CU2rCU: a CUDA-to-rCUDA Converter

Carlos Reaño[1], Antonio J. Peña[1], Federico Silla[1], José Duato,[1]
Rafael Mayo[2] and Enrique S. Quintana-Ortí[2]

Resumen— **Las GPUs (*Graphics Processor Units*, unidades de procesamiento gráfico) están siendo cada vez más utilizadas en el campo de la HPC (*High Performance Computing*, computación de altas prestaciones) como una forma eficaz de reducir el tiempo de ejecución de las aplicaciones mediante la aceleración de determinadas partes de las mismas. CUDA (*Compute Unified Device Architecture*, arquitectura de dispositivos de cómputo unificado) es una tecnología desarrollada por NVIDIA que permite llevar a cabo dicha aceleración, proporcionando para ello una arquitectura de cálculo paralelo. Sin embargo, la utilización de GPUs en el ámbito de la HPC presenta ciertas desventajas, principalmente, en el coste de adquisición y el aumento de energía que introducen. Para hacer frente a estos inconvenientes se desarrolló rCUDA (*Remote CUDA*, CUDA remoto), una tecnología que permite compartir dispositivos CUDA de forma remota, reduciendo así tanto el coste de adquisición como el consumo de energía.**

En anteriores artículos de rCUDA quedó demostrada su viabilidad, pero también se identificaron algunos aspectos susceptibles de ser mejorados en relación con su usabilidad. Ésta se veía afectada por el hecho de que rCUDA no soporta las extensiones de CUDA al lenguaje C. De esta forma, era necesario convertir manualmente las aplicaciones CUDA eliminando dichas extensiones, y utilizando únicamente C plano. En este artículo presentamos una herramienta que realiza éstas conversiones de manera automática, permitiendo así adaptar las aplicaciones CUDA a rCUDA de una manera sencilla.

Palabras clave— **CUDA, computación de altas prestaciones, aceleración basada en GPUs, virtualización, Clang, herramientas de transformación de código.**

I. Introducción

Dado el alto coste computacional de las actuales aplicaciones de cálculo intensivo, muchos científicos piensan en aprovechar la gran potencia de las GPUs para incrementar el rendimiento de sus aplicaciones. A la hora de acelerar las aplicaciones mediante GPUs, las mejoras se obtienen realizando en éstas las partes computacionalmente intensivas. Para llevarlo a cabo, han aparecido diversas soluciones como, por ejemplo, CUDA [1] u OpenCL [2]. Todo ello ha derivado en lo que se conoce como GPGPU (*General Purpose Computing on GPUs*, computación de propósito general en GPUs), dando lugar a la utilización de estos dispositivos en áreas tan diversas como álgebra computacional [3], finanzas [4], equipamiento sanitario [5], dinámica de fluidos [6], física-química [7] o análisis de imágenes [8].

En la actualidad, el enfoque adoptado para utilizar GPUs en las instalaciones de altas prestaciones suele ser el de conectar uno o más aceleradores a cada nodo del clúster. Desde el punto de vista del consumo energético, esta práctica no es eficiente, ya que una única GPU puede suponer fácilmente el 25% del consumo total de un nodo. Por otro lado, las GPUs nunca son utilizadas el 100% del tiempo, por muy elevado que sea el nivel de paralelismo de una aplicación. Así pues, el hecho de reducir el número de GPUs en un clúster de estas características redundaría en una mayor utilización de las mismas, menor coste de adquisición y menor consumo.

Una posible solución para conseguir una configuración de clúster con menos GPUs que nodos es la virtualización. Esta técnica permite reducir los costes de adquisición, mantenimiento, administración, espacio y consumo energético en los sistemas de altas prestaciones. Con la virtualización de GPUs remotas, éstas son instaladas únicamente en algunos nodos, los cuales actuarán como servidores para el resto de nodos del clúster.

