UNIVERSIDAD POLITECNICA DE VALENCIA

GRADO EN INGENIERÍA INFORMÁTICA

UNIVERSIDAD
POLITECNICA
DE VALENCIA

“Técnicas de paralelización de código para robots basados en emociones”

TRABAJO FINAL DE GRADO

Autor:
Francisco Almenar Pedrós

Tutor:
Pedro López Rodríguez

Cotutor:
Houcine Hassan Mohamed

Valencia, 2014

RESUMEN

Las arquitecturas de control basadas en emociones se están convirtiendo en una de las soluciones más prometedoras a la hora de implementar los sistemas de los robots. Los encargados de controlar estos sistemas son los procesos emocionales, que sirven de guía para el robot a la hora tomar una decisión sobre qué comportamientos se han de activar para completar sus objetivos. El número de estos procesos emocionales se incrementa en gran medida conforme lo hace la complejidad del problema, haciendo que la capacidad de cálculo de un procesador de un solo núcleo no sea suficiente para este trabajo. Por suerte, estos sistemas son altamente paralelizables y por lo tanto se puede incrementar la capacidad de cálculo del equipo en gran medida empleando técnicas de paralelización.

En este TFG vamos a emplear distintas técnicas de paralelización con el objetivo de acelerar el cálculo de las emociones que determinarán el comportamiento de los robots. Para cumplir este objetivo utilizaremos y compararemos Unidades de Procesamiento Gráfico (GPU) con procesadores de múltiples núcleos y con el uso de instrucciones SIMD (Single Instruction Multiple Data).

Palabras clave:
Robots, multinúcleo, GPUs, SIMD, sistema de emociones

ABSTRACT

Control architectures based on emotions are becoming promising solutions for the implementation of future robotic systems. The basic controllers of this architecture are the emotional processes that decide which behaviors the robot must activate to fulfill the objectives. The number of emotional processes increases (hundreds of millions/s) with the complexity level of the application, limiting the processing capacity of the main processor to solve the complex problems. Fortunately, the potential parallelism of emotional processes permits their execution in parallel, hence enabling the power computing to tackle the complex dynamic problems.

In this paper, Graphic Processing Unit (GPU), multicore processors and single instruction multiple data (SIMD) instructions are used to provide parallelism for the emotional processes.

Keywords:
Robots, multicore, GPUs, SIMD, emotional system

ÍNDICE

MOTIVACIÓN

Los sistemas que permiten a los robots tomar decisiones de forma similar a los humanos aún están en desarrollo. Para facilitar la llegada de estas capacidades a los robots es necesaria una gran capacidad de cálculo, la cual no se consigue fácilmente. Para poder alcanzarla, necesitamos tecnologías que nos lo permitan. Puede ser que en un futuro cercano se descubran nuevas tecnologías que nos permitan alcanzar estas necesidades. Sin embargo, con la tecnología actual también se pueden conseguir buenos resultados. Por eso, la motivación de este trabajo es demostrar que existen formas de conseguir excelentes resultados empleando las tecnologías actuales. Para conseguirlo, vamos a emplear componentes que se pueden encontrar en los ordenadores actuales y por lo tanto no necesitaremos utilizar ningún tipo de hardware adicional.

Las propuestas que realizamos en este TFG aprovechan las capacidades de cómputo en paralelo de los procesadores multinúcleo y de los coprocesadores gráficos. Estas propuestas son las siguientes:

- Procesadores multinúcleo, que incorporan varios núcleos de procesamiento, los cuales se encuentran en todos los procesadores actuales. Estos procesadores nos permitirían ganar potencia de cálculo, por el mero hecho de aprovechar sus capacidades inherentes.

- Unidades de procesamiento gráfico (GPUs), que incorporan múltiples elementos de procesamiento, inicialmente orientadas al procesamiento de gráficos, y posteriormente utilizadas para cálculos de propósito general. Existe un amplio abanico de alternativas con diferentes prestaciones y precios.

- Instrucciones SIMD, son instrucciones vectoriales que incorporan todos los procesadores actuales, lo que permite la explotación de un paralelismo adicional en cada uno de los núcleos que integran los procesadores.

