Facultad de Ciencias

FOTV: DESCARGA DE CÓDIGO A ACELERADORES GENÉRICOS EN OPENMP

(FOTV: Offloading code to generic accelerator devices with OpenMP)

Trabajo de Fin de Máster
para acceder al

MÁSTER EN INGENIERÍA INFORMÁTICA

Autor: Jose Luis Vázquez Gutiérrez

Director\es: Pablo Pedro Sánchez Espeso

Octubre – 2020

Indice

Terminología

- AST: Abstract Syntax Tree. Es una representación del código empleada internamente por el frontend antes de generar código IR, en la que cada nodo del árbol representa un símbolo parseado del código y cada arista una relación de dependencia, con las hojas siendo las unidades mínimas. De cada nodo se incluye su localización inicial y final, entre otros metadatos.
- Backend: La parte de un compilador dependiente del dispositivo que va a ejecutar el programa. Recibe un IR y al final del proceso genera un binario. Está formado por uno o varios pases de optimización y transformación de código.
- Cloud: Recursos de cómputo accesibles bajo demanda situados en un centro de datos, normalmente remoto.
- CUDA: Compute Unified Device Architecture, lenguaje de programación propietario de nVidia para la ejecución de código general en su gama de tarjetas gráficas. Está orientado a la aceleración de código basada en paralelismo de datos (data parallelism).
- Driver: El ejecutable de un compilador. Recibe los parámetros y de ellos extrae las opciones de frontend, backend, linker y otros subsistemas y orquesta la ejecución de todos ellos. Clang es un driver.
- Fat Binary: Archivo ejecutable que contiene no sólo el código de host si no también binarios para los distintos dispositivos aceleradores.
- FOTV: Future Offload Target Virtualization, es el nombre que le hemos dado al target virtual y al sistema para su funcionamiento. Es una referencia a la posibilidad de descargar el código a targets desconocidos sin necesidad de re-implementar el plugin y backend del compilador cada vez que se incluya un nuevo dispositivo en el sistema.
- FPGA: Field Programmable Gate Array, un tipo de hardware que permite la reconfiguración del circuito interno para propósitos específicos. Tienden a tener mayor rendimiento por vatio que soluciones software.
- Frontend: La parte del compilador dependiente del lenguaje de programación. Recibe los códigos fuente y genera uno o varios módulos IR, generalmente uno por entrada.

- GPGPU: General Purpose GPU, la tecnología que permite ejecución de código general dentro de una GPU.
- Host: Computador en donde se ejecuta el código que no es acelerado en un dispositivo específico. El host controla la ejecución del programa y la distribución de tareas a los diferentes targets. En OpenMP, el host ejecuta todo el código que no es ejecutado en una “Target Region”.
- IR: Intermediate Representation, o representación intermedia. Es la representación interna del código en el compilador. Para LLVM/Clang, dicha representación tiene múltiples formas intercambiables, entre ellas la forma textual de código ensamblador para un computador genérico.
- Kernel: Cada uno de los bloques ordenados de código que se van a acelerar. El nombre viene del estándar OpenCL, utilizándose también en CUDA. En OpenMP se denominan “Target Regions”. El trabajo emplea ambos.
- LLVM/Clang: El compilador sobre el que vamos a trabajar. Concretamente, LLVM es el framework de construcción de compiladores con el gestor de pases y backends, mientras que Clang es el frontend y driver para C y C++.
- Offloading: Técnica de descarga de código intensivo en cómputo a dispositivos aceleradores que ofrezcan mayores prestaciones.
- Pase de optimización: Transformación IR → IR. Puede ser una optimización pura, una anotación, una extracción o una generación de código binario.
- SPMD: Single Program, Multiple Data. Paradigma de programación paralela que consiste en lanzar el mismo programa en paralelo en múltiples hilos con diferentes conjuntos de datos y aplicar reducción a los resultados.
- Target virtual: El dispositivo virtual en el que se centra el trabajo. Dicho dispositivo no está asociado a un hardware específico por lo que se implementa como una máscara para x86_64 que permite que OpenMP cargue el código generado en el proyecto sin necesidad de modificar la librería estándar.
- Warp: Unidad mínima de planificación en una GPU. En nVidia corresponde a 32 threads, y en AMD a 64 threads.

1.- Introducción

En los últimos años, proyectos de computación a través del planeta están actualizando sus infraestructuras para buscar mayor potencia y eficiencia, y para ello han dirigido sus miras a los sistemas de computación heterogénea. Estos sistemas se caracterizan por estar compuestos de distintos tipos de sistemas de cómputo, generalmente un dispositivo host (típicamente una CPU) y uno o varios dispositivos aceleradores de cómputo a los que se descargan las partes intensivas del trabajo (típicamente GPGPUs). Este tipo de arquitectura requiere de herramientas de desarrollo específicas, y a día de hoy existen múltiples paradigmas de programación para este tipo de sistemas, como OpenCL u OpenACC entre otros.

Siguiendo esta tendencia, el consorcio OpenMP decidió incluir en la versión 4.0 de su librería la opción para descargar código a dispositivos aceleradores mediante la directiva target(1), representada por `#pragma omp target`. Dicha directiva identifica secciones de código que se descargan a dispositivos aceleradores. Por su parte, el compilador se encarga de transformar la directiva en una serie de llamadas para cargar el código en el dispositivo target, transferir los datos necesarios para la ejecución, ejecutar el código y recoger las salidas del cómputo para continuar la ejecución del programa. Dicha aproximación fue ampliamente acogida por la comunidad científica debido a la simplicidad del sistema. Al permitir la combinación con otras directivas ya conocidas de OpenMP, se podía tener un código listo para su ejecución acelerada añadiendo un par de directivas al código C, C++ o Fortran que se tenía de base.

Dicha aproximación tiene una limitación(2): Sólo es posible generar código de descarga en tiempo de compilación, y eso significa que sólo se puede descargar código a los dispositivos que el propio compilador soporte. Esfuerzos realizados en GCC y LLVM/Clang permiten a día de hoy la descarga de código a CPUs soportadas[1] del host y GPUs de nVidia, con el soporte para las GPUs de AMD en fase de desarrollo inicial.

Existen trabajos (3, 4), que están dirigidos a emplear OpenMP para la descarga de código a diversas plataformas empleando la metodología estándar. Aunque interesantes, mantienen la misma limitación de ser dirigidas a un dispositivo específico cuyo soporte se debe incluir dentro del compilador.

1 Se soportan arquitecturas x86_64, aarch64 (ARM 64 bits) y PowerPC

El objetivo de este trabajo es el de extender la funcionalidad de la directiva target de OpenMP para permitir la descarga de código a un dispositivo genérico, cuyo código específico será cargado de manera dinámica en tiempo de ejecución, y acompañarlo de un sistema de extracción de código fuente que permita la optimización y compilación del código para un acelerador específico no soportado por OpenMP. Para ello modificaremos la etapa de generación de código de LLVM/Clang 11.0.0, creando una generación específica OpenMP para un nuevo dispositivo. También añadiremos un dispositivo genérico al que descargar dicho código de acelerador, para tener soporte en el compilador. Además, incluiremos dos pases de optimización, uno dirigido a funciones y otro a módulos, destinados a extraer regiones de código e información sobre datos a transferir respectivamente, basados en metadatos añadidos en la generación de código personalizada. Por último, enlazaremos todo este sistema con una librería en tiempo de ejecución que se encargará de la carga y ejecución dinámica de funciones a descargar, contando con la capacidad de planificación dinámica de recursos en función de los dispositivos cargados.

El trabajo se organiza en 5 secciones, contando esta primera introducción. En la siguiente sección analizaremos los trabajos relacionados con este desarrollo, y las razones por las que nuestra aproximación puede cubrir sus deficiencias. En la tercera sección hablaremos del diseño del sistema, como implementar dicho diseño y el resultado final. En la cuarta sección presentaremos la primera versión funcional, tanto su implementación como su ejecución y resultados preliminares. Por último, hablaremos de los trabajos que quedan por hacer para lanzar una versión pública, posibles ideas para mejorar futuras versiones y agradecimientos. También se incluye una bibliografía con todos los trabajos referenciados.

2.-Trabajos Relacionados

Conceptos previos de compiladores

Para poder contextualizar el trabajo que se va a presentar, es necesario hacer una breve introducción del funcionamiento de un compilador. Empezaremos por un detalle semántico: ni Clang es un compilador, ni LLVM no es un compilador, sino que ambos son partes del conjunto que forma un compilador.

En concreto, Clang un frontend para la familia C de lenguajes, LLVM/opt es el optimizador de código y un backend específico para una arquitectura de procesador que también forma parte de LLVM. Por otra parte, el proceso de compilación integra un ensamblador (generalmente as, aunque LLVM tiene un ensamblador integrado) y un linker (en sistemas GNU/Linux el estándar es ld, aunque también existen gold y, dentro del proyecto LLVM, lld).

LLVM es un marco de desarrollo de compiladores agnóstico al procesador, que provee un completo conjunto de librerías y estructuras para el desarrollo de frontends, backends y drivers junto con el optimizador, que no toma decisiones que sean exclusivas de una arquitectura para permitir la construcción de forma nativa de un compilador cruzado.

Cuando se señala que LLVM/Clang integra tres partes – frontend, backend, optimizador – se debe a que Clang es también el driver del compilador. El driver es una suerte de interfaz de usuario que el programador invoca, bien manualmente o bien de mano de su sistema de builds, para obtener un ejecutable a partir de su código fuente. El driver se encarga de leer los parámetros que recibe de la línea de comandos y gestionar y organizar el trabajo de compilador. Generalmente el proceso de compilación va a consistir en 5 sencillos pasos:

- Invocar el frontend con el código fuente como entrada. El frontend analiza y transforma el código para generar una representación intermedia (o IR) interna del compilador[2] para transferir entre procesos.
- Optimización del IR. En este proceso se pueden realizar múltiples optimizaciones a nivel IR, como desenrrollar bucles cortos, seleccionar instrucciones específicas del procesador objetivo para incrementar el rendimiento, eliminar código

2 Aunque esto es técnicamente cierto, LLVM permite su inspección externa

muerto (incluyendo constantes que no se utilicen) y otras acciones. La salida es otro IR, con todas las optimizaciones realizadas.

- El IR optimizado se transfiere al backend, un último pase de transformación que genera el código ensamblador para la arquitectura de procesador objetivo.
- Dicho código ensamblador se transfiere al ensamblador, que genera un objeto binario representativo del código que se ha introducido como entrada original.
- El binario se traspasa al linker, junto con todos los objetos binarios que sean necesarios, librerías a las que se llama y en ocasiones librerías dinámicas que puedan ser solicitadas, para generar el ejecutable final.

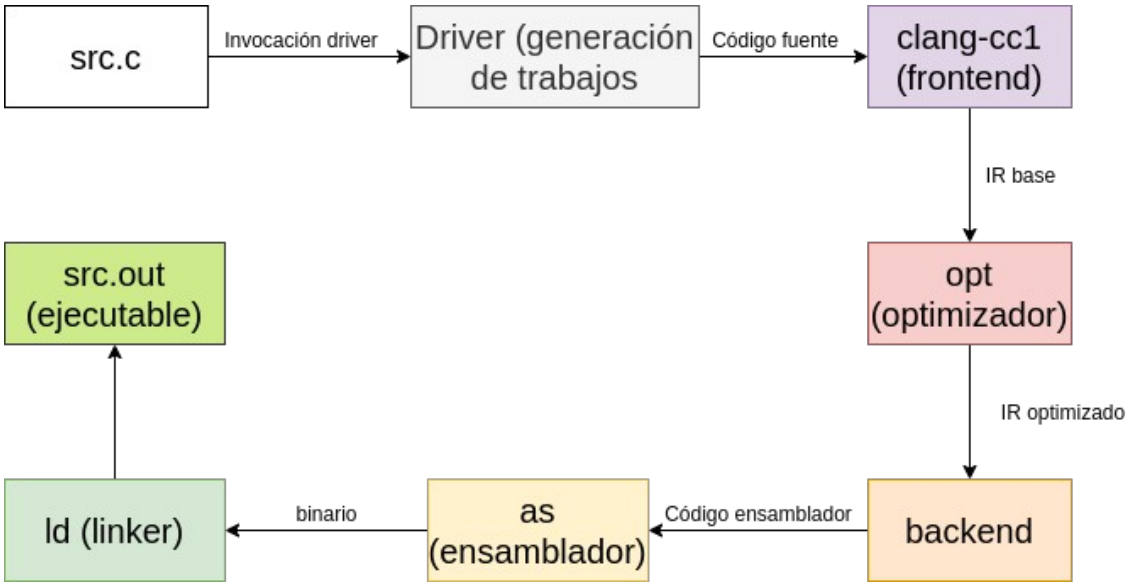

Figura 1: Esquema explicativo de un trabajo de compilación, como resumen del punto anterior

Existen varias razones que hacen a LLVM/Clang un proyecto atractivo sobre el que realizar proyectos de investigación. La primera y más importante es que se trata de un proyecto de código abierto, por lo que se puede acceder al código fuente sin problemas de licencias. En concreto, este proyecto se realizó sobre la versión 11.0.0rc2. Otras razones son su adopción a nivel profesional – proyectos comerciales compilados con Clang incluyen Chrome, el pack de códecs ffmpeg o Firefox entre otros – o la particularidad de ser uno de los pocos compiladores que mantienen la estructura de subtrabajos de forma rígida. Otros compiladores modernos tienden a fusionar algunos pasos para permitir compilación parcial, lo que a su vez permite realizar “compilaciones” rápidas para comprobar sintaxis por parte de IDEs. LLVM/Clang mantiene esta estructura en detrimento de su

adaptabilidad a esos entornos, lo que el consorcio LLVM afirma produce código de mejor calidad y mayor rendimiento.