Con el objetivo de proporcionar una solución basada en esta idea se desarrolló rCUDA, un marco de trabajo que permite el uso concurrente de dispositivos CUDA de forma remota. En trabajos previos [9], [10], [11] nos centramos en demostrar que nuestra solución era factible. Sin embargo, se identificaron algunos aspectos susceptibles de mejora respecto a su usabilidad. Así, el marco de trabajo de rCUDA estaba limitado por la falta de soporte para las extensiones de CUDA al lenguaje C. Ello se debe a que la biblioteca del Runtime API de CUDA incluye varias funciones no documentadas que son llamadas implícitamente cuando se utilizan estas extensiones. De esta forma, para evitar el uso de dichas funciones no documentadas, rCUDA únicamente soporta la API de C plano de CUDA, lo cual hace necesario reescribir aquellas líneas de código de la aplicación que hacen uso de extensiones de CUDA.

En este artículo exponemos cómo hemos mejorado la usabilidad de rCUDA, desarrollando para ello una herramienta complementaria, `CU2rCU`, que automáticamente analiza el código fuente de la aplicación en busca de las líneas de código que deben ser modificadas para adaptar la aplicación original al marco de trabajo de rCUDA. El resto del artículo está estructurado de la siguiente manera: en la Sección II presentamos la tecnología rCUDA; en la Sección III se describe la herramienta `CU2rCU` desarrollada; a continuación, evaluamos dicha herramienta en la Sección IV; para finalizar, en la

[1]DISCA, Universitat Politècnica de València, 46.022 - Valencia (España), e-mail: {carregon, apenya}@gap.upv.es, {fsilla, jduato}@disca.upv.es.
[2]DICC, Universitat Jaume I (UJI), 12.071 - Castellón (España), e-mail: {mayo, quintana}@icc.uji.es.

Sección V comentamos las conclusiones y en la Sección VI posible trabajo futuro.

II. rCUDA: Remote CUDA

El marco de trabajo de rCUDA proporciona a las aplicaciones acceso transparente a GPUs instaladas en nodos remotos, de manera que las aplicaciones no son conscientes de que realmente están accediendo a dispositivos externos. Para ello, la arquitectura de rCUDA está estructurada como un sistema cliente-servidor distribuido, tal y como muestra la Figura 1. Así, las peticiones de las aplicaciones que requieren GPUs son redirigidas por el cliente de rCUDA al servidor a través de la capa de comunicaciones.

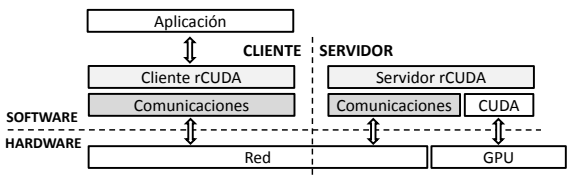

Fig. 1. Arquitectura de rCUDA.

Desde el punto de vista de las aplicaciones, el cliente rCUDA presenta la misma interfaz que el Runtime API de CUDA original, de tal forma que éstas se comportan de manera similar a como lo harían si tuvieran acceso local a las GPUs. Cuando el servidor recibe las peticiones, las procesa y ejecuta accediendo a las GPUs reales. Posteriormente, envía los resultados de nuevo al cliente.

Por su parte, la capa de comunicaciones que conecta el cliente rCUDA con el servidor proporciona una API propietaria de comunicaciones que posibilita la implementación de módulos de comunicaciones específicos para cada tecnología de red subyacente, permitiendo así el máximo aprovechamiento de la misma.

Desde un punto de vista global, el rendimiento conseguido con rCUDA debe ser previsiblemente menor que si utilizáramos directamente CUDA, puesto que el hecho de que las GPUs sean remotas introduce un sobrecoste en cuanto a comunicaciones se refiere. No obstante, el rendimiento obtenido utilizando rCUDA suele ser mucho mayor que el obtenido si realizamos los cálculos en CPUs.

La última versión de rCUDA está dirigida a sistemas operativos Linux, dando soporte a las mismas distribuciones que CUDA. Actualmente, rCUDA soporta la versión 4 del Runtime API de CUDA, a excepción de los módulos de interoperabilidad con gráficos, los cuales está previsto sean soportados en versiones futuras.

III. `CU2rCU`: Un Conversor de CUDA a rCUDA

En la presente sección explicamos la necesidad de un conversor de CUDA a rCUDA y presentamos la herramienta desarrollada con este propósito, `CU2rCU`.