La motivación principal del TFG es comprobar si estas tecnologías, disponibles en los sistemas actuales, nos pueden acercar a las necesidades de potencia de cálculo que los robots tendrán en un futuro.

INTRODUCCIÓN

Muchos trabajos de investigación [1-5] predicen que en las próximas dos décadas, tanto las industrias como los hogares familiares experimentarán un incremento en el número de robots inteligentes que poseen. La idea principal que busca la industria del robot es conseguir que estos pueden tener una inteligencia similar a la de un humano, concretamente buscan imitar nuestra capacidad para tomar decisiones por nosotros mismos, por supuesto esto es realmente difícil, sin embargo estos trabajos [2,6], indican que robots con esta capacidad, la de tomar decisiones por sí mismos, ya están en desarrollo y que los primeros prototipos podrían estar listos para el 2030. Esta habilidad para tomar decisiones de forma autónoma es necesaria, pues se espera que estos robots sean capaces de realizar acciones como los humanos, algunas de estas actividades pueden ser: operaciones de abstracción y generalización, diagnósticos médicos y planificación y toma de decisiones [3-5]. Para poder realizar estas tareas es necesario que sean capaces de tomar sus propias decisiones y de sacar sus propias conclusiones.

Algunos trabajos indican que ya podemos ver robots con capacidades, aunque realmente son aproximaciones. Este es el caso del Robot Warrior desarrollado por iRobot [7], este robot tiene la habilidad para resolver tareas simples adaptándose a pequeños cambios en el entorno. Pese a que este programa es capaz de ser ejecutado en un procesador de un solo núcleo sin grandes problemas, se espera que conforme avancen estas tecnologías, y los robots alcancen inteligencias similares a las de los humanos, el requisito computacional sea mucho mayor, por poner un ejemplo, se espera que para el 2050 los robots necesitarán ejecutar miles de millones de operaciones por segundo, pero no hace falta esperar tanto para encontrar aplicaciones que necesitan más capacidad, este el caso de algunas sistemas robóticos, en éstos los procesadores mononúcleo son insuficientes para dicha tarea.

Las arquitecturas de control basadas en emociones, se están convirtiendo en una solución prometedora a la hora de implementar sistemas robóticos avanzados [3,8-11]. Un punto a favor del uso de esta arquitectura es que facilita el proceso de toma de decisiones [12,13], además permite obtener resultados en menos tiempo, esto se debe a que está pensada para ser ejecutada en paralelo. Esta arquitectura posee las siguientes ventajas: permite al robot ser autónomo para centrar su atención en el comportamiento que sea más prometedor, provee una respuesta acotada en el tiempo, lo que ayuda a la organización de los procesos de decisión, permite la ordenación de los problemas basándose en las expectativas de éxito, además de que permite al robot adaptar la carga computacional según la capacidad del procesador disponible y de la complejidad del problema.

En este trabajo final de grado, se va a emplear la arquitectura basada en emociones, ésta se usará para implementar la toma de decisiones de un supuesto robot. Bajo el modelo basado en emociones coexisten dos tipos principales de procesos: las conductas o comportamientos y las emociones. Las conductas son las soluciones a los problemas a los que se enfrenta el

robot, mientras que las emociones son los mecanismos que motivan al robot a realizar la conducta más conveniente.

Los procesos de cálculo emocional se han de aplicar a todos los problemas/subproblemas que se encuentran en la agenda del robot, conforme esta agenda se hace más compleja, la carga de trabajo que ha de realizar el procesador aumenta significativamente, esto incrementa la cantidad de MOPS (millones de operaciones por segundo) que ha de realizar, en nuestro proyecto se entiende como operación el cálculo de una emoción, es decir, el tratamiento de 6 observaciones, cada tratamiento consiste en el cálculo de la tangente hiperbólica, y además se realiza la suma de esas 6 observaciones tratadas, este proceso se encuentra más detallado en secciones posteriores (ver sección 1, arquitectura emocional). Inicialmente todos los procesos (incluyendo las conductas y las emociones) eran ejecutados en el procesador principal del ordenador (ej. Intel a 2.6 GHz). Sin embargo, el ordenador principal no aguantaba la carga de trabajo que se le imponía, dado que su capacidad de cómputo solo alcanzaba los 25 MOPS.