Otra razón que justifica la popularidad de LLVM/Clang en cualquier trabajo de investigación o personalización es su IR. El IR de LLVM tiene la particularidad de mantener la misma forma a lo largo de todos los pasos de optimización y backend. Dicha forma es además representable de tres formas: como objeto binario (que se puede obtener al compilar binarios sin enlazar con la opción -emit-llvm), como conjunto de objetos C++ (que es la representación que se manipula en el código fuente) o como ensamblador para una máquina teórica. Dicha máquina tiene un conjunto de instrucciones RISC, un ancho de datos arbitrario y un número infinito de registros, siguiendo el esquema SSA (Static Single Assignment) por el cual cada valor se define una única vez. Se puede encontrar más información sobre el ensamblador en la documentación del proyecto(17).

En este trabajo nos centramos en dos partes de todo este proceso: el frontend y el optimizador. Más concretamente, vamos a modificar parte del frontend y vamos a añadir pases de optimización al optimizador. Para aclarar esto debemos detallar la descripción del frontend.

Dentro del frontend existen cuatro submódulos que se ejecutan en serie. Siguiendo con la filosofía de LLVM, Clang separa completamente todas las partes de ese proceso. Dichos submódulos son el preprocesador, el analizador léxico o lexador, el parser y el generador de código, siendo la salida de uno la entrada del siguiente. La funcionalidad de cada submódulo se detalla como sigue:

- Preprocesador: Recibe el archivo de código fuente y expande todas las directivas de preprocesador en él, como inclusión de archivos, macros y definiciones de constantes. También realiza la separación de nombres, “name mangling”, en los lenguajes de programación que permitan sobrecarga de funciones con distintas entradas. La salida es una representación final del código tras la expansión de directivas, que puede ser completamente diferente al código original.
- Analizador léxico o Lexador: Lee el código preprocesado e identifica cualquier posible token. Un token es cualquier texto que el lenguaje interprete como importante: definición de comentario, tipo de dato, operador, nombre de variable, fin de línea, estructuración de código… El resultado es una transformación del código en un conjunto de tokens ordenados

por aparición. Este módulo se puede implementar en pipeline con el siguiente componente.

- Parser: Transforma los tokens en entidades sintácticas. Para ello agrupa los tokens que recibe para formar órdenes o construcciones del lenguaje. Por ejemplo, el parser puede agrupar los tokens 'float', 'x', '=', '6.f' y ';' en la declaración "float x = 6.f;". También añade información de localización a cada construcción o token. El resultado es el llamado AST, o Abstract Syntax Tree, en el que cada hoja representa a un token y los distintos niveles de paternidad representan distintos niveles del código – las órdenes son el primer nivel, en el siguiente nivel las regiones de código estructurado, que son anidables hasta el nivel superior que es la declaración de función y contiene todas las regiones inferiores.
- Generación de código (o CodeGen): Lee el AST y transforma la información que se puede encontrar en él en objetos de IR de tal forma que se tiene una representación correcta del código original. Por ejemplo, un bucle estaría representado en AST como un nodo ForStmt, y el módulo CodeGen tomaría el nodo ForStmt, definiría un bloque para el bucle y a continuación recorrería las órdenes internas del mismo para emitir las instrucciones (Instructions) de ese bloque, así como la condición de bucle.

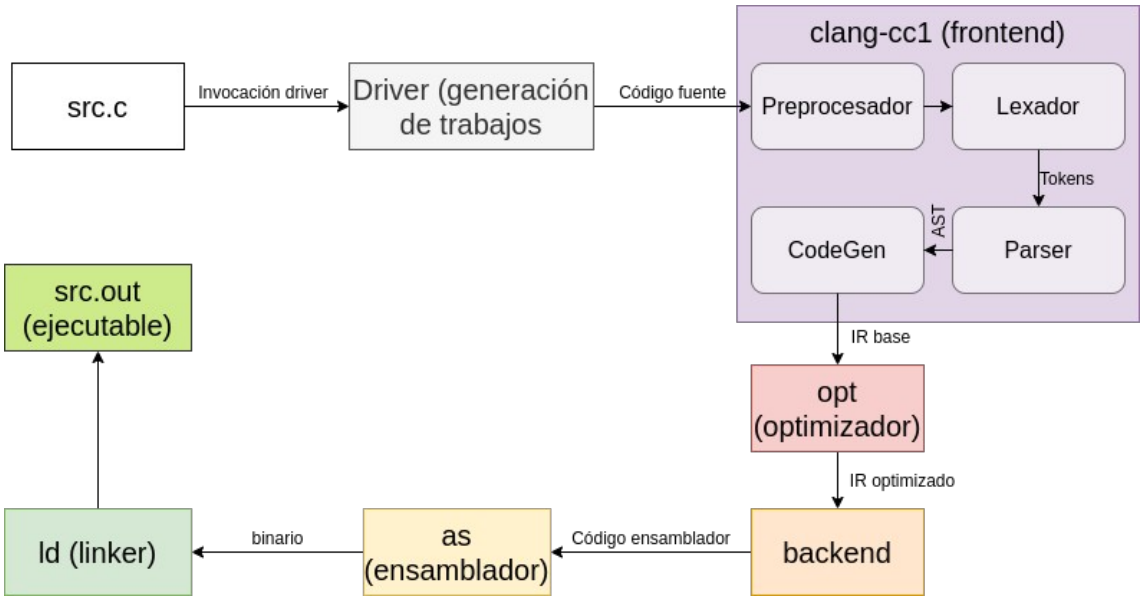

Figura 2: El proceso de compilación con el frontend extendido

Nuestro trabajo va a centrarse en la sección CodeGen del frontend. El resultado del trabajo será añadir más detalle al IR original para permitir la extracción de secciones del código fuente que se correspondan con regiones target de OpenMP. En lo relativo al

optimizador, vamos a añadir pases de optimización, que son plugins del optimizador que éste carga y ejecuta en una secuencia dada por el gestor de pases. Trabajar con pases de optimización es relativamente sencillo y extensamente documentado, tanto por parte del propio consorcio LLVM(16) como de centenares de proyectos realizados en base a pases de optimización, como el mencionado DawnCC(14) y todo tipo de proyectos de instrumentación de código.

Escala del proyecto

Algo que quizá no queda muy claro tras esta explicación de lo que es un compilador es el tamaño de un proyecto de este tipo. En concreto, el repositorio llvm-proyect(18) contiene el núcleo de LLVM, el driver y frontend Clang, la librería de OpenMP que emplea dicho driver (libomp) para la generación de código, una serie de herramientas para el núcleo y el driver, el linker lld, además de una implementación de libc y otras librerías específicas. Desde la versión 11.0.0 se incluye también el frontend y driver de Fortran, denominado Flang. Esto hace que el proyecto sobre el que se trabaja incluya más de 100.000 archivos, y algunos de estos archivos contienen miles de líneas de código. Si a ello le sumamos la escueta documentación de las implementaciones del frontend, que se reducen a los comentarios del código y lo que se pueda extraer del debugger, hacen que el proceso de entender el funcionamiento de todas las partes del conjunto requiriese meses de estudio, ensayo y error.

Por suerte este trabajo no necesita de todos los subproyectos – de todo el repositorio solo se han tocado los subproyectos de Clang, llvm y OpenMP. Para poder incluir nuestro dispositivo virtual fue necesario modificar ligeramente el backend para que existiese una región dependiente de target para nuestro dispositivo. En total se han leído, anotado y repasado cerca de 200 archivos de código, y se han limitado las ediciones a alrededor de una docena de archivos detallados en la sección de Implementación.

Estado del arte

Este trabajo se basa en gran medida en el trabajo realizado por Bertolli et al. (5) relativo a la generación de código para descarga (offloading). Basándonos en el esquema expuesto en dicho trabajo y en el esquema específico para GPUs que definieron un año después (6) construimos un sistema de generación de código que nos permite lanzar la ejecución de regiones ejecutables cargadas desde objetos externos, y de ahí extender la aproximación para cargar objetos dinámicos. Dichos trabajos(5, 6) detallan cómo se emite código en

LLVM para regiones OpenMP de manera conceptual, y en nuestro trabajo ha sido necesario adaptarlo para la nueva versión de LLVM/Clang.

Mencionamos en la introducción dos trabajos relativos a la extensión de las capacidades de descarga. Por un lado, investigadores de la Universidad Tecnológica de Darmstadt han desarrollado un sistema para descarga de código OpenMP a FPGAs(3) de manera que se puedan ejecutar regiones target en hardware. Dicho sistema consiste en una serie de herramientas con el nombre ThreadPoolComposer(7) o TPC, un plugin que enlaza un ejecutable TPC con la librería libomptarget y una API que enlaza el plugin y libomptarget con la plataforma. Empleando dicha toolchain y la herramienta de síntesis de alto nivel Xilinx Vivado(8) consiguen ejecutar código en una FPGA con solo utilizar una directiva target en el código original. Dicho sistema tiene dos carencias para alcanzar un rendimiento óptimo. La primera es que solo genera y ejecuta un kernel en la FPGA, lo que suele producir un uso ineficiente de los recursos del chip (en los ejemplos mencionan que su caso de uso tenía una ocupación aproximada del 1% del silicio disponible). La segunda es que al automatizar completamente el trabajo es imposible refinar el código para obtener un resultado óptimo en FPGA. Es cierto que el objetivo de su trabajo era la portabilidad del código, pero en este caso la pérdida de eficiencia incurrida resulta inaceptable. Es por esta razón que en nuestro trabajo buscamos permitir la ejecución de funciones cargadas dinámicamente, para así permitir refinar el código en base al acelerador a emplear, sea GPGPU, FPGA o cualquier otro. La razón por la que es necesario refinar el código es porque un algoritmo que resulta eficiente para ser ejecutado en una CPU puede no ser eficiente cuando se ejecuta en una FPGA, en donde los anchos de memoria y el número de operadores disponibles es diferente.

El otro trabajo que queremos destacar es el de Hervé Yviquel y Guido Araújo, de la Universidad de Campinas(4), en el que emplean OpenMP para descargar código al *cloud*. Su aproximación es similar al del trabajo anterior, juntando un plugin propio y una API para la comunicación y ejecución entre la plataforma y el plugin, que en este caso se trata de una API para la orquestación de recursos en cloud. Concretamente se valen del framework Apache Spark(9) y un paradigma de paralelización MapReduce(10). El plugin contiene las llamadas de Spark para autenticación en el cloud y ejecución de código en el cluster remoto. Este trabajo incluye algunas ideas interesantes, como la carga de configuraciones en runtime por parte del plugin (necesario para la autenticación y reserva de recursos del

cloud) o la extensión de la directiva target data para permitir la partición de datos. Sin embargo, este trabajo presenta el problema de tener que realizar la descarga en dos etapas, definiendo primero un área donde se va a definir el mapeo de datos y donde van a estar las secciones y después identificando el código a descargar con una directiva “parallel”. También requiere la generación automática de un módulo Scala(11) para controlar los kernels en el cloud, limitando la eficiencia por diseño. Los propios investigadores mencionan que su propósito no es el de ejecutar código buscando el mayor rendimiento, si no permitir un rápido despliegue de una ejecución altamente paralela para aplicaciones big data, como puede ser analizar un bloque de datos obtenido de un array de sensores IoT. Nuestro framework busca mayor rendimiento, de ahí que incluya un extractor de código de kernel para refinar el rendimiento al dispositivo acelerador, sea una GPU, una FPGA o incluso una instancia cloud.