A. Motivación

La API de CUDA permite a los programadores controlar qué operaciones se realizan en la CPU y cuáles en la GPU, simplificando de esta manera la programación GPGPU. CUDA extiende el lenguaje C con el objetivo de simplificar la programación de GPUs y hacerla más accesible. Normalmente, las extensiones van dirigidas a disminuir el número de líneas de código que obtendríamos si únicamente utilizáramos C plano.

A modo de ejemplo, el siguiente código muestra parte de un sencillo programa en CUDA. En el mismo, las funciones *cudaMalloc* (línea 13), *cudaMemcpy* (líneas 15 y 19) y *cudaFree* pertenecen a la API de C plano de CUDA, mientras que la sentencia de la línea 17 utiliza una de las extensiones comentadas:

```
1  #include <cuda.h>

3  // Código a ejecutarse en la GPU
4  __global__ void helloWorld(char* str) {
5    // procesamiento en la GPU.
6  }

8  // Código a ejecutarse en la CPU
9  int main(int argc, char **argv) {
10   char h_str[] = ''Hello World!'';
11   // ...
12   // reservamos espacio en memoria de la GPU
13   cudaMalloc((void**)&d_str, size);
14   // copiamos los datos necesarios a la GPU
15   cudaMemcpy(d_str, h_str, size,
                 cudaMemcpyHostToDevice);
16   // ejecutamos función en la GPU
17   helloWorld<<< BLOCKS, THREADS >>>(d_str);
18   // copiamos los resultados de la GPU
19   cudaMemcpy(h_str, d_str, size,
                 cudaMemcpyDeviceToHost);
20   // liberamos espacio en memoria de la GPU
21   cudaFree(d_str);
22   // ...
23 }
```

Un programa CUDA, como por ejemplo el anterior, se compila utilizando el compilador `nvcc` [12] que proporciona NVIDIA, el cual separa el código a ejecutar en la CPU del código a ejecutar en la GPU, y los compila de forma separada. Además, durante el proceso de compilación del código CPU se introducen referencias a funciones no documentadas. Ello dificulta la creación de herramientas que reemplacen directamente la biblioteca del Runtime API de CUDA original, como ocurre en el caso de rCUDA. Así pues, para evitar el uso de estas funciones no documentadas, necesitamos compilar el código de CPU con un compilador de C plano en lugar de compilarlo con `nvcc`. Como contrapartida, el hecho de no utilizar `nvcc` impide que utilicemos las extensiones del lenguaje C propias de CUDA, de tal forma que dichas extensiones deberían convertirse a C plano antes de la compilación. Como la tarea de realizar estos cambios manualmente en programas de gran tamaño puede ser muy tediosa, e incluso puede provocar la introducción de errores en el código, hemos desarrollado una herramienta que los

realiza de forma automática.

Retomando el programa de ejemplo mostrado con anterioridad, a continuación ofrecemos el código equivalente en C plano a la sentencia de la línea 17, la cual era la única que empleaba la sintaxis extendida de CUDA:

```
1 #define ALIGN_UP(offset, align) (offset) = \
2 ((offset) + (align) - 1) & ~((align) - 1)

4 int main() {
5   // ...
6   cudaConfigureCall(BLOCKS, THREADS);
7   int offset = 0;
8   ALIGN_UP(offset, __alignof(d_str));
9   cudaSetupArgument(&d_str, sizeof(d_str),
                      offset);
10  cudaLaunch(''helloWorld'');
11  // ...
12 }
```

A la hora de compilar un programa CUDA para que sea ejecutado posteriormente dentro del marco de trabajo de rCUDA, éste debe ser dividido en dos partes:

- Código CPU: ejecutado en la CPU y compilado con un compilador de C como puede ser `gcc`. No debe tener extensiones CUDA, únicamente C plano. Tampoco debe tener código GPU. Este código se obtiene tras aplicar el conversor desarrollado al código original.
- Código GPU: ejecutado en la GPU y compilado con `nvcc`. El propio compilador `nvcc` permite compilar únicamente el código GPU, descartando el código CPU, y generar un archivo binario con el resultado de dicha compilación.

