclavo en cada sub-población), y obtener evolución de poblaciones semi-independientes.

3. GPUs

La unidad de procesamiento gráfico (GPU), es un dispositivo diseñado originalmente, como lo indica su nombre, para estar dedicado exclusivamente al procesamiento de gráficos, para aligerar la carga de trabajo del procesador central (CPU). De esta forma, mientras gran parte de lo relacionado con los gráficos se procesa en la GPU, la CPU puede dedicarse a otro tipo de cálculos.

Si bien en un computador genérico no es posible reemplazar la CPU por una GPU, hoy en día las GPU son muy potentes, disponen de decenas o centenas de cores con frecuencias de reloj aceptables. Estas dos propiedades se traduce en una potencia de cálculo mucho mayor gracias a su arquitectura intrínsicamente paralela.

3.1. Programación de GPUs

En un principio, el avance en el hardware no fue acompañado por un avance en el software de manejo de las GPUs. La programación de las GPUs se realizaba mediante llamados a servicios de interrupción de la BIOS o utilizando las APIs gráficas como OpenGL [4] y DIRECTX [3]. Posteriormente, se comenzó a desarrollar en el lenguaje ensamblador específico de cada modelo, lo que implicaba la existencia de varios lenguajes y una baja portabilidad de los programas. Para tratar de solucionar esto, se comenzaron a utilizar lenguajes de más alto nivel, que funcionaran sobre los diferentes modelos de GPUs existentes. Alguna de las herramientas desarrolladas son Brook [5] y Cg [1]. Sin embargo, cada herramienta seguía siendo muy dependiente de la arquitectura de la GPU, el modelo, etc. Posteriormente, en el año 2007 NVIDIA presentó CUDA [2] buscando solucionar algunos de estos inconvenientes.

CUDA es una arquitectura de software para la gestión de cálculos en la GPU como un dispositivo paralelo. Está disponible para las tarjetas gráficas GeForce 8 Series en adelante, es compatible con los sistemas operativos Linux de 32/64 bits y Windows XP y sucesores de 32/64 bits, y está basado en una extensión del lenguaje de programación C.

CUDA se compone de una pila de capas de software, tal como muestra la Figura 2. Entre estas capas se encuentran un controlador de hardware, una interfaz de programación de aplicaciones y dos bibliotecas matemáticas de alto

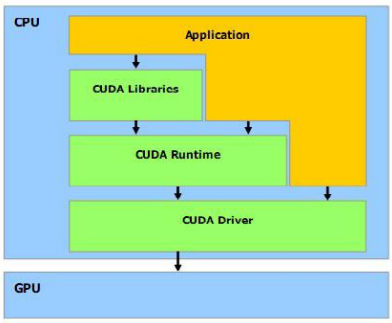

Figura 2. Arquitectura de CUDA

nivel, CUFFT y CUBLAS. El driver CUDA está dedicado a la transferencia de datos entre la GPU y la CPU.

Cuando se programa en CUDA, la GPU se ve como un dispositivo de cómputo capaz de la ejecución de un número muy elevado de hilos (threads) en paralelo. Funciona como un coprocesador de la CPU principal (o host). Más precisamente, una parte de una aplicación que se ejecuta muchas veces, pero independiente sobre diferentes datos, puede ser aislada en una función que se ejecutará en el dispositivo mediante muchos hilos de ejecución diferentes. A tal efecto, dicha función se compila utilizando el conjunto de instrucciones del dispositivo y el programa resultante, llamado núcleo (o kernel), se descarga en el dispositivo.

La arquitectura de CUDA es construida alrededor de un arreglo escalable de multiprocesadores (multithreaded Streaming Multiprocessors, SMs). Un multiprocesador en las GPU actuales, consiste en ocho procesadores escalares (SP), así como de unidades adicionales como una unidad de instrucciones multithread y un chip de memoria compartida. Los multiprocesadores crean, gestionan y ejecutan threads concurrentes en hardware sin overhead de planificación.