Es imposible hablar de nuestro trabajo sin mencionar otro trabajo realizado en esta universidad de la mano de Ángel Álvarez et al. (12, 13), que es la base para la parte de runtime. Este trabajo consiste en una librería runtime externa que permite la ejecución de código orientado a aceleradores, y ejecutado en aceleradores de tipo GPU, FPGA o similar. En el target FOTV se adaptará dicha librería para su enlazado al módulo de dispositivo, haciendo de puente entre la librería libomptarget y los diferentes dispositivos a los que queramos descargar código. Además se encargará de la carga dinámica de kernels, así como su planificación. Una vez unida al sistema de generación de código modificado para nuestro target y al sistema de extracción de código y datos de transferencia de memoria, sería posible modificar secciones del código a acelerar o añadir nuevos dispositivos sin la necesidad de añadir más dispositivos y backends a Clang o nuevos plugins de libomptarget. Esta flexibilidad es ideal para proyectos de investigación, donde se prueban múltiples aceleradores, o en proyectos en los que se busque la optimización de recursos al poder dirigir la ejecución del kernel a distintos dispositivos según estén o no ocupados.

En cuanto a la extracción de código fuente, existen algunos proyectos de anotación de código para offloading, como DawnCC (14) o AClang (15). El primero de estos proyectos detecta bucles paralelizables en un código y añade anotaciones OpenMP con el objetivo de transferir ese código a un compilador, mientras que el segundo es una reimplementación de Clang que transforma las regiones OpenMP en kernels OpenCL permitiendo así mayor flexibilidad. Ambos trabajos son interesantes en el campo de la simplificación de la programación

paralela, y empleados en tándem pueden abrir el mundo de la programación HPC a todo tipo de programadores, pero ambos sistemas modifican el código. Nuestro objetivo es extraer la región “tal cual está” para que sea el diseñador el que realice las optimizaciones específicas para el dispositivo, y las anotaciones y transformaciones que realizan otras aproximaciones, como las anteriormente comentadas, pueden ofuscar el código original de cara a su legibilidad para el diseñador, afectando a su capacidad para la optimización y refinado del código. Aun así, la idea de emplear un pase de optimización para la extracción se empleó en este trabajo al permitir la obtención de detalles del IR que pueden fácilmente añadirse en la fase de generación de código personalizada.

En el próximo apartado abordaremos en profundidad el diseño del sistema y nuestra aproximación a la implementación, detallando las distintas soluciones empleadas en el desarrollo del sistema.

3.-Diseño e Implementación

Esquema de offloading

El esquema de offloading de OpenMP en LLVM/Clang se define en los trabajos de Bertolli (5, 6, 19), que voy a tratar de explicar a grandes rasgos en esta sección. La idea inicial es que el proceso de descarga se ha de realizar en tiempo de compilación, y eso requiere extender el trabajo de compilación más allá de los 5 pasos que se mencionan en Conceptos previos de compiladores. En concreto necesitamos que se cumplan los siguientes puntos:

- El resultado de la compilación, entendida como el proceso completo de ejecutar los 5 pasos definidos en la sección anterior, ha de ser un Fat Binary.
- El código de acelerador ha de estar físicamente separado del código de host.
- El acelerador no tiene por qué compartir backend, ensamblador o linker con el host.

La solución propuesta en OpenMP fue que el proceso de compilador crease un nuevo trabajo completo de compilación para cada dispositivo con anotaciones obtenidas en la fase de CodeGen para extraer los kernels de manera automática. Por tanto el proceso se desdobla antes del optimizador, iniciando un nuevo driver con el código fuente como entrada y el IR como entrada auxiliar de anotación.

El driver por tanto construye un nuevo trabajo de compilación empezando por el frontend, que puede ser diferente para el acelerador si requiere de generación de código específica, y siguiendo con el optimizador, el backend, el ensamblador y el linker, siendo estas tres últimas partes también específicas del dispositivo. Clang abstrae el conjunto de herramientas en la clase “Toolchain”, y en la construcción del trabajo por parte del driver obtiene la “Toolchain” específica de cada dispositivo.

Una vez tiene el ejecutable del dispositivo es necesario juntar la parte del acelerador con la parte del host. Para ello se desarrolló la herramienta “clang-offload-wrapper”, cuyo propósito es tomar un ejecutable de un dispositivo acelerador y aplicar un envoltorio alrededor que lo presente a OpenMP como una imagen binaria que pueda cargar al arrancar el programa (con la “apariencia” específica de un símbolo “.offload-entries”). La salida del wrapper es un IR en

formato binario, que se pasa por backend y ensamblador del host para finalmente enlazarse con el dispositivo host y construir el Fat Binary.

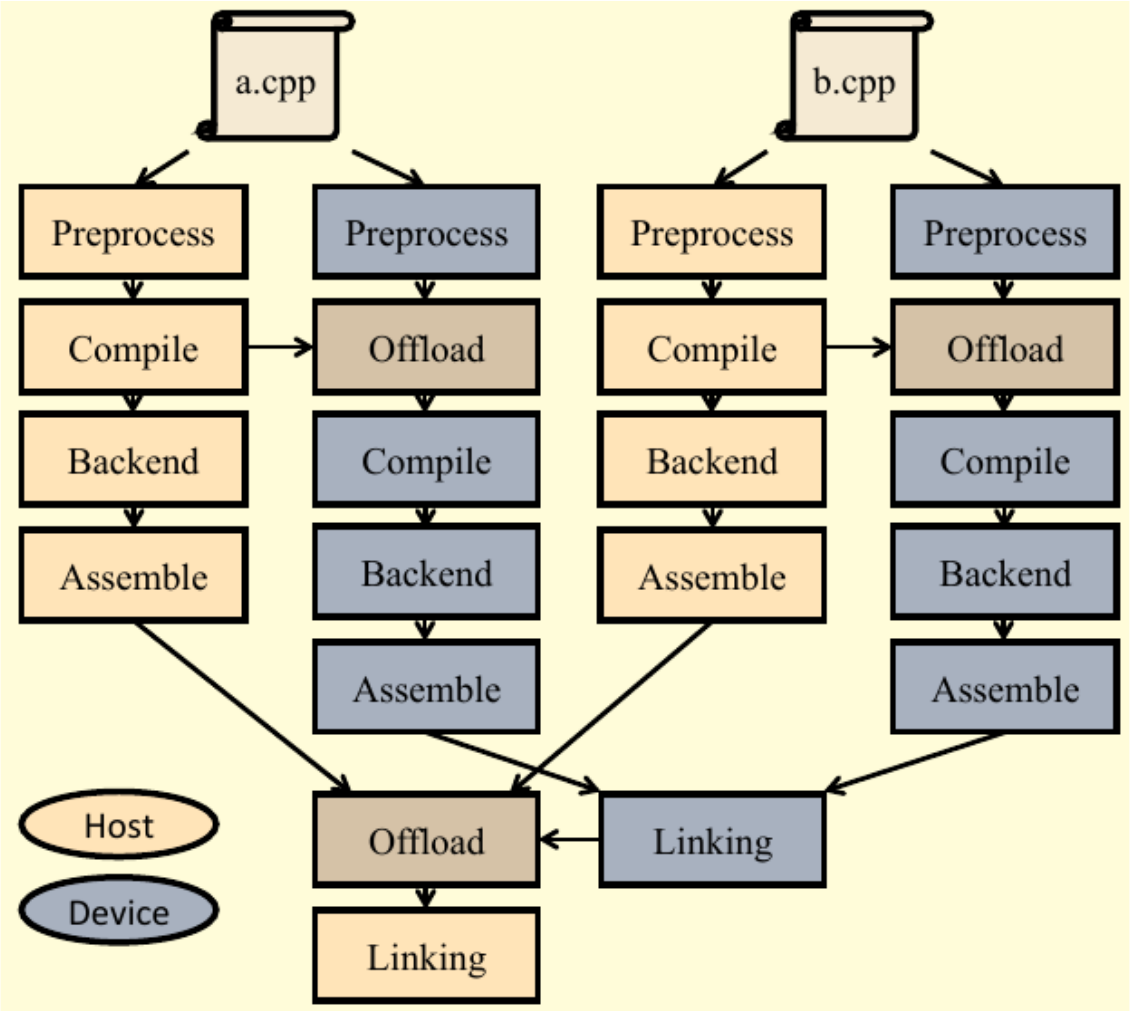

Figura 3: Imagen extraída de la refeerencia 6. Aquí vemos el proceso de doble compilación descrito, donde los nodos "offload" son los puntos donde se realiza la separación y unión de las partes en el programa. En concreto, en la parte superior de la imagen vemos donde se desdobla el trabajo y en la parte inferior vemos donde se realiza el wrapping.

A la hora de ejecutarse, la librería OpenMP se encarga de cargar el módulo de la librería que controla el offloading de manera general, que a su vez carga dinámicamente los plugins que estén disponibles en el sistema. Los plugins se asocian con el símbolo .offload-entries correspondiente (identificado mediante un ID de dispositivo). Para poder ser manejado por el plugin, éste copia el contenido del símbolo a un archivo temporal, que después se carga como si fuese una librería dinámica. Cada plugin contiene una serie de funciones que maneja las tareas de movimiento de datos, ejecución para uno o varios threads (o warps en el caso de GPGPU) y, opcionalmente, utilidades para offloading no bloqueante.

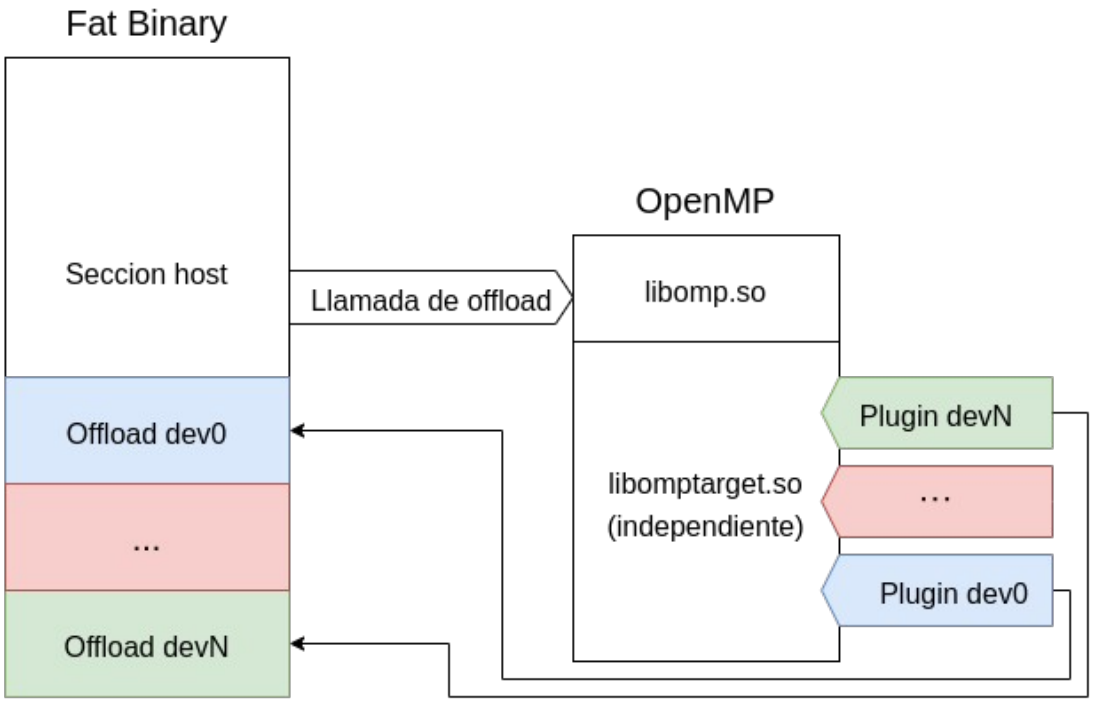

Figura 4: Esquema básico de ejecución con offloading con OpenMP. El host ejecuta el código base hasta que llega a la sección de offloading, donde cede el control a la parte independiente de dispositivo de libomp/libomptarget. Éstas cargan de forma dinámica el plugin de ejecución en el dispositivo, que carga la sección correspondiente del fat binary al dispositivo acelerador.

Objetivo

Nuestro objetivo con FOTV es el de conseguir, empleando el esquema de offloading de OpenMP, añadir un runtime que nos permita la ejecución de código descargado a nuevos dispositivos sin necesitar construir nuevos plugins, añadir los dispositivos como backends del compilador o modificar la librería de offloading para cada dispositivo. Por lo tanto, lo que buscamos es posibilitar la descarga de código a dispositivos sin incluir soporte específico para el mismo dentro de LLVM/Clang.

Para ello partimos del esquema original de offloading de OpenMP (Figura 4) y lo extenderemos con la ayuda de modificaciones a la generación de código, como vemos en la sección de diseño.