En la Figura 2 se ilustra de manera gráfica este proceso. Mediante `nvcc` obtendríamos el código GPU en un fichero binario. Aplicando el conversor al código original tendríamos el código CPU en C plano, el cual compilaríamos, por ejemplo, con `gcc` para obtener un ejecutable del código CPU. Éste último tendría referencias al código GPU del fichero binario generado previamente.

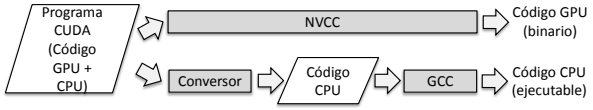

Fig. 2. Proceso de adaptación de un programa CUDA al marco de trabajo de rCUDA.

B. Herramientas para la transformación de código

Para implementar nuestro conversor de código hemos utilizado una de las denominadas herramientas de transformación de código a código (*source-to-source transformation tools*). Entre las diferentes opciones disponibles encontramos desde simples herramientas que sustituyen patrones de texto, hasta complejos entornos de trabajo que analizan sintácticamente el código fuente, generando lo que se denomina un AST (*Abstract Syntax Tree*, árbol de sintaxis abstracta), y que permiten transformar el código en base a dicha información. Dado que, como veremos en la Sección III-C, nuestro conversor necesita realizar transformaciones complejas, únicamente nos centraremos en estas últimas.

Existen varios marcos de trabajo que nos permiten llevar a cabo transformaciones complejas como, por ejemplo, ROSE [13], GCC [14], y Clang [15]. Entre ellos, hemos decidido utilizar Clang puesto que, por un lado, es ampliamente utilizado y, por otro, soporta explícitamente programas escritos en CUDA. Además, existen otros conversores de código CUDA que también están basados en Clang, como `CU2CL` [16].

Clang, uno de los subproyectos primarios de LLVM [17], es un compilador de C que tiene como objetivo, entre otros, facilitar una plataforma para construcción de herramientas a nivel de código, incluyendo conversores de código.

En la Figura 3 se muestra cómo la herramienta que hemos desarrollado interacciona con Clang. La entrada al conversor es el código fuente original escrito en CUDA, el cual contiene código a ejecutar tanto en la CPU como en la GPU. El *driver* de Clang (un driver del compilador que proporciona acceso al propio compilador de Clang y a las herramientas que éste dispone) analiza el código fuente generando un AST. A continuación, el *plugin* para Clang que hemos desarrollado utiliza la información que proporciona el AST para realizar las transformaciones necesarias, generando un nuevo código fuente que únicamente contiene código a ejecutar en la CPU en C plano. Cabe destacar que durante el proceso de conversión nuestra herramienta también analiza los fuentes incluidos en otros fuentes, convirtiéndolos si fuera necesario.

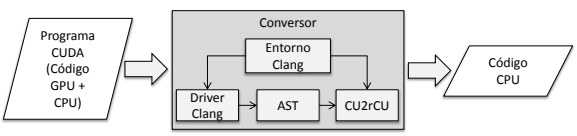

Fig. 3. Proceso de conversión de CUDA a rCUDA detallado.

C. Transformaciones de código realizadas por `CU2rCU`

Como ya hemos explicado, nuestro conversor transforma el código inicial escrito en CUDA en código que únicamente utiliza la API de C plano de CUDA. Para ello, convierte las extensiones de CUDA al lenguaje C en su equivalente en C plano. Además, también elimina el código a ejecutar en la GPU, de forma que el código obtenido por nuestro conversor sólo contiene código a ejecutar en la CPU. A conti-

nuación detallamos algunas de las transformaciones más importantes que realiza el conversor.