Los threads se agrupan en conjuntos denominados bloques (blocks), que tienen la particularidad de ser ejecutados en un único multiprocesador de la tarjeta. Un bloque puede estar formado hasta por 512 threads. Se denomina grilla (grid) a un conjunto de bloques. No hay un orden fijo de ejecución entre bloques, se ejecutan paralelamente si hay suficientes multiprocesadores disponibles en la tarjeta o si no en tiempo compartido. Cuando un programa CUDA en la CPU invoca una grilla a ser ejecutada en la GPU, los bloques de la grilla son enumerados y distribuidos a los multiprocesadores disponibles. Los threads de un bloque ejecutan concurrentemente en un multiprocesador, para ello se agrupan en conjuntos de 32 threads denominados warps. Los threads de un warp se ejecutan en un procesador.

Un warp ejecutará una sola instrucción a la vez, por lo que el máximo de eficiencia se logra cuando los 32 threads del warp ejecutan la misma instrucción. En el caso de que los threads de un warp diverjan (por ejemplo debido a una instrucción de control de flujo), el warp ejecuta de manera serial cada una de las

rutas tomadas por los threads, desactivando aquellos threads que no siguen esa ruta, y cuando se completan todas las rutas, todos los threads convergen. Esto solo ocurre dentro de un warp, los diferentes warps ejecutan de forma independiente, sin importar si están ejecutando rutas comunes o disjuntas. Cuando un bloque termina, un nuevo bloque es asignado al multiprocesador que queda disponible.

Tanto el host como el dispositivo mantienen su propia DRAM, denominadas como memoria del host y memoria del dispositivo, respectivamente. Se pueden copiar datos de una DRAM a la otra, a través de llamadas optimizados a la API que utilizan el dispositivo Direct Memory Access (DMA).

Los threads pueden tener acceso a los datos a través de múltiples espacios de memoria durante su ejecución. Cada thread posee su propia memoria local. Cada bloque tiene un espacio de memoria compartida que es visible por todos los threads pertenecientes a ese bloque, y su tiempo de vida es igual al del bloque. Finalmente, todos los threads tienen acceso a una misma memoria global de la tarjeta. La memoria global no está cacheada, por lo que es muy importante minimizar y seguir patrones de acceso, para obtener el máximo ancho de banda, ya que los accesos a la memoria global son muy costos. Al igual que la memoria global, la memoria local tampoco está cacheada por lo que el acceso a esta memoria es tan costoso como acceder a la memoria global. La memoria compartida que está en el chip, es mucho más rápida que las memorias global y local. De hecho, para todos los threads de un warp acceder a la memoria compartida es casi tan rápido como acceder a los registros.

3.2. Uso de GPU para EAs

Esta subsección presenta el relevamiento de trabajos en el área de utilización de GPUs para computar GAs. Debido a que la utilización de GPUs como plataforma de ejecución para GAs es relativamente nueva y hay por ende muy pocos trabajos referentes al tema, el relevamiento abarca la aplicación de paralelismo con GPUs a otras estrategias evolutivas. Además, muchas ideas de la paralelización de estrategias evolutivas son extrapolables en forma inmediata a los GAs.

Uno de los primeros trabajos en el área es el de Wong et al. [21], en el que se estudiaron estrategias de paralelismo tipo maestro-esclavo para aplicaciones de Programación Evolutiva (EP) aplicadas a la resolución de problemas de test simples. La EP difiere de los GAs en que solo posee como operador evolutivo, el operador de mutación. En esta propuesta, denominada Fast Evolutionary Programming (FEP), utilizan texturas para almacenar la población y una textura extra para almacenar los valores de fitness, y emplean Fragment Shader para implementar las funciones. En cuanto a las estrategias de cómputo, la selección se realiza en CPU mientras que la evaluación de la función de fitness y la mutación se realiza en GPU. La evaluación experimental del trabajo se realizó en una máquina Pentium IV 2.4 GHz con una tarjeta GeForce 6800 Ultra, mostrando valores de speedup en el rango de $0{,}62x$ y $5x$ dependiendo del tamaño de la población utilizada. Los mayores valores de speedup son alcanzados trabajando con las poblaciones de mayores tamaños. Posteriormente los autores [19]