Diseño

Nuestro desarrollo necesitará de un plugin para la librería libomptarget. Pero dicho plugin ha de estar asociado a un dispositivo y es dependiente de él. Para ello, definiremos un target en LLVM al que asociaremos un backend y lo añadiremos a la lista de plugins genéricos de 64 bits. Por comodidad en nuestro entorno de trabajo,

se optó por asociar el nuevo plugin al backend de x86_64, obteniendo archivos elf64 compatibles con el host. Tal como se mencionó en el apartado anterior, el objetivo final es que este plugin ejecute un runtime propio que va a correr en la CPU host, por lo que un plugin que se descarga a sí mismo es la mejor opción.

```
extern "C" void LLVMInitializeX86TargetInfo() {
  RegisterTarget<Triple::x86, /*HasJIT=*/true> X(
      getTheX86_32Target(), "x86",
      "32-bit X86: Pentium-Pro and above", "X86");

  RegisterTarget<Triple::x86_64, /*HasJIT=*/true> Y(
      getTheX86_64Target(), "x86-64",
      "64-bit X86: EM64T and AMD64", "X86");

  RegisterTarget<Triple::fotv, /*HasJIT=*/true> Z(
      getTheFOTVTarget(), "fotv",
      "64-bit X86 masked as FOTV", "X86");
}
```

Fragmento 1: Extracto del registro de targets x86. Nótese la adición de nuestro target, heredando backend y otros módulos de x86. Hemos añadido este tipo de registro en otras partes del backend x86

```
class Triple {
public:
  enum ArchType {
    UnknownArch,

    arm,        // ARM (little endian): arm, armv.*, xscale
    armeb,      // ARM (big endian): armeb
    …
    ve,         // NEC SX-Aurora Vector Engine
    fotv,       // Future Offload Target Virtualization target
    LastArchType = fotv
  };
…
}
```

Fragmento 2: Extracto de la inclusión de nuestro target en el registro de tripletas de LLVM. Este paso es esencial para la detección por parte del compilador, siendo el registro central de arquitecturas soportadas.

Por otro lado, tenemos que definir la nueva región de offloading en el Fat Binary para cada región target. Esta sección se genera de forma automática en el proceso de compilación, construyendo nuevos módulos para cada uno de los dispositivos aceleradores a emplear.

Analizando el proceso definido en LLVM/Clang para la generación del Fat Binary (Figura 5), vemos que el mismo tiene 7 pasos. En ①, el compilador obtiene un IR que servirá para delimitar las secciones que

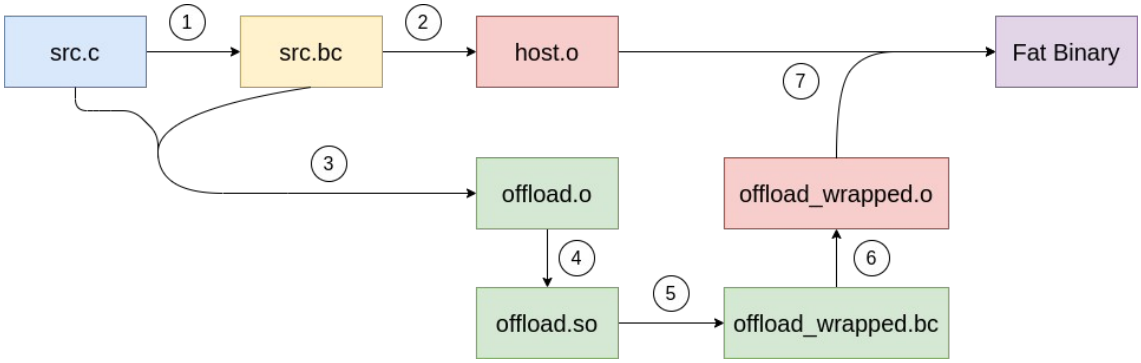

Figura 5: Secuencia de acciones en el proceso de generación de Fat Binary estándar.

tiene que descargar y el nombre que ha de dar a esos símbolos para su ejecución. En ② genera el objeto host en base a ese IR, dado que representa al programa. En ③ transfiere el código y su IR a un backend soportado, sea uno interno de clang o uno al que clang pueda llamar (como por ejemplo el ensamblador ptxas de CUDA para dispositivos nVidia) y construye un objeto offload. Para que dicho objeto se pueda cargar por el plugin ha de ser una librería dinámica, por lo que en ④ lo que hace es transformar el objeto en un shared object, recompilando el objeto con -shared para darle las características de librería dinámica. En ⑤ se toma dicha librería dinámica y se pasa por el offload-wrapper de clang, que resulta en un IR que representa la librería dinámica y los puntos de enlace al objeto host. El paso ⑥ es básicamente el mismo paso que ② – coger un IR y compilarlo como objeto binario. Por último, en ⑦ se enlazan ambos objetos y se obtiene el Fat Binary.

Según el dispositivo acelerador, es probable que en el IR generado en ① encontremos ligeras diferencias en la definición de la función según las necesidades del dispositivo y de su backend particular. Un buen ejemplo de esta capacidad está en el generador de kernels para nVidia (definido en CGOpenMPRuntimeNVPTX.cpp), en el que hay dos caminos de generación diferentes dependiendo de si el kernel emplea el paradigma SPMD o no (ver Fragmento 3).

Volviendo a la Figura 5, podemos ver que en el paso 4 el objeto de offloading se recompila para construir un objeto dinámico. Es en este punto donde se pueden introducir otras piezas que puedan ser utilizadas por el dispositivo, como un runtime u objetos adicionales.

En la Figura 6 podemos ver una posible modificación haciendo uso de esta posibilidad. En la misma hay dos cambios principales. Por un lado, se incluye “rtl_dev.o” en el objeto target. Esto es posible porque al generar el código del objeto target original hemos introducido llamadas a símbolos que se encontrarían en rtl_dev.o, y por tanto lo introduciremos en el proceso de construcción del objeto dinámico del

dispositivo. Por otro, enlazamos rtl_host.o en el Fat Binary final. Es necesario este paso para controlar el runtime desde el host, dado que el runtime va a tener su propio entorno de control específico de los dispositivos que está manejando. Dichas llamadas no serían

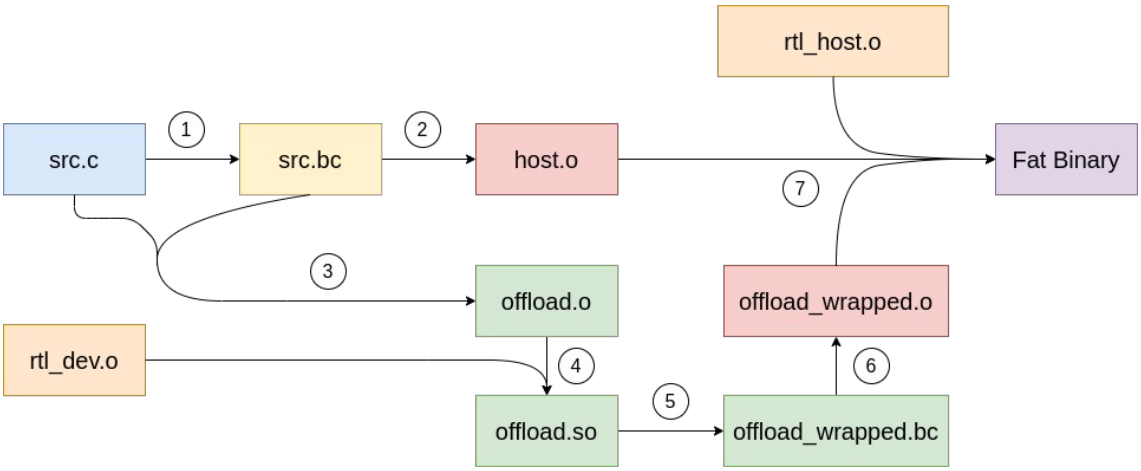

Figura 6: Esquema modificado, con la introducción de nuestro runtime

```
void CGOpenMPRuntimeNVPTX::emitTargetOutlinedFunction(
    const OMPExecutableDirective &D, StringRef ParentName,
    llvm::Function *&OutlinedFn, llvm::Constant *&OutlinedFnID,
    bool IsOffloadEntry, const RegionCodeGenTy &CodeGen) {
  if (!IsOffloadEntry) // Nothing to do.
    return;

  assert(!ParentName.empty() &&
    "Invalid target region parent name!");

  bool Mode = supportsSPMDExecutionMode(CGM.getContext(), D);
  if (Mode){
    emitSPMDKernel(D, ParentName, OutlinedFn,
        OutlinedFnID, IsOffloadEntry, CodeGen);
  }else{
    emitNonSPMDKernel(D, ParentName, OutlinedFn,
        OutlinedFnID, IsOffloadEntry, CodeGen);
  }
  setPropertyExecutionMode(CGM, OutlinedFn->getName(), Mode);
}
```

Fragmento 3: Función de entrada al generador de funciones de nVidia. Nótese que es un envoltorio para dos funciones según modo de programación

accesibles si todo el runtime se introdujese en el objeto offload, porque “clang-offload-wrapper” lo encapsula en una caja negra. Un esquema similar es empleado por nVidia, que se vale de un módulo CUDA incluido dentro de la librería libomp para la planificación de kernels en los warps, que en el paso ④ compila y enlaza dentro de su objeto de dispositivo, enlazando el resto de la librería con el código de host (ver Fragmento 4).

```
.master:                                          ; preds = %.mastercheck
  %nvptx_warp_size4 = call i32
    @llvm.nvvm.read.ptx.sreg.warpsize(), !range !14
  %nvptx_num_threads5 = call i32
    @llvm.nvvm.read.ptx.sreg.ntid.x(), !range !13
  %thread_limit6 = sub nuw i32 %nvptx_num_threads5,
    %nvptx_warp_size4
  call void @__kmpc_kernel_init(i32 %thread_limit6, i16 1)
  call void @__kmpc_data_sharing_init_stack()
  %7 = call i32 @__kmpc_global_thread_num(%struct.ident_t* @1)
  %8 = load float, float* %conv, align 8
  %conv7 = bitcast i64* %a.casted to float*
  store float %8, float* %conv7, align 4
  %9 = load i64, i64* %a.casted, align 8
  store i32 %7, i32* %.threadid_temp., align 4
  call void @__omp_outlined__1(i32* %.threadid_temp., i32*
%.zero.addr, [100 x float]* %0, i64 %9, [100 x float]* %1) #2
  br label %.termination.notifier
```

Fragmento 4: Ejemplo de IR generado para un kernel nVidia. Nótense las dos llamadas resaltadas: son funciones definidas dentro del código de libomp

Para poder incluir las llamadas al nuevo runtime necesitaremos modificar la generación de código para el dispositivo. Hemos mencionado antes el esquema de nVidia como ejemplo de la posibilidad de realizar generación de código personalizada dentro del frontend, y hacemos uso de esa capacidad para modificar el código para nuestro dispositivo. Nuestro objetivo es, de forma similar a lo mencionado en el paso ④ de nVidia, realizar una indirección dentro del código del acelerador de tal forma que transfiera la ejecución del kernel a nuestro runtime, junto con los datos necesarios para su ejecución. Para ello necesitaremos una clase para la generación de llamadas OpenMP propia, que llamaremos CGOpenMPRuntimeFOTV, y que solo se encargará de las directivas target y target data.

Además del sistema de offloading a dispositivos genéricos, también queremos tener un sistema de extracción de código automatizada que nos permita obtener el código fuente de los kernels durante el proceso de compilación, para poder así refinarlos en función del dispositivo que queremos utilizar para su ejecución.

Para ello introduciremos en la generación de código una serie de anotaciones dentro del módulo IR, que serán interpretadas por pases de optimización para generar archivos con código e información. El flujo de compilación quedaría, de manera simplificada, como sigue:

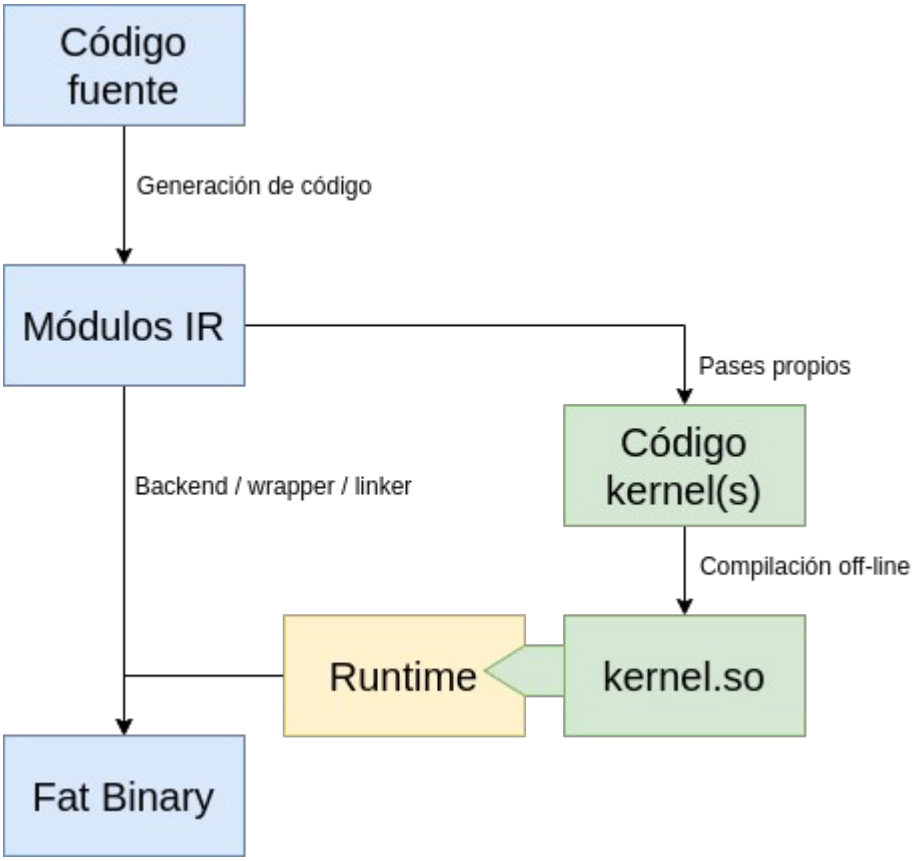

Figura 7: El flujo de compilación. En la parte izquierda tenemos el resultado habitual, mientras que por la derecha tenemos un kernel generado a partir de los códigos extraídos, que se carga de forma dinámica en el runtime ya enlazado en el proceso estándar

De esta manera podemos alcanzar los objetivos que nos proponíamos en la sección anterior. Una vez aclarado el diseño que vamos a seguir, podemos plantearnos la implementación.