C.1 Kernels

Dentro de la terminología de CUDA, se denomina *kernel* a una función definida por el programador que se ejecutará en la GPU. Por ejemplo, la función *helloWorld* en la línea 4 del código de ejemplo visto en la Sección III-A es un *kernel*. A la hora de realizar una llamada a un *kernel* utilizando las extensiones de CUDA a C, como pudiera ser mediante una expresión del tipo:

```
miKernel <<< Dg, Db >>> (arg1,...,argN);
```

ésta debe ser transformada para utilizar la API de CUDA en C plano de la siguiente forma:

```
cudaConfigureCall(Dg, Db);
int offset = 0;
setupArgument(arg1, &offset);
setupArgument(..., &offset);
setupArgument(arg2, &offset);
cudaLaunch(``miKernel'');
```

La función `setupArgument()` se proporciona como parte del marco de trabajo de rCUDA. Simplemente simplifica la llamada a la función de la API de C plano `cudaSetupArgument()`, eliminando la necesidad de tener que gestionar explícitamente los *offsets*.

Cabe destacar que, en la llamada a la función `cudaLaunch()` del ejemplo anterior, el valor a utilizar como cadena de caracteres que identifica al *kernel*, *"miKernel"* en nuestro ejemplo, varía en función de cómo haya sido declarado el *kernel*:

- Si el *kernel* fue declarado como una función externa de C (*extern "C"*), se debe utilizar el propio nombre del *kernel* como argumento de `cudaLaunch()`. Por ejemplo, si tenemos la siguiente declaración de *kernel*:

  ```
  extern ``C'' __global__ void
      increment_kernel(int* x, int y);
  ```

  la llamada a `cudaLaunch()` se realizaría de la siguiente manera:

  ```
  cudaLaunch(``increment_kernel'');
  ```

- En caso contrario, se debe utilizar como argumento de `cudaLaunch()` el nombre *mangled*[1] del *kernel*. Por ejemplo, si el *kernel* fue declarado como sigue:

  ```
  __global__ void
      increment_kernel(int* x, int y);
  ```

  la llamada a `cudaLaunch()` sería así:

  ```
  cudaLaunch(``_Z16increment_kernelPii'');
  ```

La obtención del nombre *mangled* se convierte en una tarea compleja cuando se trata de *kernels* donde el tipo de los argumentos depende de la llamada realizada al *kernel*, no únicamente de cómo este fue declarado. Por ejemplo, dado el siguiente *kernel*:

```
template<class TData> __global__
  void testKernel(TData *d_odata,
                  TData *d_idata,
                  int numElements);
```

el nombre *mangled* del *kernel* a utilizar depende del tipo que tenga `TData`:

```
if((typeid(TData)==typeid(unsigned char))) {
  cudaLaunch(``_Z10testKernelIhEvPT_S1_i'');
} else if((typeid(TData)==
           typeid(unsigned short))) {
  cudaLaunch(``_Z10testKernelItEvPT_S1_i'');
} else if((typeid(TData)==
           typeid(unsigned int))) {
  cudaLaunch(``_Z10testKernelIjEvPT_S1_i'');
}
```

C.2 Symbols

Cuando hablamos de *symbols* en CUDA nos referimos a variables que residen en la memoria de la GPU. Éstos pueden ser referenciados mediante una variable declarada en código GPU o mediante una cadena de caracteres que nombra a una variable declarada en la GPU. Como el código de GPU es eliminado por nuestro conversor, únicamente podemos utilizar la segunda opción. Por ello, cuando el conversor encuentra una referencia a un *symbol* usando la primera opción, como por ejemplo la siguiente llamada:

```
__constant__ float symbol[256];
float src[256];
cudaMemcpyToSymbol(symbol, src,
                   sizeof(float)*256);
```

el argumento `symbol` tiene que ser entrecomillado para transformarlo en una cadena de caracteres:

```
cudaMemcpyToSymbol(``symbol'', src,
                   sizeof(float)*256);
```

Hemos de tener en cuenta que el argumento `symbol` también puede ser una macro. En ese caso, no bastaría con entrecomillarlo, sino que debería entrecomillarse el resultado de dicha macro. Por ejemplo, en la siguiente llamada:

```
#define MY_CONST(var) (``prx_'' #var)

__constant__ float symbol[256];
float src[256];
cudaMemcpyToSymbol(MY_CONST(symbol), src,
                   sizeof(float)*256);
```

es el resultado de la macro `MY_CONST(symbol)` lo que debe entrecomillarse:

```
cudaMemcpyToSymbol(``prx_symbol'', src,
                   sizeof(float)*256);
```

[1] Del inglés *mangled name*. También conocido como nombre "decorado".