presentaron el Parallel Hybrid Genetic (HGA) ejecutando completamente sobre GPUs. El HGA propuesto es un GA incorporando un operador de mutación de Cauchy, donde todos los pasos de la evolución, excepto la generación de los números aleatorios, se realiza en GPU. La propuesta se basa en operadores de selección diseñados especialmente para explotar las prestaciones de las GPUs, en particular presentan dos propuestas de operadores de selección, selección global y selección pseudo determinista. Trabajan con una computadora AMD Athlon 64 3000 con 1GB de memoria RAM y una tarjeta gráfica GeForce 6800 Ultra con 256 MB. La evaluación se realizó comparando HGA con una implementación equivalente en CPU, presentan valores de speedup entre $1{,}14x$ y $4{,}24x$ para tamaños de población entre 400 y 6400. Finalmente, los autores [20] extienden sus pruebas alcanzando valores de speedup de $5{,}50x$ para otros casos con evaluación de fitness más compleja.

Harding y Banzhaf [10] implementaron la metaheurística programación genética (GP) sobre GPU. En el trabajo se presenta una evaluación de las herramientas disponibles para el desarrollo en GPU de la época, y las razones de los autores para haber escogido Accelerator. La plataforma de experimentación fue un procesador Intel Centrino T2400 con una GPU NVidia GeForce 7300. Presentan importantes reducciones en los tiempos de ejecución, dependiendo el tipo de función abordada. Para los casos reales alcanzan valores de speedup de decenas, centenas para las funciones booleanas y de miles para las funciones con punto flotante. Utilizando ideas similares, Harding [9] estudió variantes de GP para implementar filtros para imágenes. En este caso, la funcion de fitness consiste en aplicar filtros a varias imágenes. El trabajo se focaliza en la estrategia de resolución del problema con heurísticas y no en los niveles de paralelismo alcanzados.

También dentro de la categoría de maestro-esclavo, Maitre et al. [16] extendieron EASEA (metalenguaje para implementar EAs) para que explote en forma automática las capacidades de la GPU. La propuesta es validada resolviendo dos casos, el problema de Weierstrass-Meldelbrot y un problema real del área de química de definición de estructuras de cristales . Para los experimentos utilizan dos tarjetas gráficas diferentes, una NVidia 8800 GTX y una NVIDIA GTX 260, y una computadora con procesador Pentium a 3.6 GHz. La evaluación se centra en la capacidad de escalado de la propuesta, mostrando incrementos sublineales de tiempos de ejecución al aumentar el tamaño de la población. En cuanto a las mejoras de tiempos de ejecución al comparar con la versión en CPU, miden el speedup considerando únicamente la etapa de evaluación de la función de fitness. Para el problema de Wierstrass presentan un estudio del speedup de la etapa de evaluación, alcanzado valores de $33x$ y $100x$ para una población de 4096 con la tarjeta 8800 GTX y GTX 260 respectivamente. Para el problema de química, trabajan con una población de 20000 individuos. Al utilizar la codificación de EASEA la etapa de evaluación tomaba 23s en CPU y 80s en GPU. Sin embargo, haciendo modificaciones en forma manual en las estructuras de datos del problema, logran en $7{,}66$s evaluar los individuos utilizando la CPU y en $0{,}33$ con la tarjeta GTX 260.

El trabajo pionero en utilizar GPUs para aplicar estrategias de paralelismo celular es el de Yu et al. [22], en el que se propuso una implementación de un cEA completamente sobre GPU para resolver el problema de minimización de Colille. La implementación utiliza fragment programs para las funciones y almacena los individuos en texturas. En el trabajo, presentan experimentos sobre una computador AMD Atholon 2500, con 512 MB de RAM y con una tarjeta GeForce 6800 GT mostrando valores speedup en el entorno de $20x$ para poblaciones de 512^2 individuos.

Otro trabajo sobre celulares es la propuesta de Li et al. [14], en la que el algoritmo también ejecuta completamente en GPU. La propuesta utiliza codificación binaria, texturas para almacenar la población y una textura extra con números aleatorios. Trabajando con una computadora Pentium IV de 2.66 GHz, 256 MB memoria RAM y una tarjeta GeForce 6800 alcanzan valores de speedup entre $1{,}4x$ y $5{,}4x$ para poblaciones entre 200 y 800, speedups de $9{,}6x$ a $16{,}9x$ para poblaciones de 3200 y valores entre $21x$ y $73x$ para una población de 10000 individuos.