Los resultados muestran que, pese a que las GPUs pueden tener un cuello de botella en la transmisión de datos entre el host (CPU) y el dispositivo (GPU), las GPUs de media y alta gama pueden solucionar hasta los problemas de mayor complejidad que hemos empleado, mientras que las de gama baja superan los problemas hasta una complejidad media. Las instrucciones SIMD no son suficientes para solventar los problemas más complejos y en algunos de dificultad media, los procesadores de dos núcleos muestran resultados similares a dichas instrucciones, mientras que los de cuatro núcleos tienen un rendimiento cercano al de las GPUs de gama baja.

El resto del trabajo final de grado se divide de la siguiente forma, en la sección 1 describimos la arquitectura emocional y el problema que vamos a tratar, en la sección 2 se describen las tecnologías empleadas y la implementación del problema para esa tecnología, en la sección 3 se muestra la evaluación de las distintas herramientas empleadas y por último la sección 4 muestra las conclusiones obtenidas a partir de la evaluación.

TRABAJO PREVIO

El punto de partida de este trabajo es un artículo realizado por uno de los cotutores, en este artículo se realizaba un análisis de la arquitectura basada en emociones, para el caso concreto de las FPGAs. Este artículo implementaba el sistema utilizando FPGAs, en concreto se usó los modelos Statrix III y Statrix IV, pero que pese a obtener buenos resultados se vio que era una solución con problemas de viabilidad debido a su excesivo coste.

Teniendo en cuenta la inherente facilidad para ser paralelizado de la arquitectura, se ha decidido emplear técnicas, que se presentarán en la sucesivas secciones, para solucionar las aplicaciones del robot que se proponen en la sección de evaluación, lo que se busca con estas propuestas es obtener buenos resultados sin tener que utilizar hardware especial y caro, por eso las propuestas empleadas utilizan hardware que se puede encontrar en cualquier ordenador actual.

Con la idea de buscar alternativas económicas al problema se han empleado tres implementaciones distintas que utilizan técnicas distintas para obtener el paralelismo y con las que no se había trabajado hasta ahora, a excepción de la implementación multihilo, pues ya había trabajado anteriormente con Open MP.

Para facilitar la comparación se va a dedicar una sección entera a la evaluación y comparación de los resultados obtenidos, en esta evaluación se comparará con los resultados que se obtuvieron con la implementación para FPGAs con el fin de facilitar la comparación y comprender mejor los resultados obtenidos.

1. ARQUITECTURA EMOCIONAL

1.1 Especificación del proceso emocional

Una emoción se puede describir como el proceso de observar una situación y motivar ciertas conductas del robot, con la intención de que éste pueda superar la situación actual.