C.3 Textures y Surfaces

CUDA soporta un subconjunto de texturizado hardware que la GPU utiliza en los gráficos para acceder a las memorias denominadas *texture* y *surface*. De ahora en adelante, nos referiremos a las variables que referencian a estas memorias como *textures* y *surfaces*.

De forma similar a como ocurre con las extensiones de CUDA a C, para utilizar las funciones de alto nivel de la API de C++ disponible en el Runtime API de CUDA, una aplicación necesita ser compilada con el compilador `nvcc` de NVIDIA. Ya hemos comentado que, para ejecutar una aplicación en el marco de trabajo de rCUDA, ésta debe ser compilada con un compilador de C. En consecuencia, necesitamos transformar también dichas funciones. Éste es el caso de las *textures* y *surfaces*. Así, las *textures* declaradas utilizando esta API, como:

```
texture<float, 2> textureName;
```

son transformadas como sigue:

```
textureReference *textureName;
cudaGetTextureReference((const
              textureReference **)
             &textureName, ''textureName'');
```

Como consecuencia de esta transformación, las variables que referencian a *textures* se convierten en punteros y, por lo tanto, el acceso a sus atributos como, por ejemplo:

```
textureName.attribute = value;
```

pasaría a ser así:

```
textureName->attribute = value;
```

Las mismas transformaciones comentadas para *textures* son validas para las *surfaces*.

IV. Evaluación

En esta sección se describen los experimentos que hemos realizado para evaluar la herramienta desarrollada `CU2rCU`. Para ello, hemos utilizado algunas de las aplicaciones disponibles en la *NVIDIA GPU Computing SDK* [18], que cubren un amplio rango de técnicas relativas a la programación con CUDA.

Para los experimentos hemos utilizado un ordenador de sobremesa equipado con un procesador Intel(R) Core(TM) 2 DUO E6750 (2.66GHz, 2GB RAM) y una tarjeta gráfica GeForce GTX 590, bajo sistema operativo Linux (Ubuntu 10.04).

En la Tabla I podemos ver los tiempos de conversión de las distintas aplicaciones evaluadas, así como la cantidad de líneas del código original y el número de líneas modificadas o añadidas para convertir ese código. Tanto los tiempos de conversión mostrados en dicha tabla, como los tiempos de compilación que mostraremos más adelante en esta misma sección, son el resultado de repetir las mediciones de tiempo hasta que la desviación típica ha sido inferior al 5%.

TABLA I
Estadísticas de Conversión de las Aplicaciones de la SDK de NVIDIA

Aplicación de la SDK de NVIDIA	Tiempo (s)	Líneas	
		Código Original	Modif./ Añadidas
alignedTypes	0.259	186	32
asyncAPI	0.191	78	6
bandwidthTest	0.407	708	0
BlackScholes	0.364	281	13
clock	0.196	75	8
concurrentKernels	0.196	100	11
convolutionSeparable	0.591	319	18
cppIntegration	0.685	129	12
dwtHaar1D	0.221	266	11
fastWalshTransform	0.360	241	20
FDTD3d	1.082	860	13
inlinePTX	0.351	91	6
matrixMul	0.394	272	34
mergeSort	0.917	1124	105
scalarProd	0.358	138	10
scan	0.548	359	26
simpleAtomicIntrinsics	0.367	211	6
simpleMultiCopy	0.202	211	22
simpleTemplates	0.211	241	13
simpleVoteIntrinsics	0.196	222	19
SobolQRNG	1.278	10586	8
sortingNetworks	0.761	571	70
template	0.357	97	7
vectorAdd	0.192	88	8

La suma total de tiempo necesario para convertir todas las aplicaciones evaluadas es de 10.68 segundos. Si comparamos este tiempo con el empleado por un experto del equipo de rCUDA para convertir las mismas aplicaciones de forma manual, 31.5 horas, queda claramente demostrado el beneficio de utilizar el conversor. Aunque la conversión manual produce como resultado unas pocas líneas de código menos, el resultado final es bastante similar al código obtenido de forma automática por el conversor.