Otra línea de trabajo dentro del paralelismo de grano fino es la propuesta de Li et al [15] que estudiaron la incorporación de estrategias de búsquedas locales, basadas en algoritmos inmunes, a un GA. Utilizan CUDA para implementar. Al aplicar el algoritmo propuesto al TSP obtiene valores de speedup entre $2{,}42x$ y $11{,}5x$ sobre una computadora AMD 3000+ con una tarjeta 9600 GT.

En el trabajo [17] se presenta una implementación de GAs paralelos siguiendo el modelo de islas aplicado a la resolución del QAP Para la evaluación experimental emplean una computadora con procesador Intel i7 965 con una tarjeta NVidia GTX285. Presentan valores de speedup de entre $3x$ y $12x$.

Una aproximación distinta fue seguida por Lewis et. al [13], el trabajo versa sobre programación genética cíclica. En el trabajo se presentan valores de speedup impresionantes, del orden de cientos. Sin embargo, la característica más destacada del trabajo es la única aproximación que evalúa el uso de varias GPUs al mismo tiempo (2 GPUs y 2 CPUs). Además, el trabajo presenta una interesante discusión sobre como evaluar el speedup al utilizar GPUs.

El único trabajo de algoritmos evolutivos para la resolución de problemas multiobjetivo (MOEA) y GPUs es la propuesta de Wong [18]. El autor presenta un algoritmo inspirado en el NSGA-II e implementado con CUDA. Se muestran dos estrategias para la implementación en GPU de la verificación de no dominancia, criterio utilizado para cuantificar la calidad de las soluciones en los algoritmos de resolución de problemas multiobjetivos. Se presentan resultados utilizando una computadora Intel Pentium Dual E2220, 2GB de memoria RAM y con una tarjeta gráfica GeForce 9600 GT con 512 MB. Se aplica el MOEA a la resolución de las funciones de prueba estándares ZDT y DTLZ, utilizando poblaciones de 4096 y 16384 individuos. La propuesta consigue valores de speedup total en el rango de $5{,}65x$ y $10{,}75x$.

En el relevamiento realizado se puede observar que la mayoría de los trabajos discuten implementaciones de programación evolutiva. En cuanto a las estrategias de paralelismos implementadas, se han cubierto las estrategias maestro-

esclavo [10,16,20,19,21], las estrategias celulares [14,15,22] y de islas [17]. Únicamiente, se relevó un trabajo que aborda las estrategias multi-objetivos [18], lo mismo sucede con la utilización de más de una GPUs al mismo tiempo [13] y con el desarrollo de frameworks o metalenguajes [16]. Otro aspecto importante del relevamiento es que no se encontró ningún trabajo que implementara técnicas de paralelismo maestro-esclavo con estrategias híbridas de cómputo utilizando GPU-CPU.

4. Propuestas

En esta sección se presentan las cuatro versiones de GA implementadas y se detalla la forma de uso de la GPU para las versiones que la utilizan.

4.1. GA**(CPU)**

La primera implementación, es un algoritmo genético que ejecuta completamente en CPU. Esta versión se desarrolló con el objetivo de tener un punto de comparación para las restantes versiones.

4.2. GA**(GPU)**

La segunda versión, utiliza la GPU para evaluar la función de fitness. En cada generación se copian todos los individuos desde la memoria del host a la memoria del dispositivo. Luego, se evalúan los individuos utilizando un thread para cada uno. La mecánica se puede observar en la Figura 3. Dependiendo del tamaño de la población se agrupan los threads en bloques de distintos tamaños. Si el tamaño de la población es potencia de 2, los threads se agrupan en bloques de 8, sino se agrupan en bloques de 10 threads. En ambos casos, se crean tantos bloques como sea necesario para cubrir toda la población. Una vez calculado el fitness, cada thread escribe el valor calculado en un vector, la posición donde escribe cada thread está preestablecida. Por último, el vector con los valores de fitness calculados es copiado desde la memoria del dispositivo a la memoria del host.