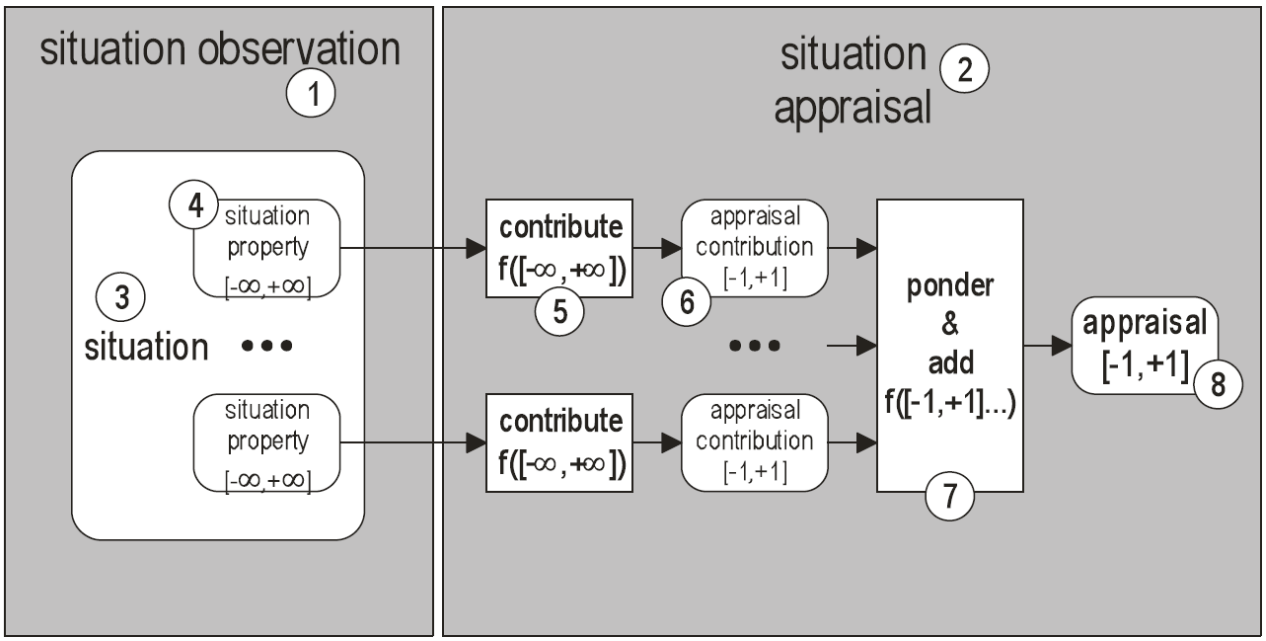

Figura 1: Proceso de observación de la situación actual.

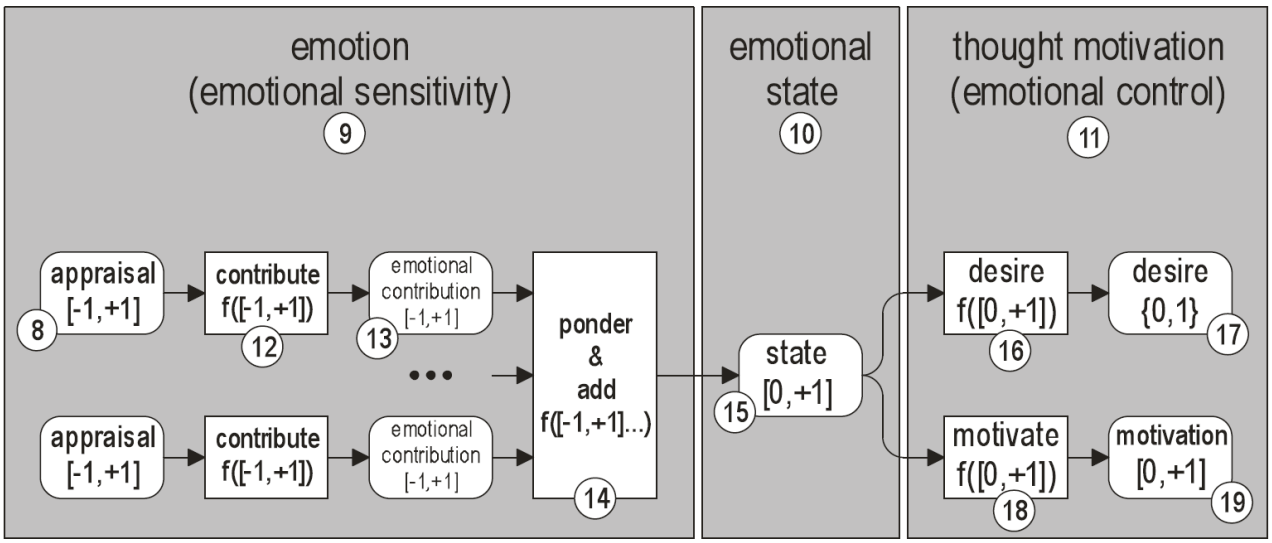

Figura 2: Proceso de motivación emocional.