Adicionalmente, también hemos comparado el tiempo empleado en compilar el código original de las aplicaciones con el tiempo de convertir y compilar el código convertido de las mismas. En la Figura 4 se muestran los resultados, donde podemos observar que el tiempo de convertir el código original y posteriormente compilarlo para su ejecución con rCUDA es similar al tiempo de compilar el código original para ejecutarlo directamente en CUDA.

V. Conclusiones

Inicialmente, la usabilidad de rCUDA estaba limitada por el hecho de que no soportaba las extensiones de CUDA al lenguaje C, siendo necesario reescribir las partes del código original CUDA que hacían uso de estas extensiones. Con el objetivo

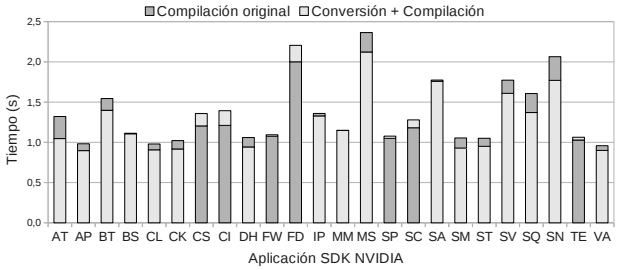

Fig. 4. Tiempo de compilación de la SDK de NVIDIA comparado con el tiempo de conversión con `CU2rCU` y posterior compilación.

de resolver esta limitación hemos desarrollado un conversor de CUDA a rCUDA, `CU2rCU`. Se trata de una herramienta complementaria al marco de trabajo de rCUDA que, automáticamente, analiza las aplicaciones originales escritas en CUDA en busca de las partes de código que deben ser modificadas para adaptar la aplicación a los requisitos de rCUDA. De esta forma, el conversor realiza los cambios necesarios sin que el programador de la aplicación original deba intervenir.

La evaluación del nuevo conversor `CU2rCU` ha sido realizada utilizando códigos de ejemplo de la *NVIDIA GPU Computing SDK*, que contiene un amplio rango de aplicaciones y técnicas de programación con CUDA. Los resultados obtenidos han demostrado que la utilización del conversor es muy beneficiosa en cuanto al tiempo empleado en convertir una aplicación se refiere.

VI. Trabajo Futuro

Como trabajo futuro, está previsto que el conversor `CU2rCU` sea integrado en el flujo de compilación de rCUDA. De esta forma, sería posible reemplazar la llamada al compilador `nvcc` de NVIDIA por el equivalente en rCUDA, el cual internamente aplicaría el conversor y, posteriormente, llamaría a los compiladores necesarios.

Agradecimientos

El personal de la UPV ha sido subvencionado por el Ministerio de Ciencia e Innovación (MICINN), con fondos del Plan E, bajo el acuerdo TIN2009-14475-C04-01 y también por el programa PROMETEO de la Generalitat Valenciana (GVA) bajo el acuerdo PROMETEO/2008/060. Por otro lado, el personal de la UJI ha sido financiado por el Ministerio de Ciencia español, el programa FEDER (contrato TIN2011-23283) y la Fundación Caixa-Castelló/Bancaixa (contrato P1-1B2009-35).

Referencias

[1] NVIDIA, *The NVIDIA CUDA API Reference Manual*, NVIDIA, 2011.

[2] A. Munshi, Ed., *OpenCL 1.0 Specification*, Khronos OpenCL Working Group, 2009.

[3] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Ortí, and G. Quintana-Ortí, *Exploiting the capabilities of modern GPUs for dense matrix computations*, Concurr. Comput. : Pract. Exper., vol. 21, no. 18, pp. 2457-2477, 2009.

[4] A. Gaikwad and I. M. Toke, *GPU based sparse grid technique for solving multidimensional options pricing PDEs*, in Proceedings of the 2nd Workshop on High Performance Computational Finance, D. Daly, M. Eleftheriou, J. E. Moreira, and K. D. Ryu, Eds. ACM, Nov. 2009.

[5] S. S. Stone, J. P. Haldar, S. C. Tsao, W. Hwu, Z.P . Liang, and B. P. Sutton, *Accelerating advanced MRI reconstructions on GPUs*, in Proceedings of the 2008 conference on Computing Frontiers (CF'08), pp. 261–272. ACM, New York, 2008.