4.3. GA**(GPU-MC)**

La metodología del algoritmo es similar a la de la versión anterior. Pero en este caso se busca aprovechar la velocidad de la memoria compartida. Para implementar esto se tiene un thread por cada gen de la población. Cada thread copia el valor correspondiente desde la memoria global del dispositivo, hacia la memoria compartida del bloque correspondiente al thread. La mecánica se puede observar en la Figura 4. Luego de esto, solo continúa ejecutando un thread por individuo que se encarga de realizar la evaluación, pero realizando los accesos a la memoria compartida y no a la memoria global. El resto del algoritmo procede de forma análoga al algoritmo GA(GPU).

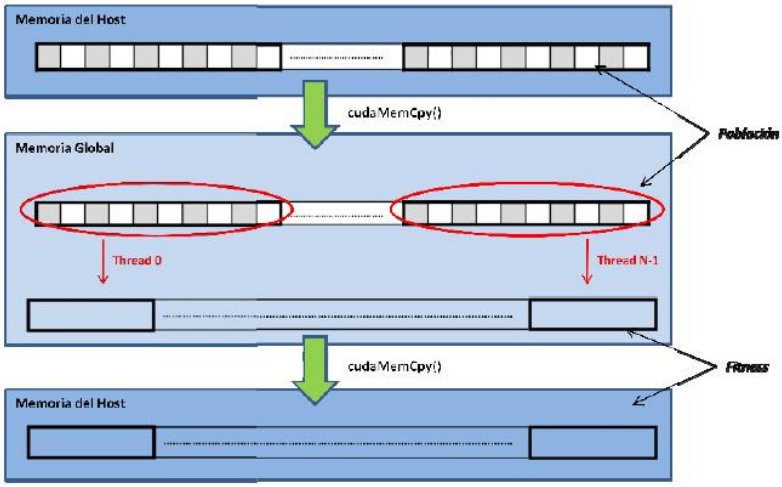

Figura 3. Estrategia de utilización de la GPU de la versión GA(GPU).

Al igual que el algoritmo anterior, se tienen dos tamaños de bloque diferentes, según el tamaño de la población que se utilice. En caso de que la población sea múltiplo de 10 se utilizan bloques de tamaño 100, y para los tamaños potencias de 2, los bloques de 80 thread.

4.4. GA**(HPU)**

Esta versión propone una estrategia de cálculo híbrido, en donde la evaluación del fitness de parte de la población se realiza en la GPU y el resto en la CPU. Este algoritmo pretende aprovechar el asincronismo entre la CPU y la tarjeta gráfica, esto implica aprovechar que en los algoritmos anteriores la CPU está ociosa durante el tiempo que tarda la evaluación de la función de fitness. La parte de la población que se evalúa en GPU se realiza con el algoritmo GA(GPU-MC).

5. Experimentos

En esta sección se presentarán los resultados obtenidos en los diferentes experimentos realizados. El estudio se centró en evaluar y comparar los tiempos de ejecución de las distintas versiones de GAs desarrolladas para distintos tamaños de población. No se realizó una comparación de la calidad de los resultados, debido a que todos los algoritmos presentados recorren el espacio de búsqueda de la misma forma.

Las pruebas se realizaron utilizando una tarjeta NVIDIA GeForce 9800 GTX+, y una PC con procesador Pentium(R) Dual Dual-Core CPU E5200 2.50GHz con 2 GB de memoria RAM. El algoritmo serial, GA(CPU), se desarrolló en lenguaje C,

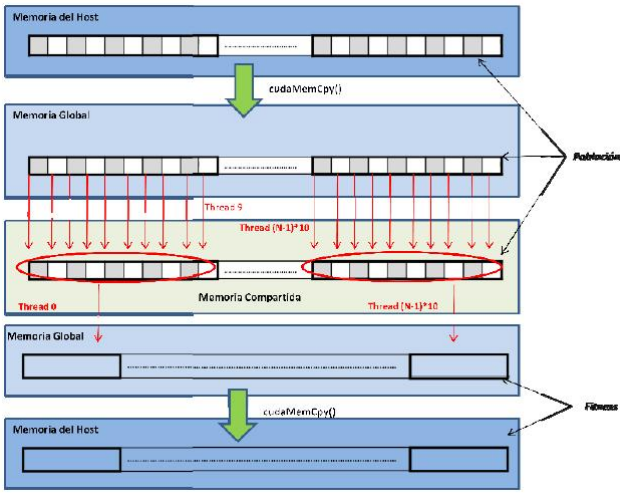

Figura 4. Estrategia de utilización de la GPU de la versión GA(GPU-MC).