La figuras 1 y 2 detallan el proceso emocional que motiva las acciones del robot. Las situaciones (3) son generadas por los procesos de observación (1) y son representadas como un conjunto de propiedades reales (4) que conforman dicha situación. El proceso de valoración (2) depende del robot y

de su carácter, este carácter se ajusta dinámicamente a los parámetros de este proceso, para calcular la valoración de la situación (8) el robot realiza una ponderación y suma un conjunto de contribuciones (6), que se evalúan mediante el uso de funciones de contribución (5). La ecuación 1 representa la i^{th} valoración de la situación actual.

$$a_i = \sum_{\{k=1\}}^{l} w_{ak} * f_{ak}\,(p_k)$$

Dónde:

p_k es la k^{th} propiedad de la situación, f_{ak} es la k^{th} función de contribución, w_{ak} es el peso de la función y l es el número de observaciones que contribuyen a la emoción, su valor oscila dentro del rango de 1 a 6.

La figura 2 muestra el proceso de motivación emocional, este proceso se divide en dos fases. La primera, activación emocional (9), establece el estado emocional (10), la segunda fase llamada la respuesta emocional construye y motiva las conductas 11). Las contribuciones emocionales (13), evaluadas a través de funciones de contribución (12), son ponderadas y sumadas (14) para finalmente generar un estado emocional (15). Las funciones de contribución emocional están definidas dentro del rango [-1, +1] y el estado emocional puede tomar valores entre el 0 y el 1. La respuesta emocional crea nuevos deseos (16, 17), y esto motiva el comportamiento de completar los deseos (18,19).

Una emoción se representa de la siguiente manera:

$$s_j = \sum_{\{i=1\}}^{l} w_{ci} * f_{ci}(a_i)$$

Dónde:
s_j es el estado de la emoción j^{th}, f_{ci} es la i^{th} función de contribución, w_{ci} es el i^{th} peso de la función.

Las funciones de contribución emocional, f_{ci}, deben de tener ciertas propiedades, por ejemplo deben tener pequeñas variaciones, al final del rango de valores, que tienden hacia valores asintóticos y variaciones abruptas. Para conseguir que nuestro sistema cuente con estas propiedades se han empleado tangentes hiperbólicas, como se puede ver en la siguiente función:

$$f_{ci(x)} = \mathrm{th}(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Dónde:

X es el valor de la observación a_i cuando se realiza el cálculo de la emoción. Para permitir pequeños ajustes en el cálculo de la tangente hiperbólica, se ha transformado la ecuación 3 de la siguiente forma:

$$th^{*(x)} = \left(\frac{e^{2(x-x_0)\delta_y} - 1}{e^{2(x-x_0)\delta_y} + 1} - y_0\right) * \delta_y$$

Los parámetros x_0, y_0, δ_y permiten la traslación y el reescalado de las funciones de contribución.