[6] E. H. Phillips, Y. Zhang, R. L. Davis, and J. D. Owens, *Rapid aerodynamic performance prediction on a cluster of graphics processing units*, in Proceedings of the 47th AIAA Aerospace Sciences Meeting, no. AIAA 2009-565, Jan. 2009.

[7] D. P. Playne and K. A. Hawick, *Data parallel three-dimensional Cahn-Hilliard field equation simulation on GPUs with CUDA*, in International Conference on Parallel and Distributed Processing Techniques and Applications, H. R. Arabnia, Ed., 2009, pp. 104–110.

[8] Y. C. Luo and R. Duraiswami, *Canny edge detection on NVIDIA CUDA*, in Computer Vision on GPU, 2008.

[9] J. Duato, F. D. Igual, R. Mayo, A. J. Peña, E. S. Quintana-Ortí, and F. Silla, *An efficient implementation of GPU virtualization in high performance clusters*, in Euro-Par 2009 Workshops, ser. LNCS, vol. 6043, 2010, pp. 385-394

[10] J. Duato, A. J. Peña, F. Silla, R. Mayo and E. S. Quintana-Ortí, *Performance of CUDA Virtualized Remote GPUs in High Performance Clusters*, in International Conference on Parallel Processing 2011, pp. 365–374, 2011.

[11] J. Duato, A. J. Peña, F. Silla, J. C. Fernández, R. Mayo and E. S. Quintana-Ortí, *Enabling CUDA Acceleration within Virtual Machines using rCUDA*, in International Conference on High Performance Computing 2011, Bangalore, 2011.

[12] NVIDIA, *The NVIDIA CUDA Compiler Driver NVCC*, NVIDIA, 2011.

[13] D. Quinlan, C. Liao, T. Panas, R. Matzke, M. Schordan, R. Vuduc, and Q. Yi, *ROSE User Manual: A Tool for Building Source-to-Source Translators*, online: http://www.rosecompiler.org, último acceso: Mayo 2012.

[14] Free Software Foundation, Inc. , *GCC, the GNU Compiler Collection*, online: http://gcc.gnu.org/, último acceso: Mayo 2012.

[15] LLVM, *Clang: a C language family frontend for LLVM*, online: http://clang.llvm.org/, último acceso: Mayo 2012.

[16] G. Martinez and W. Feng and M. Gardner, *CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-core Architectures*, online: http://eprints.cs.vt.edu/archive/00001161/01/CU2CL.pdf, último acceso: Mayo 2012.

[17] LLVM, *The LLVM Compiler Infrastructure*, online: http://llvm.org/, último acceso: Mayo 2012.

[18] NVIDIA, *The NVIDIA GPU Computing SDK*, NVIDIA, 2011.

[19] Semantic Designs, Incorporated, *The DMS Software Reengineering Toolkit*, online: http://www.semanticdesigns.com/Products/DMS/DMS Toolkit.html, último acceso: Mayo 2012.

[20] Q. Yi, *POET: a scripting language for applying parameterized source-to-source program transformations*, in Software: Practice and Experience, vol. 42, pp. 675-706, 2012.

[21] G. Rudy, M. Khan, M. Hall, C. Chen, and J. Chame, *A Programming Language Interface to Describe Transformations and Code Generation*, in Lecture Notes in Computer Science, vol. 6548, pp. 136-150, 2011.

[22] E. Visser, *Program Transformation with Stratego/XT*, in Lecture Notes in Computer Science, vol. 3016, pp. 315-349, 2004.

[23] G. Necula, S. McPeak, S. Rahul, and W. Weimer, *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*, in Lecture Notes in Computer Science, vol. 2304, pp. 209-265, 2002.

[24] LLVM, *The Clang Compiler User's Manual*, online: http://clang.llvm.org/docs/UsersManual.html, último acceso: Mayo 2012.

[25] LLVM, *The LLVM Programmer's Manual*, online: http://llvm.org/docs/ProgrammersManual.html, último acceso: Mayo 2012.