y se utilizó el compilador gcc de Ubuntu, versión 4.3.2. El problema test que se utilizó es sencillo, consiste en minimizar la suma de las componentes de un vector. Es decir, dado un vector x de 10 valores con $x = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10})$ el objetivo es:

$$f(x) = min(\sum_{i=1}^{10} x_i) \tag{1}$$

En todos los casos, la población se organiza de forma lineal y cada cromosoma está compuesto por diez números enteros entre 0 y 1200.

Debido a la sencillez de la función a resolver, y de forma de trabajar con tiempos de ejecución razonables se decidió incorporar artificialmente un overhead a la evaluación de la función. Para esto, durante la evaluación de cada individuo, luego de calcular el valor de fitness correspondiente, se lee un gen del individuo, y con este valor se realizan operaciones sencillas de lectura de memoria, suma y escritura en la memoria, durante 5000 iteraciones.

5.1. Calibración

Se utilizó el operador de cruzamiento de un punto con una probabilidad de $0{,}8$ y $0{,}1$ como probabilidad de mutación. Estos valores son los recomendados según la literatura [].

En el caso del algoritmo GA(HPU) los tiempos de ejecución dependen del porcentaje de individuos evaluados en CPU y GPU. Por esta razón en la primera etapa del estudio se determinaron dichos porcentajes. Los mejores tiempos se obtuvieron evaluando en GPU el $93{,}75\,\%$ de la población cuando el tamaño de la misma es potencia de 2 y $95\,\%$ en el resto de los casos.

5.2. Resultados experimentales

En la Tabla 1 se muestra los tiempos de ejecución promedio para cada uno de los algoritmos, con diferentes tamaños de población.

Tamaño de la población	GA(CPU) tiempo (seg)	GA(GPU) tiempo (seg)	GA(GPU-MC) tiempo (seg)	GA(HPU) tiempo (seg)
100	21	16	4	3
128	28	19	5	4
200	40	30	7	7
256	57	38	7	7
500	103	74	14	12
512	117	77	13	13
1000	214	156	33	29
1024	237	161	34	31
2000	461	346	95	93
2048	512	354	99	98
4000	1098	819	317	311
4096	1153	846	332	328

Tabla 1. Tiempos de ejecución de las distintas versiones.

La Tabla 2 resume las mejoras alcanzadas utilizando la GPU para distintos tamaños de población. Se describen los tamaños de población y los speedups (tiempo serial/tiempo paralelo) alcanzados por los algoritmos paralelos.

La primera observación de los experimentos es que resulta ventajoso utilizar la GPU para computar los GAs, ya que se lograron tiempos de ejecución menores que usando solamente la CPU.

Otro aspecto destacable que muestran los resultados es la importancia de trabajar con poblaciones de tamaño múltiplo de 32, se puede observar que se obtiene un mayor beneficio al utilizar la GPU cuando la cantidad de bloques es múltiplo de la cantidad de multiprocesadores disponibles (en este caso 16), y/o el tamaño de estos bloques es múltiplo de la cantidad de unidades de procesamiento por multiprocesador (en este caso 8).

En cuanto a los distintas versiones basadas en GPU puede notarse la importancia de minimizar los accesos a la memoria global de la GPU, utilizando el espacio de memoria compartida de los bloques siempre que sea posible. Realizar esto permitió a la versión GA(GPU-MC) obtener importantes mejoras en el tiempo de ejecución respecto del algoritmo serial, e incluso del GA(GPU). Si bien en principio se utilizan en el GA(GPU-MC) una cantidad de threads 10 veces mayor que en el GA(GPU), es solamente para realizar la carga de los datos en la memoria compartida, luego para realizar los cálculos se utiliza un solo thread por individuo.