Implementación

La implementación se hace como un conjunto de clases y funciones en el proyecto LLVM/Clang, además de dos pases de optimización, centrándonos principalmente en las secciones que nos permiten incorporar nuevos dispositivos y generación de código especial para ellos. Estos añadidos se hacen sobre la versión 11.0.0rc2, subrayados los archivos nuevos:

- clang/lib/Basic/Targets/FOTV.cpp: Implementación del objeto TargetInfo que Clang emplea para conocer la información de un dispositivo. Es en gran medida una implementación dummy, dado que nuestro target para el compilador es el host y no tiene funciones específicas.
- clang/lib/Basic/Targets/FOTV.h: La definición de la clase, conteniendo la información que le puede resultar útil al

generador de código y el backend como espacios de direcciones, sistemas de direccionamiento, tamaño de los punteros, propiedades y otras particularidades. Es necesaria para que exista como target distinto y está basado en una de las implementaciones de X86_64.

- clang/lib/CodeGen/CGOpenMPRuntimeFOTV.cpp:
  Implementación de la generación de código para el target. Contiene la generación modificada de código y una serie de funciones de ayuda para simplificar y modularizar el proceso.
- clang/lib/CodeGen/CGOpenMPRuntimeFOTV.h: Definición de la clase de generación de código para el target. Se define como herencia de CGOpenMPRuntime, e incluye algunos overrides y funciones privadas para ayuda.
- clang/lib/CodeGen/CGOpenMPRuntime.h: Se le da acceso a CGOpenMPRuntimeFOTV a algunas de las funciones de ayuda dentro del generador de propósito general.
- clang/lib/CodeGen/CodeGenModule.cpp: En esta clase se construye el módulo de generación de código, y por ello hay que modificarla para que cree el objeto de generación pertinente para nuestro target.
- clang/lib/Driver/Driver.cpp: Archivo que genera la invocación del driver, y construye la toolchain. Es necesario incluir aquí la construcción de la toolchain del dispositivo para poder realizar una compilación.
- clang/lib/Driver/Toolchains/FOTV.cpp: Implementación de la definición del toolchain, entendido como cadena de comandos. Contiene la creación de la toolchain, a qué invocar para cada paso (compilación, ensamblaje, enlazado) y si se necesitan opciones particulares de compilación por defecto.
- clang/lib/Driver/Toolchains/FOTV.h: La definición del toolchain. Contiene definiciones para la creación de los diversos comandos y opciones por defecto.
- clang/lib/Frontend/CompilerInvocation.cpp: De no definirse explícitamente el dispositivo como opción, se lanza un error al tratar de emplear el dispositivo virtual como target de OpenMP, por lo que se añade la opción.
- llvm/include/llvm/ADT/Triple.h: El archivo central de tripletas target, identificadores únicos que el compilador utiliza como

base para construir el trabajo de compilación. Se incluyó el dispositivo virtual en este archivo.

- llvm/lib/Support/Triple.cpp: Implementación de funciones definidas en el archivo mencionado sobre estas líneas.
- llvm/lib/Target/X86: Dado que el target virtual se implementa como un target hacia el propio host que va a ejecutar un runtime, se añade el target virtual a las definiciones de backend para x86_64.
- openmp/libomptarget/CmakeLists.txt: Instrucciones para la construcción de plugins de libomptarget, es necesario incluir la orden para construir nuestro plugin genérico.
- openmp/libomptarget/plugins/fotv: La carpeta contiene instrucciones Cmake para construir un plugin genérico x86_64 de nombre “fotv” para que haga las veces de dispositivo genérico para OpenMP.
- openmp/libomptarget/src/rtl.cpp: Se tiene que añadir el plugin genérico creado para el target virtual dentro de la lista de plugins que va a buscar y cargar dinámicamente al arrancar.

A esto se le añaden dos pases de optimización nuevos, construidos fuera del árbol LLVM, con los nombres de “Transform” y “Metadata” que actúan a distintos niveles:

- UCTransform: Es el paso de extracción de código. Actúa a nivel de función, tomando de las funciones generadas por el compilador como encapsulamiento para las regiones target de OpenMP una serie de metadatos incluidos en la generación de código para localizar y extraer el código de dicha región a una función en un archivo aparte. También genera un wrapper para manejar los cambios de variable realizados por el generador de código OpenMP para libomptarget y una serie de información sobre mapeos y variables presentados en el archivo generado como comentarios de código antes de la función extraída.
- UCMetadata: Es el paso de extracción de información de regiones target data. Actúa a nivel de módulo, extrayendo la información generada de cada una de esas regiones como objeto .JSON y encapsulando todo en un archivo por módulo. La información extraída incluye dirección del mapeo, variables afectadas, líneas de código abarcadas y tipo de región target data (si es una región o una directiva independiente, y en este caso que clase de directiva independiente es).

A los cambios del compilador y los pases añadidos se le une el runtime, que se ha de enlazar con el Fat Binary y contiene las siguientes partes:

- Módulo de gestión (DynLibManagement.c): Interfaz para el programador que se encarga del manejo manual de librerías dinámicas – carga y descarga de las mismas así como la localización en el sistema de nuevas librerías.
- Módulo de control (Lib.c): El control central del proceso. Contiene watchers para variables del entorno, como localización por defecto de librerías o número de módulos cargados, así como su estado. También incluye funciones de control de los dispositivos y su arranque, así como funciones para reportar esta información. Este módulo también define funciones de inicio, comprobación y desconexión de dispositivo por defecto.
- Tipos de datos (types.h): Una serie de estructuras de datos con las que se controlan los dispositivos e implementaciones disponibles, así como datos auxiliares para la planificación de las ejecuciones. En concreto, mantiene un registro de las funciones descargadas, las implementaciones disponibles para dichas funciones, los dispositivos manejados y si existen múltiples variantes de la misma implementación.

Con estas tres partes unidas, se puede desarrollar un código y anotarlo con directivas OpenMP y disponer de la capacidad de refinar y ejecutar el kernel más adelante de manera automática. El compilador/extractor se encarga de generar un código ejecutable básico y de generar información para la transferencia de datos al kernel, así como de extraer el código del kernel junto con una interfaz que el runtime pueda entender sin problemas. El compilador se encarga de incluir el runtime para realizar la carga y ejecución de kernels de manera dinámica, por lo que el diseñador solo tiene que encargarse de compilar el kernel para su dispositivo deseado y refinarlo si así desea.

En el próximo apartado hablaremos del estado del desarrollo, que actualmente se encuentra en fase de prueba de concepto. Entraremos en profundidad en las implementaciones actuales y las soluciones a las que se ha llegado, las funcionalidades que todavía faltan por implementar y la usabilidad del sistema actual.

4.-Desarrollo actual y prueba de concepto

En el momento de presentar este trabajo se ha finalizado la generación de código personalizada para el dispositivo virtual y el extractor de código. La librería requiere de un proceso de adaptación desde el trabajo de Ángel et al. (9) que debido a diversas complicaciones en el proyecto no ha sido posible iniciar todavía. Aun así, se dispone de un sistema funcional, aunque rígido.

Generador de código

El sistema empieza por la generación de código personalizada. En concreto, nuestro objetivo es poder ejecutar funciones externas dentro de la región target. Usando el esquema de generación de kernels para nVidia, nuestro generador también incluye una sección en la que llamar a una función externa.

```
entry:
…
  %0 = load i64*, i64** %num_steps.addr, align 8
  %conv1 = bitcast i64* %x.addr to double*
  %1 = load double*, double** %step.addr, align 8
  %2 = load double*, double** %sum.addr, align 8
  %3 = load i32, i32* %conv, align 8
  %conv2 = bitcast i64* %i.casted to i32*
  store i32 %3, i32* %conv2, align 4
  %4 = load i64, i64* %i.casted, align 8
  %5 = load double, double* %conv1, align 8
  %conv3 = bitcast i64* %x.casted to double*
  store double %5, double* %conv3, align 8
  %6 = load i64, i64* %x.casted, align 8
  %7 = call i32 @__omp_offloading_812_c20010_main_l65_indirect
       (i64 %4, i64* %0, i64 %6, double* %1, double* %2)
  %off_success = icmp eq i32 %7, 0
  br i1 %off_success, label %.exit, label %.run
```

Fragmento 5: Extracto de IR objetivo. Lo que se quiere conseguir es generar la llamada a __omp..., recoger su resultado (%7) y emplearlo como condición de un branch entre el código de host (.run) y el final de la función (.exit). Se omite el denominado prólogo de la función, que consiste en la copia de variables desde la llamada

Para modificar la generación de funciones, Clang emplea un esquema de pre/post acciones de compilado. Dado que el generador construye un IR agnóstico al dispositivo, el cuerpo de la función se genera siempre siguiendo el mismo esquema, permitiendo al programador del compilador introducir las modificaciones que considere oportunas

en forma de acciones previas o posteriores a la generación del cuerpo de la función.

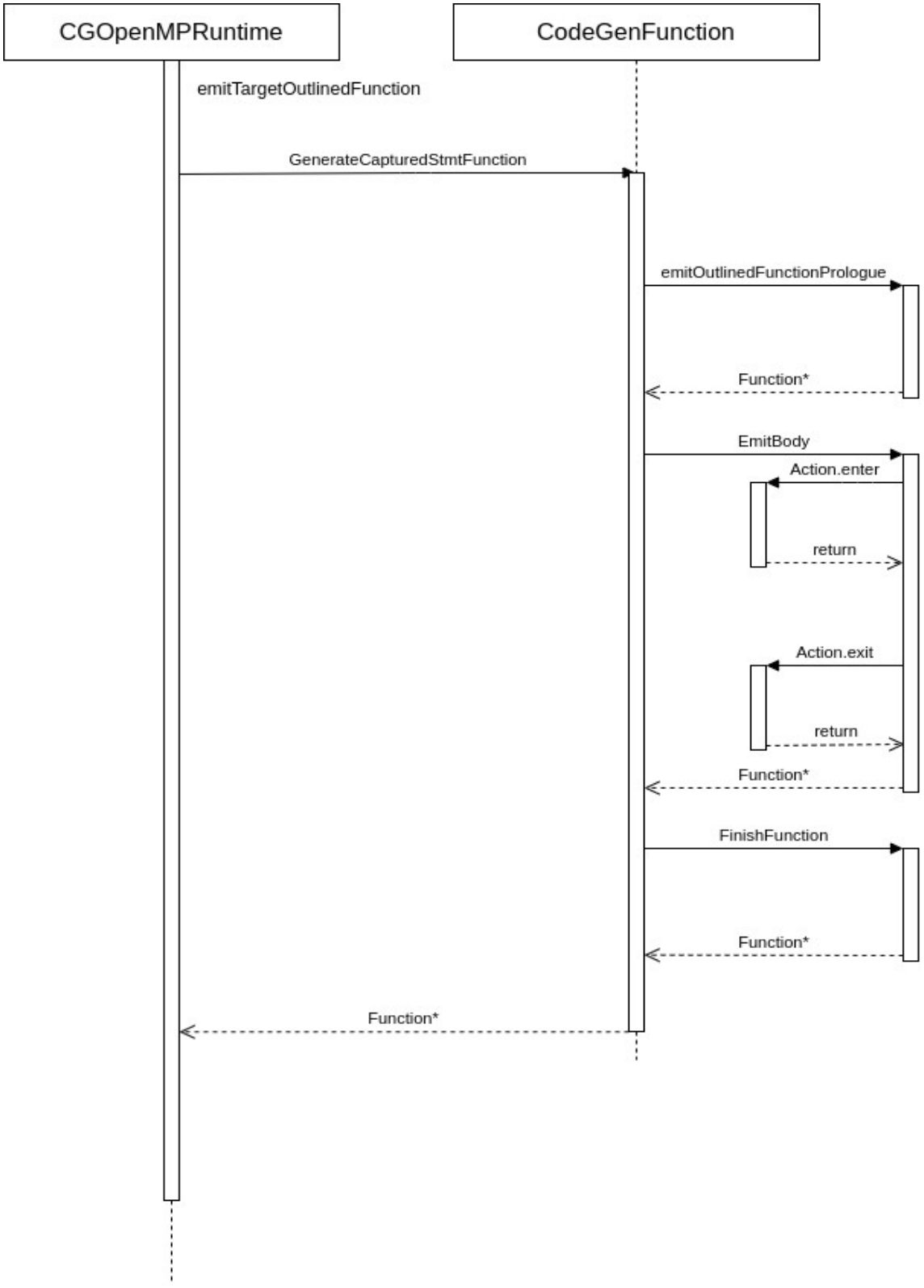

Figura 8: Esquema de llamadas del sistema de pre-post acciones. Action.enter es la pre-acción, Action.exit es la post-acción