Tamaño de la población	GA(GPU) speedup	GA(GPU-MC) speedup	GA(HPU) speedup
100	1,31	5,25	7,00
128	1,47	5,60	7,00
200	1,33	5,71	5,71
256	1,50	8,14	8,14
500	1,39	7,36	8,58
512	1,52	9,00	9,00
1000	1,37	6,48	7,38
1024	1,47	6,97	7,64
2000	1,33	4,85	4,97
2048	1,45	5,17	5,22
4000	1,34	3,46	3,53
4096	1,36	3,47	3,52

Tabla 2. Valores de speedup de las distintas versiones.

Aunque se observa que las versiones GA(GPU-MC) y GA(HPU) son los que obtienen las mejores resultados en cuanto al tiempo de ejecución también son los algoritmos con mayor degradación de la performance al aumentar el tamaño de la población. Probablemente esto se deba a una saturación de la GPU a causa de la cantidad de threads generados.

Si realizamos una comparación de tiempos de ejecución del GA(GPU-MC) y GA(HPU), se puede observar que el GA(HPU) logra una leve mejora respecto a los tiempos obtenidos por el GA(GPU-MC). Por lo tanto se puede pensar que esta estrategia de paralelismo puede conducir a tiempos de ejecución menores, que si solo se utilizara una GPU. Obviamente, un factor determinante para lograr esto es el porcentaje de distribución del trabajo entre la CPU y la GPU.

Otra característica de la versión GA(HPU), y vinculada a la necesidad de trabajar en GPUs con tamaños de población múltiplos de 32, es que permite evaluar en GPU la cantidad de individuos que se desee, independientemente de la cantidad total de individuos utilizados. En este sentido la Tabla 3 presenta algunos ejemplos de poblaciones con tamaño que no son potencia de 2 y los beneficios de ejecutar en GPU la evaluación de una cantidad potencia de 2 de individuos.

Tamaño de la población	Tamaño en GPU	GA(HPU) tiempo (seg)	GA(CPU) tiempo (seg)	Factor de mejora
200	192	6	40	6,67
538	512	13	120	9,23

Tabla 3. Tiempos de ejecución explotando tamaños de población multiplos de 32.

Observando los resultados de la Tabla 3 se puede concluir que explotar ambas técnicas, el trabajo asíncrono y la utilización de poblaciones de tamaño múltiplo de 32, es una línea de trabajo promisoria ya que en este caso se consiguieron los mejores valores de speedup, $9{,}23x$.

6. Conclusiones y Trabajo Futuro

En el trabajo se realizó un breve estudio del uso de GPUs para acelerar el cómputo de algoritmos genéticos. Se implementaron tres versiones de algoritmos genéticos paralelos explotando las capacidades de las GPUs modernas. Se evaluaron los diferentes espacios de memoria que maneja CUDA, y cómo afectan a la performance la forma en que se utilizan.

De los resultados obtenidos, puede observarse los beneficios de utilizar la GPU para computar GAs ya que las versiones que la utilizan logran buenos valores de speedup. También se destaca la importancia de minimizar los accesos a la memoria global de la GPU, utilizando el espacio de memoria compartida de los bloques siempre que sea posible. Realizar esto permitió al GA(GPU-MC) obtener importantes mejoras en el tiempo de ejecución respecto del algoritmo en CPU, e incluso del GA(GPU).

Otro aspecto a destacar es que los valores de mejora más alto se obtienen para tamaños de población que son múltiplos de 32. Esto se debe a que se realiza un mejor aprovechamiento de los recursos, ya que la tarjeta utilizada se compone de 16 multiprocesadores y cada uno a su vez posee 8 procesadores, por lo que al utilizar bloques de tamaño múltiplo de 8, y cantidad de bloques múltiplo de 16 hay una distribución más equitativa de la carga sobre los procesadores obteniendo así un mejor rendimiento.

La estrategia híbrida, GA(HPU), es la que obtiene en todos los casos los menores tiempos de ejecución. Mostrando las bondades de este tipo de estrategias de cómputo híbridas CPU-GPU que aprovechan el asincronismo entre la CPU y la GPU. En espacial cuando se hace en forma combinada para trabajar con poblaciones de tamaño múltiplo de 32 en GPU, ya que así se alcanzaron los mejores valores de speedup, $9{,}23x$.