En nuestro desarrollo, la acción previa extrae las variables de la función para llamar a una función externa, y si la función retorna con éxito se salta al final de la ejecución. Si no, ejecuta el código normalmente, como se ve en el Fragmento 5. Optamos por esta estrategia de branching por la simple razón de que nuestro dispositivo es el host, por lo que, si la llamada a la función falla, ejecuta el código en el host. Esta estrategia es una forma de mantener el funcionamiento estándar de OpenMP con un dispositivo que, por sus características, siempre va a retornar `OFFLOAD_SUCCESS` cuando se llame a la función target de OpenMP. La post-acción no es más que la llamada de retorno de la función, pero es necesaria para poder realizar el branching dado que se tiene que emitir el bloque final.

```
CGOpenMPRuntimeFOTV::emitTargetOutlinedFunction([args]) {
  assert(!ParentName.empty()
      && "Invalid target region parent name!");
  EntryFunctionState EST;

  // Emit target region as a standalone region.
  class FOTVPrePostActionTy : public PrePostActionTy {
    CGOpenMPRuntimeFOTV::EntryFunctionState &EST;
    CGOpenMPRuntimeFOTV &RT;
    const OMPExecutableDirective &D;
    StringRef ParentName;

  public:
    FOTVPrePostActionTy(
      CGOpenMPRuntimeFOTV::EntryFunctionState &EST,
      CGOpenMPRuntimeFOTV &RT,
      const OMPExecutableDirective &D, StringRef ParentName)
      : EST(EST), RT(RT), D(D), ParentName(ParentName) {}

    void Enter(CodeGenFunction &CGF) override {
      RT.emitTargetEntryHeader(CGF, EST, D, ParentName);
      // Skip target region initialization.
      RT.setLocThreadIdInsertPt(CGF, /*AtCurrentPoint=*/true);
    }
    void Exit(CodeGenFunction &CGF) override {
      RT.clearLocThreadIdInsertPt(CGF);
      RT.emitTargetEntryFooter(CGF, EST);
    }
  } Action(EST, *this, D, ParentName);

  CodeGen.setAction(Action);
  …
}
```

Fragmento 6: Clase y objeto pre/post-acción de nuestro generador, junto con su asociación al generador. Es práctica generalizada la de definir y crear la clase de pre/post-acción en la propia función de generación.

```
CGOpenMPRuntimeFOTV::emitTargetEntryHeader([args]){

  CGBuilderTy &Bld = CGF.Builder;
  const CapturedStmt *CS = D.getCapturedStmt(OMPD_target);

  llvm::BasicBlock *runnerBB = CGF.createBasicBlock(".run");
  EST.ExitBB = CGF.createBasicBlock(".exit");

  //Success conditions
  llvm::Value *status, *cond;
  //function arguments
  llvm::SmallVector<llvm::Value*, 16> fn_args;
  CGF.GenerateOpenMPCapturedVars(*CS, fn_args);

  //function name
  …
  SmallString<64> EntryFnName;
  …
  //function type
  llvm::SmallVector<llvm::Type*, 16> arg_types;
  //from fn_args

  llvm::FunctionType *fnty=
    llvm::FunctionType::get(CGF.Int32Ty, arg_types, false);
  llvm::FunctionCallee callee =
      CGM.CreateRuntimeFunction(fnty, EntryFnName);

  //Return status
  status = CGF.EmitRuntimeCall(callee, fn_args);
  cond = Bld.CreateICmpEQ(status, Bld.getInt32(0),
      "off_success");

  Bld.CreateCondBr(cond, EST.ExitBB, runnerBB);

  CGF.EmitBlock(runnerBB);
```

Fragmento 7: Código de generación simplificado. Las líneas resaltadas son responsables del IR resaltado en Fragmento 5

La clave está en la región entre las líneas 23 y 33 del Fragmento 7. En ese espacio se define una función externa (callee), se crea una instrucción call a dicha función externa (EmitRuntimeCall), se extrae su resultado (status) y se hace un branch en función de dicho resultado (cond / CreateCondBr). En la prueba de concepto actual, dicha función es un wrapper directo que recibe los mismos parámetros que la función definida por el compilador para la región target, con un nombre generado como derivación del nombre de la función original (<funcion_de_offloading>_indirect).

Esto conforma una parte del generador de código, la encargada de generar código que permite la ejecución de funciones del runtime. La

otra parte es la que trabaja en conjunción con los dos pases de extracción. Para ello es necesario extraer una serie de datos relativos a la función dentro del código original.

El camino que se decidió tomar fue incluir dentro del IR información relativa a la función obtenida del AST, junto con algunos metadatos conocidos en el propio IR como tipos, argumentos o el nombre dado a la función. Esto se hace mediante una función que añade la información a la función en forma de *Named Attributes*, con nombres y valores dados en strings. Dado que el objetivo de esta parte de la generación es construir un archivo de texto, que los datos se reduzcan a strings no es un problema.

```
//Get the parameters
int paramNum = OutlinedFn->arg_size();
OutlinedFn -> addFnAttr("NumArgs", std::to_string(paramNum));
int i = 0;
const RecordDecl *RD = CS.getCapturedRecordDecl();
auto CurCap = CS.captures().begin();
for(auto f=RD->field_begin(), fe=RD->field_end(); f!=fe;
    ++f, ++CurCap){
  QualType qt = f->getType();
  bool isPtr=true;
  if(CurCap->capturesVariableByCopy()){
    if(!f->getType()->isAnyPointerType()){
      isPtr=false;
    }
  }
  llvm::Argument *arg = OutlinedFn -> getArg(i);
  OutlinedFn -> addFnAttr("FieldType"+std::to_string(i),
      qt.getAsString());
  OutlinedFn -> addFnAttr("FieldName"+std::to_string(i),
      arg->getName());
  OutlinedFn -> addFnAttr("isPointer"+std::to_string(i),
      isPtr?"1":"0");
  ++i;
}
```

Fragmento 8: Extracto del código para captura de atributos. En este fragmento se obtienen, en orden, el número de argumentos de la función, el tipo de cada uno, el nombre y si se captura por referencia o por copia

Uno de los detalles de la generación de *Named Attributes* es la inclusión de información sobre bucles anidados. Esta información se obtiene mediante una función recursiva que obtiene los niveles de anidación y sus correspondientes variables de control iniciales y finales. Esta información puede ser importante para la generación de código orientada a dispositivos que utilicen paradigmas de programación orientados a paralelismo de datos, como OpenCL.

```
OutlinedFn -> addFnAttr("NestedDirectiveKind", kind);
int nesting = 0;
if(kind=="for"||kind=="simd"){
  //It's a compound directive
  const CapturedStmt *for_parent =
      D.getInnermostCapturedStmt();
  const ForStmt *f_stmt =
      dyn_cast<ForStmt>(for_parent->getCapturedStmt());
  annotateTargetOutlinedFnLoops(OutlinedFn, f_stmt, &nesting,
      PrintingPolicy(CGM.getLangOpts()));
  OutlinedFn -> addFnAttr("LoopNestingLevel",
      std::to_string(nesting+1));
}
```

*Fragmento 9: Extracto de la sección de determinación de bucles anidados. La función "*annotate...Loops*" realiza ese trabajo de forma recursiva, recorriendo la cadena ForStmt → CompoundStmt hasta el cuerpo de éste no es solo un ForStmt*

Para extraer los datos de las regiones target data y las directivas declaratorias target enter/exit data y target update tenemos que volver al generador del host[3], y es el único fragmento de código que técnicamente modifica comportamientos anteriores del código en lugar de añadir nuevos comportamientos. Esto es necesario porque en el generador del target no se tratan dichas regiones, el módulo asume que recibe las regiones de datos pertinentes y las funciones relativas a la transferencia de datos no pueden incluirse dentro de un kernel por limitaciones del esquema de offloading de OpenMP. No obstante, se ha limitado dicha modificación a la generación de un símbolo antes de empezar a generar el entorno de datos definido por la región target.

Para ello seguimos un esquema similar al anterior, pero a nivel de módulo. Por cada directiva de transferencia de datos, sea una región target data o una declaración suelta, se extraen metadatos del AST que se transfieren a un símbolo de tipo *string literal*, que contiene una definición en formato *json* de dicha región. El objeto *json* contiene los siguientes datos:

- Nombre para la directiva
- Línea inicial
- Línea final, si es una región
- Lista de mapeos:
  - Dirección
  - Tipos
  - Nombres

3 El código está duplicado en la clase generada de cara a la presentación

```
"OMP_METADATA_TARGET_DATA_REGION_l9":{
     "startLine":9,
     "endLine":21,
     "mappings":[{
          "mapType":"tofrom",
          "varTypes":["float", "float [100]", "float [100]"],
          "varNames":["a", "x", "y"]
          }
     ]
}
```

Fragmento 10: Ejemplo de json obtenido

En este caso no se hace uso del IR, obteniendo los datos directamente de la declaración en el AST. Esto se debe a que, a diferencia de las directivas target y derivadas, éstas no son directivas ejecutables, por lo que en el proceso de generación de código su información queda ofuscada al transformarse en llamadas a funciones de transferencia de datos y representaciones en forma de estructuras genéricas[4].

```
@.offload_sizes = private unnamed_addr constant [3 x i64]
    [i64 4, i64 400, i64 400]
@.offload_maptypes = private unnamed_addr constant [3 x i64]
    [i64 35, i64 35, i64 35]
…
@.offload_sizes.1 = private unnamed_addr constant [3 x i64]
    [i64 400, i64 4, i64 400]
@.offload_maptypes.2 = private unnamed_addr constant [3 x i64]
    [i64 547, i64 800, i64 547]
```

Fragmento 11: La misma información de mapeo, con las mismas variables y las mismas direcciones, según se representa en IR. Nótese que los nombres de las variables se pierden, y el tipo "tofrom" se desdobla en el movimiento de ida y el de vuelta.

El código con el que se construyen los símbolos se introduce en las funciones emitTargetDataCalls (para regiones target data) y emitTargetDataStandaloneCall (para target enter data, target exit data y target update), tomando la directiva que se va a utilizar para generar el call y emitiendo su información a una constante global de tipo c-string (array de caracteres terminado con un nulo).

4 Se entiende como estructura genérica a aquella definida por un puntero, un tamaño y una serie de offsets

```
SmallString<64> sym_name;
{
  llvm::raw_svector_ostream OS(sym_name);
  OS << "OMP_METADATA_TARGET_DATA"; //Magic value
  switch (k){
  case OMPD_target_data:
    OS << "_REGION";//Type and line as identifiers
    break;
  case OMPD_target_enter_data:
    OS << "_ENTER";
    break;
  case OMPD_target_exit_data:
    OS << "_EXIT";
    break;
  case OMPD_target_update:
    OS << "_UPDATE";
    break;
  default:
    llvm_unreachable("Unexpected directive kind");
  }
  OS << "_l" << std::to_string(pl.getLine());
}
```

Fragmento 12: Extracto del código de generación del json, en concreto el nombre del símbolo así como el identificador del objeto. La línea "Magic Value" representa el valor que el pase de optimización va a buscar en el módulo para extraer, mientras que el resto (tipo y línea) representan los identificadores de cada región o directiva.

Pases de optimización

Los pases implementados difieren bastante entre ellos – mientras que UCTransform ha de interpretar la información dada para construir un wrapper, una función y una serie de comentarios, UCMetadata simplemente copia los datos desde el símbolo hasta un archivo de texto. Por tanto, vamos a centrarnos en cómo funciona UCTransform.

El pase sigue las acciones descritas a continuación: Primero identifica una función como función de offloading generada con OpenMP, empleando el nombre de la función como valor identificativo. En el caso de que la función sobre la que está actuando no sea una función de offloading, simplemente la ignora.

Una vez está reconocida, obtiene los datos (“Named Attributes”) según los va necesitando. El primero es el nombre de la función, del que se derivan el nombre del wrapper, el nombre para la función que realiza el trabajo y el nombre del archivo al que va a extraer el código. Esto se hace para comenzar a construir las declaraciones de las funciones, especialmente las del wrapper.

El siguiente paso es extraer los argumentos de las funciones, tanto su nombre como su tipo. Aquí es donde toma importancia la última línea

resaltada en el Fragmento 8: cuando el generador extrae las variables para llamar a la función externa del AST, aquellas variables que han sido físicamente copiadas a la región (declaraciones “firstprivate” o implícitas) son transferidas como bloques genéricos de 64 bits (unsigned long). Es por ello necesario indicar su tipo real en la función mientras se declara como “unsigned long” en el wrapper (Fragmento 14).

Una vez se extraen las declaraciones de las funciones solo es necesario extraer el cuerpo de las mismas. Para el wrapper es necesario encontrar los argumentos capturados por copia, definir un puntero al tipo de dato original y obtener la dirección del unsigned long para copiarla en dicho puntero. Este proceso se realiza en paralelo al de extracción y emisión en el archivo de kernels de la información de los tipos originales debido a su relación.

```
//wrapper body
//wrps: Stream con salida al cuerpo del wrapper
//cw: Stream con salida específica a la llamada al worker
cw << F.getFnAttribute("Name").getValueAsString() <<
"_worker(";
for(int a = 0; a<numargs; a++){
  std::string argn = <nombre argumento a>
  std::string argt = <tipo original argumento a>
  out << "//Argument " << a << ": " << argn;
  out << ", original type " << argt;
  if(<argumento a es una referencia>){
    out << ", Captured by pointer\n";
    cw << "*" << argn;
  }else{
    out << ", Captured by copy, casted to void*\n";
    std::string aux = argn+"_cast";
    wrps << '\t' << argt << " *" << aux << "=";
    wrps << " (" << argt << "*) " << "&" << argn << ";\n";
    cw << "*" << aux;
  }
  cw << ", ";
}
wrps << "\treturn " <<
    call_wrk.substr(0,call_wrk.find_last_of(',')) << ");\n";
```

Fragmento 13: Obteniendo el wrapper y la información de tipos originales, piezas necesarias para el funcionamiento del framework y para la correcta optimización por parte del diseñador.

Una vez se ha obtenido el cuerpo del wrapper y las definiciones de ambas funciones solo queda emitirlas al archivo de texto, junto con el cuerpo de la función original obtenido como bloque de texto con las herramientas de LLVM. Se hace de la misma forma que con cualquier otro stream C++, en solo 10 lineas de texto (Fragmento 15).

```
SmallString<128> wrp_sig;
SmallString<128> wrk_sig;
{
  raw_svector_ostream os(wrp_sig);
  raw_svector_ostream ws(wrk_sig);
  os<<"unsigned " << <wrapper_fun_name> <<'(';
  ws<<"unsigned " << <original_fn_name> <<"_worker"<<'(';
  int i=0;
  <numargs>

  std::string wrp_type, wrk_type;
  while(i<numargs){
    std::string atype = <arg i qualified type>;
    size_t aux = atype.find("&");
    wrk_type = atype;

    //wrp_type es wrk_type* si wrk_type es derreferenciable
    //o wrk_type en otro caso

    //If the type isn't ref captured, use ulong as 64b generic
    if(<arg i captura por referencia>){
      os << wrp_type;
    }else{
      os << "unsigned long ";
    }
    ws << wrk_type;
    …
    std::string asize="";
    … //Obtenemos tamaño array en asize
    os<< ' ' << <nombre arg i> << asize <<", ";
    ws<< ' ' << <nombre arg i> << asize <<", ";
          ++i;
     }
}
int end = wrp_sig.find_last_of(',');
```

Fragmento 14: Extracto simplificado del código que emite las declaraciones de las funciones wrapper (wrp_sig) y worker (wrk_sig). Se omiten algunos detalles específicos de como se llega a ciertas variables

```
out<<"\nextern \"C\" "<<wrp_sig.substr(0,
    wrp_sig.find_last_of("("))<<");\n\n";

out<<wrk_sig.substr(0, wrk_sig.find_last_of(','))<<"){\n";
out<<input->getBuffer()<<cut<<"\n\treturn 0;\n}";
out<<"\n\n";

out<<wrp_sig.substr(0, end)<<"){"<<'\n';
out<<wrp_body<<"}\n";
out.flush();
```

Fragmento 15: Emisión del código al archivo. Vemos que el wrapper se especifica como extern "C" *para evitar "name mangling". Cut es un carácter para emparejar la llave de inicio en la línea anterior en función de como termina el buffer.*

Compilando y ejecutando el prototipo

Llegados a este punto todavía está sin integrar la toolchain para nuestro dispositivo, por lo que la parte de enlazado se ha de hacer manualmente. Eso requiere volver a la Figura 6 y replantearnos cómo queda el sistema si en lugar de hacerlo con el sistema estándar lo hacemos incluyendo nuestros pases y con enlazado manual. El esquema final es como sigue:

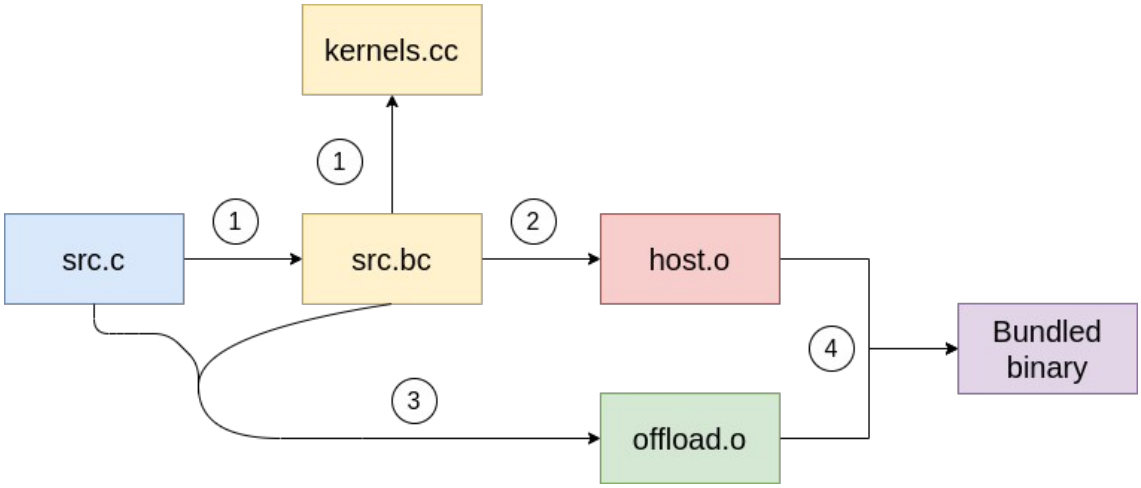

Figura 9: Esquema sin linkado automático y con extracción. Vemos dos diferencias clave en ① y en ④

Resulta que ahora el proceso ha cambiado. En lugar de enlazar los objetos para su ejecución, en ④ los agrupa con una herramienta llamada "clang-offload-bundler", que sirve para unir y separar bloques de código no enlazados para offloading. Nuestro objetivo ahora es conseguir, a partir del "bundled binary" y los "kernels", un Fat Binary que permita su ejecución. Una de las partes es sencilla: podemos obtener objetos a partir de los kernels con simplemente compilarlos para el dispositivo target. También empleamos la capacidad de "unbundling" de la herramienta introducida para volver a separar host.o y offload.o:

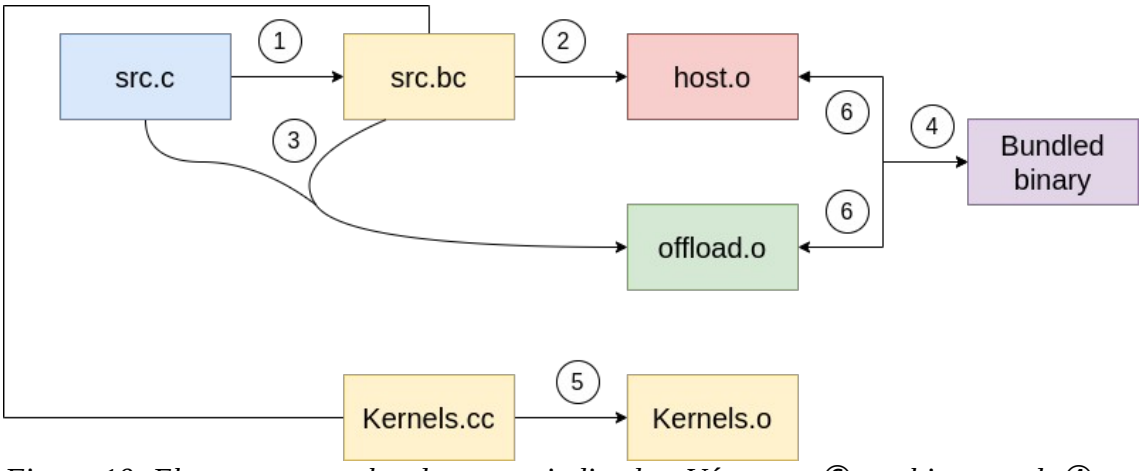

Figura 10: El esquema tras los dos pasos indicados. Véase que ⑥ es el inverso de ④

Desde aquí solo queda repetir los últimos 4 pasos de la Figura 6 cambiando nuestro runtime por el objeto de los kernels en la parte de dispositivo. El esquema final queda como sigue:

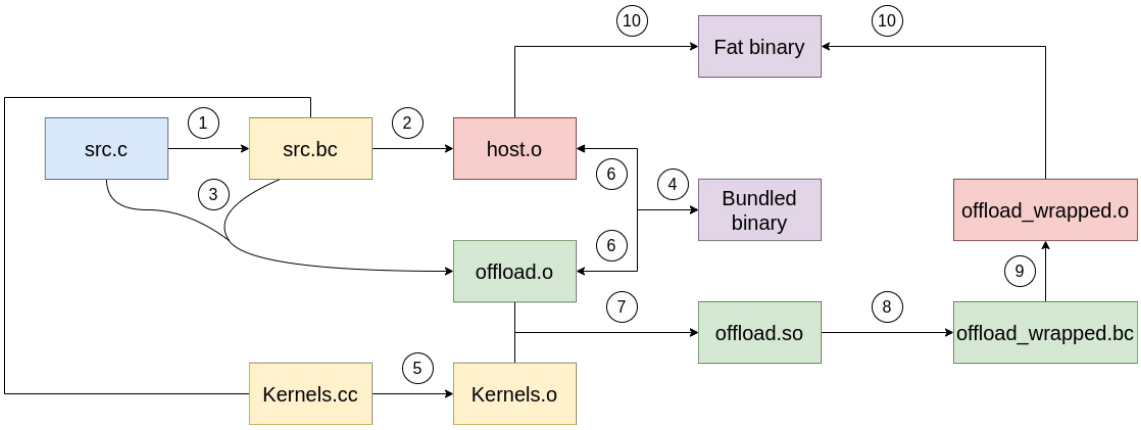

Figura 11: Esquema final. Nótese que los pasos ⑦, ⑧, ⑨ y ⑩ corresponden a los pasos ④, ⑤, ⑥ y ⑦ de la Figura 6

Para poder compilar el programa en un ejecutable de momento se tienen que hacer a mano los pasos ⑤ a ⑩. En el anexo 1 se incluyen las órdenes que se emplearon a la hora de hacer las pruebas.