Como conclusión general se puede afirmar el potencial del uso de GPUs para paralelizar GAs, ya que se obtuvieron valores de speedup cercanos a $10x$. Este resultado es más destacado si se considera que el aumento en la inversión económica debido a la GPU es de apenas un $30\,\%$.

Diversos tópicos no fueron abordados durante el trabajo por diferentes razones que ameritan un mayor estudio en trabajos futuros.

En primer término parece interesante extender las pruebas realizadas, en particular, evaluando otros problemas. Incluyendo problemas más realistas con instancias más complejas que tengan funciones de fitness de evaluación más costosas.

Siguiendo otra línea de trabajo futuro parece necesario evaluar otras configuraciones de arquitectura de hardware, utilizando computadoras con procesadores con más núcleos y otros modelos de GPUs.

Por último, resulta atractivo estudiar la implementación en GPUs de otros modelos de GAs paralelos, así como estudiar el uso de GPUs para paralelizar otras estrategias evolutivas.

Referencias

1. Cg website. `http://developer.nvidia.com/page/ cg_main.html`. Accessed on January 2010.
2. CUDA website. `www.nvidia.com/object/cuda_home.html`. Accessed on January 2010.
3. DIRECTX website. `http://developer.nvidia.com/page/ directx.html`. Accessed on January 2010.
4. OpenGL website. `http://www.opengl.org/`. Accessed on January 2010.
5. Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. on Graphics*, 23:777–786, 2004.
6. Erick Cantu-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.
7. M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, 21:948–960, 1972.
8. D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.
9. S. Harding. Evolution of image filters on graphics processor units using cartesian genetic programming. In Jun Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.
10. S. Harding and W. Banzhaf. Fast genetic programming and artificial developmental systems on GPUs. In *21st International Symposium on High Performance Computing Systems and Applications (HPCS'07)*, page 2, Canada, 2007. IEEE Computer Society.
11. John H. Holland. Outline for a logical theory of adaptive systems. *J. ACM*, 9(3):297–314, 1962.
12. John H. Holland. Genetic algorithms and the optimal allocation of trials. *SIAM J. Comput.*, 2(2):88–105, 1973.
13. Tony E. Lewis and George D. Magoulas. Strategies to minimise the total run time of cyclic graph based genetic programming with GPUs. In Guenther Raidl, Franz Rothlauf, Giovanni Squillero, Rolf Drechsler, et al., editors, *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1379–1386, Montreal, 8-12 July 2009. ACM.
14. Jian Ming Li, Xiao Jing Wang, Rong Sheng He, and Zhong Xian Chi. An efficient fine-grained parallel genetic algorithm based on GPU-Accelerated. In *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*, pages 855–862, 2007.
15. Jian Ming Li, Lihua Zhang, and Linlin Liu. A parallel immune algorithm based on fine-grained model with gpu-acceleration. In *ICICIC '09: Proceedings of the 2009 Fourth International Conference on Innovative Computing, Information and Control*, pages 683–686, Washington, DC, USA, 2009. IEEE Computer Society.
16. Ogier Maitre, Laurent A. Baumes, Nicolas Lachiche, Avelino Corma, and Pierre Collet. Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1403–1410, New York, NY, USA, 2009. ACM.

17. Shigeyoshi Tsutsui and Noriyuki Fujimoto. Solving quadratic assignment problems by genetic algorithms with gpu computation: a case study. In *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 2523–2530, New York, NY, USA, 2009. ACM.
18. Man Wong. Parallel multi-objective evolutionary algorithms on graphics processing units. In *GECCO '09: Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 2515–2522, New York, NY, USA, 2009. ACM.
19. Man Wong and Tien Wong. Parallel hybrid genetic algorithms on Consumer-Level graphics hardware. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 2973–2980, 2006.
20. Man Wong and Tien Wong. Implementation of parallel genetic algorithms on graphics processing units. In *Intelligent and Evolutionary Systems*, pages 197–216. 2009.
21. Man Wong, Tien Wong, and Ka Fok. Parallel evolutionary algorithms on graphics processing unit. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 3, pages 2286–2293 Vol. 3, 2005.
22. Qizhi Yu, Chongcheng Chen, and Zhigeng Pan. Parallel genetic algorithms on programmable graphics hardware. In *Advances in Natural Computation*, pages 1051–1059. 2005.