Una vez se obtiene el Fat Binary final, éste se ejecuta como cualquier otro binario. La parte diferente está en el momento de ejecutar una sección target. En un binario normal, a la sección target se entra mediante la llamada __tgt_target, de libomp, y se ejecuta el código que se encuentra en offload.o originalmente. Pero en nuestro sistema, en offload.o no está el código a ejecutar[5] sino que nos encontramos es una llamada al código extraído. Si utilizamos un esquema de llamadas, en este caso existe una doble indirección:

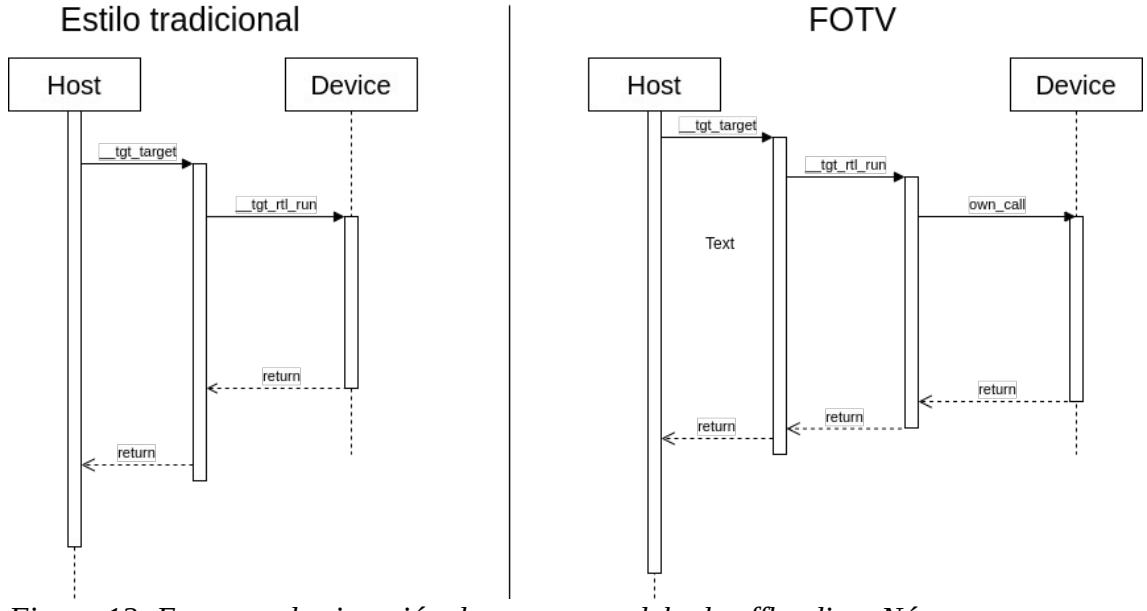

Figura 12: Esquema de ejecución de nuestro modelo de offloading. Nótese que tenemos control total sobre la segunda indirección en FOTV

5 Técnicamente sí está pero solo como fallback en caso de fallo en la función real y solo para el host

De esta manera conseguimos nuestro objetivo, aunque de forma un poco tosca en este primer prototipo: Podemos ejecutar cualquier código que deseemos en el dispositivo mientras responda a un wrapper que se proporciona tras la compilación, por lo que podemos utilizar este esquema para ejecutar código de todo tipo de aceleradores.

A partir de aquí el trabajo que queda es el de incrementar la funcionalidad y usabilidad. En los próximos meses se implementará el runtime y la toolchain, permitiendo la utilización del sistema de forma automatizada. Lo único que no incluiremos será la automatización de la compilación para kernels, dado que nuestro objetivo era la flexibilidad. La aproximación propuesta proporciona una implementación base para el host, y un wrapper para ejecutar el código extraído en cualquier dispositivo al que se pueda conectar. Pero por diseño no conocemos los dispositivos que se van a conectar y, por tanto, no podemos incluir backends para éstos. Se detallarán un poco estos trabajos en el último apartado.

5.-Conclusiones y trabajo futuro

En este proyecto se ha explorado la extensión de un compilador de código abierto, LLVM/Clang, para descargar código a un dispositivo acelerador genérico, de forma que no sea necesario integrar el proceso de compilación para dicho dispositivo en LLVM/Clang. La aproximación también permite que el código que se va a descargar pueda ser modificado y adaptado al dispositivo final, después de que el código OpenMP haya sido compilado, gracias a un proceso de extracción del código fuente del acelerador en tiempo de compilación y de carga del binario específico del acelerador en tiempo de ejecución.

El trabajo realizado ha incluido un profundo análisis del funcionamiento interno del proceso de descarga (offloading) en LLVM/Clang y del propio entorno de compilación, la modificación del driver y frontend para incluir un nuevo dispositivo genérico, la inclusión en el IR de nuevos metadatos que faciliten la extracción de código, el desarrollo del soporte al dispositivo genérico durante el proceso de compilación y la generación de toda la infraestructura necesaria para incluir el soporte a dicho dispositivo en el Fat Binary que genera el compilador como ejecutable. Además, se han creado pases de optimización para extraer a ficheros externos el código fuente que será descargado a los dispositivos target, así como un amplio conjunto de metadatos en formato json (como información sobre los datos que se transfieren entre el host y el dispositivo o los lazos incluidos en directivas OpenMP orientadas a paralelismo de datos) que facilitan la optimización y adaptación del código extraído.

El proyecto desarrollado facilita la programación de sistemas heterogéneos en donde las metodologías actuales normalmente emplean interfaces de programación (como OpenCL) que únicamente soportan paralelismo de datos. El uso de OpenMP también permite utilizar otros paradigmas, como paralelismo de tareas. Además, el proyecto evita tener que incluir en LLVM/Clang el soporte a la compilación de código a nuevos dispositivos, lo que permite utilizar los entornos propios de cada dispositivo, que generan una implementación o binario ejecutable a partir de código fuente C/C++.

Aunque tenemos un prototipo funcional del sistema, aún no podemos decir que el desarrollo esté finalizado. Nos faltan dos partes esenciales para poder distribuir una versión pública: principalmente la

integración de la “toolchain” y la adaptación de la librería de runtime. En concreto, nos tenemos que centrar en la automatización de la compilación por ambos lados.

Para implementar la toolchain es necesario asociar un compilador, un ensamblador y un linker a FOTV. Pese a que podemos simplemente copiar los usados en x86 (concretamente en la toolchain GNU), debemos detallar una serie de particularidades de nuestro target. En concreto tenemos que centrarnos en dos puntos de contención:

1. Nuestro objeto target no es autocontenido, esto es, incluye una llamada a una función que no está dentro del propio objeto. Es necesario definir en el linker la adición de un objeto externo, que será el que en la Figura 6 llamamos rtl_dev.so, en definitiva, la sección de offloading del runtime.
2. Aunque nuestro dispositivo sea esencialmente un x86, es imperativo que el target destino del wrapper sea FOTV. En caso de ser x86 se podría sobreescribir un módulo de offloading x86 ya existente en el Fat Binary, que se emplea también para el offloading a aceleradores tipo Xeon Phi.

Una vez se limen esas asperezas en la toolchain, quedaría adaptar el runtime, que debería ser “separado” en dos objetos: uno para el host, que necesitaría una API de control por el usuario y se incluiría de forma consciente, y uno para el dispositivo, que sería incluido automáticamente por el compilador con la toolchain modificada. El objeto del dispositivo sería el soporte para las acciones de manejo automático de recursos de cómputo y se encargaría de la carga y descarga de kernels bajo demanda, mientras que el de host tendría un acceso a dicho control para tomar control manual de ciertas características. La parte del host requiere una API (ya existente) que sirva de puente entre el código de usuario y el runtime.

Una vez completados estos dos hitos se tendrá una versión lista para usuarios finales, que cumple con los objetivos detallados en Diseño e Implementación.

No quiero terminar este trabajo sin agradecer a las personas que me han ayudado con la investigación y desarrollo que han sido necesarias para llegar hasta este punto, así como las que han hecho que el trabajo en sí sea posible. Agradezco por tanto a mi director, Pablo Sánchez, que me ha apoyado durante todo el proceso de aprendizaje y desarrollo desde enero; a Ángel Álvarez, que dejó detrás más de 2GB de documentación en digital e impresa que han sido cruciales para encontrar todas las piezas que encajar en el enorme proyecto que es LLVM; al coordinador del máster, Jose Luis

Bosque, que me ha permitido cambiar de proyecto cuatro días antes del plazo final, y a los directores del anterior proyecto, Mario Aldea y Héctor Pérez, que no han puesto ninguna pega al cambio pese a haber trabajado con ellos desde agosto de 2019 y encontrarnos con problemas por la situación actual.

Bibliografía

1. “OpenMP Application Program Interface” Version 4.0, Julio 2013
2. Liao, C., Yonghong Yan, B. Supinski, D. Quinlan y B. Chapman. “Early Experiences with the OpenMP Accelerator Model.” IWOMP (2013).
3. L. Sommer, J. Korinth and A. Koch, "OpenMP device offloading to FPGA accelerators," 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), Seattle, WA, 2017, pp. 201-205, doi: 10.1109/ASAP.2017.7995280.
4. H. Yviquel and G. Araújo, "The Cloud as an OpenMP Offloading Device," 2017 46th International Conference on Parallel Processing (ICPP), Bristol, 2017, pp. 352-361, doi: 10.1109/ICPP.2017.44.
5. Carlo Bertolli, Samuel F. Antao, Gheorghe-Teodor Bercea, Arpith C. Jacob, Alexandre E. Eichenberger, Tong Chen, Zehra Sura, Hyojin Sung, Georgios Rokos, David Appelhans, and Kevin O'Brien. 2015. Integrating GPU support for OpenMP offloading directives into Clang. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (*LLVM '15*). Association for Computing Machinery, New York, NY, USA, Article 5, 1–11. DOI: https://doi.org/10.1145/2833157.2833161
6. S. F. Antao et al., "Offloading Support for OpenMP in Clang and LLVM," 2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), Salt Lake City, UT, 2016, pp. 1-11, doi: 10.1109/LLVM-HPC.2016.006.
7. J. Korinth, D. de la Chevallerie y A. Koch, "An Open-Source Tool Flow for the Composition of Reconfigurable Hardware Thread Pool Architectures," 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, Vancouver, BC, 2015, pp. 195-198, doi: 10.1109/FCCM.2015.22.
8. Vivado Design Suite, https://www.xilinx.com/products/design-tools/vivado.html
9. Apache Spark: Unified analytics engine for Big Data processing. https://spark.apache.org
10. Apache Hadoop, implementación Open Source de MapReduce. http://hadoop.apache.org

11. The Scala Programming Language, https://www.scala-lang.org
12. Ángel Álvarez, Íñigo Ugarte, Víctor Fernández y Pablo Sánchez, “Design Space Exploration in Heterogeneous Platforms Using OpenMP”, XXXIV Conference on Design of Circuits and Integrated Systems, 2019.
13. Ángel Álvarez, Íñigo Ugarte, Víctor Fernández y Pablo Sánchez, “OpenMP Dynamic Device Offloading in Heterogeneous Platforms”, 15th International Workshop on OpenMP, 2019.
14. “DawnCC : a Source-to-Source Automatic Parallelizer of C and C++ Programs”, Gleison Souza et al., 2016
15. Marcio M. Pereira, Rafael C. F. Sousa y Guido Araujo, “Compiling and Optimizing OpenMP 4.X Programs to OpenCL and SPIR” in Proceedings of the 13th International Workshop on OpenMP (IWOMP 2017)
16. LLVM 11.0.0 Documentation Overview, 12 de Octubre de 2020. https://releases.llvm.org/11.0.0/docs/index.html
17. LLVM Language Reference, version 11.0.0, 12 de Octubre de 2020. https://releases.llvm.org/11.0.0/docs/LangRef.html
18. Página del proyecto en Github. https://github.com/llvm/llvm-project
19. C. Bertolli et al., "Coordinating GPU Threads for OpenMP 4.0 in LLVM," 2014 LLVM Compiler Infrastructure in HPC, New Orleans, LA, 2014, pp. 12-21, doi: 10.1109/LLVM-HPC.2014.10.

ANEXO I: Secuencia de compilación

```
$ clang  -Xclang -load -Xclang ./UCTransform.so -Xclang\
> -load -Xclang ./UCMetadata.so -c -fopenmp \
> -fopenmp-targets=fotv  <src>.c

$ clang-offload-bundler -unbundle -type=o \
> -targets=host-x86_64-gnu-unknown,openmp-fotv \
> -outputs=test_hst.o,test_dev.o -inputs=<src>.o

$ clang -c <extracted_code>.cc –target=fotv[6]

$ clang -shared -fPIC -g -fopenmp -m64 -o test_dev.so \
> test_dev.o <extracted_code>.o

$ clang-offload-wrapper -o test_dev_wrap.bc test_dev.so \
> --target=fotv

$ clang -c test_dev_wrap.bc

$ ld -z relro -m elf_x86_64 –eh-frame-hdr \
> -dynamic-linker /lib64/ld-linux-x86-64.so.2 \
> /usr/lib/x86_64-linux-gnu/crt1.o \
> /usr/lib/x86_64-linux-gnu/crti.o \
> /usr/lib/gcc/x86_64-linux-gnu/7.5.0/crtbegin.o \
> -L/usr/lib/gcc/x86_64-linux-gnu/7.5.0 \
> -L/usr/lib/x86_64-linux-gnu \
> -L/lib/x86_64-linux-gnu \
> -L/lib64 -L/usr/lib/x86_64-linux-gnu -L/usr/lib \
> -Lclang/install/dir/lib -L/lib -L/usr/lib  \
> -lomp -lomptarget -lgcc  \
> --as-needed -lgcc_s \
> --no-as-needed -lpthread -lc -lgcc \
> --as-needed -lgcc_s \
> --no-as-needed \
> /usr/lib/gcc/x86_64-linux-gnu/7.5.0/crtend.o \
> /usr/lib/x86_64-linux-gnu/crtn.o \
> -o test_exec test_hst.o test_dev_wrap.o

$./test_exec

[Resultado]
```

6 Para cada archivo extraído de código