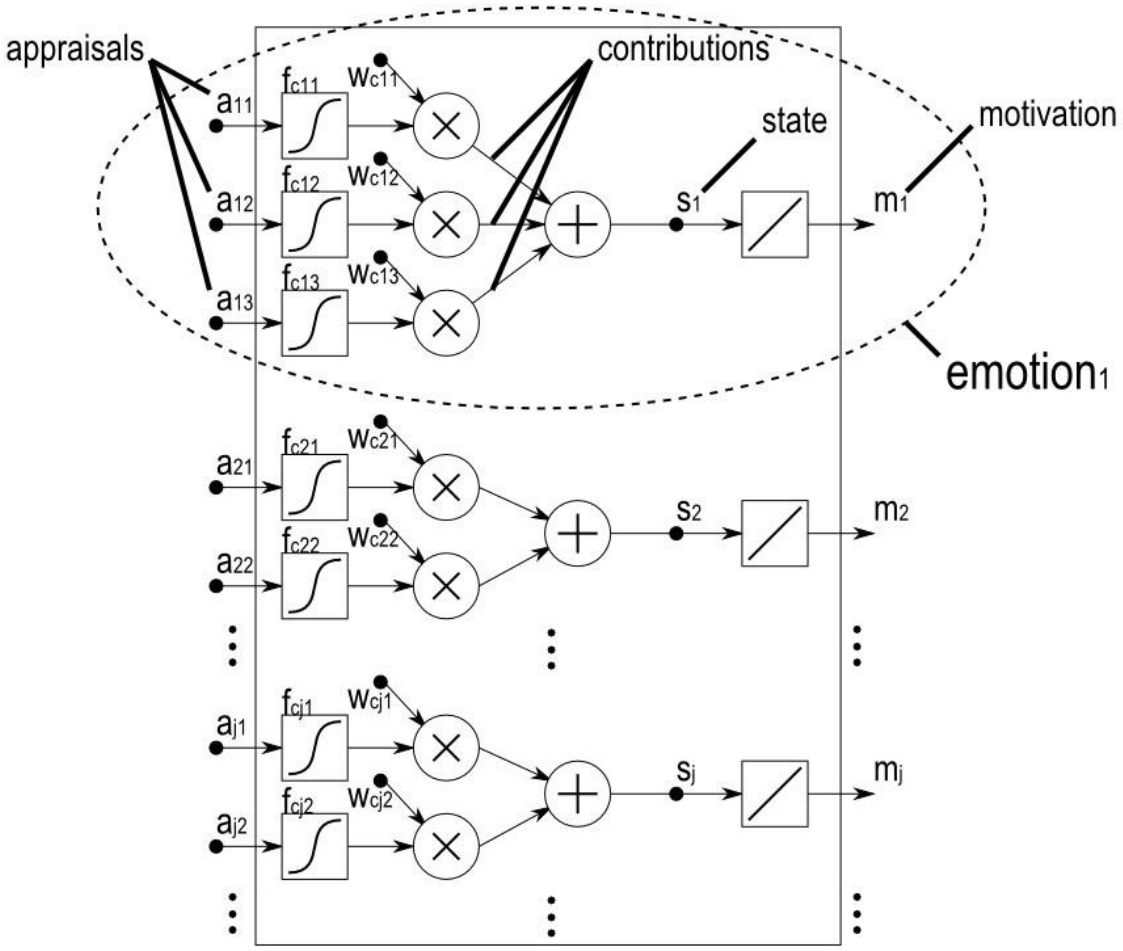

Figura 3: Estructura del sistema emocional

Estas emociones se agrupan en sistemas emocionales y su estructura es la que se puede ver en la figura 3. Este sistema recibe una entrada desde un conjunto de *n* observaciones (por ejemplo 2M, donde M hace referencia a millones) y produce un set de *k* motivaciones (por ejemplo 0.5M). La tangente hiperbólica se aplica a cada observación, el resultado obtenido es multiplicado por el peso de esa observación. Cada emoción está compuesta por hasta 6 funciones de contribución distintas. Las emociones obtenidas pueden pasar hacia la función final (f_i). En este TFG, se usa la función

identidad y por lo tanto, no se realiza ningún postproceso. El número total de observaciones en el sistema emocional depende de la complejidad del problema, así como de las condiciones del robot. En el caso de estudio la cantidad de observaciones empleadas puede alcanzar los 200M.

Se ha realizado una implementación inicial del sistema, esta implementación está realizada en secuencial. El objetivo de esta implementación servir como base para el resto de implementaciones, no obstante, estas operaciones pueden ser calculadas en paralelo, esto se debe a que no existe ninguna dependencia entre ellas y por lo tanto no hay problema en que se ejecuten a la vez. Debido a los requisitos computacionales del sistema basado en emociones, el controlador (se considera que el controlador del robot solo posee un núcleo) del robot es incapaz de completar todos los objetivos en algunos escenarios, por lo que quedaba clara la necesidad de cómputo adicional. Debido a esta necesidad de cálculo y a que es un sistema altamente paralelizable, se decidió emplear técnicas de paralelización con la finalidad de poder superar todos los escenarios propuestos.

En la siguiente subsección se hablará del algoritmo secuencial implementado.

1.2 Algoritmo secuencial

En esta subsección se presentarán las partes más importantes del código secuencial, explicando aquellas secciones que forman la parte principal del programa, sin embargo no presentaremos el código completo. Todos los códigos completos se presentarán en los anexos pertinentes.

La parte más importante del algoritmo diseñado se encuentra en el cálculo de la tangente hiperbólica y en la posterior agrupación, por lo que estas serán los fragmentos de código que se mostrarán en cada subsección de implementación. Cada implementación paralela puede necesitar que se expliquen partes adicionales del código, de esta forma se facilita la compresión de la herramienta empleada.

El fragmento de código que se muestra está implementado en el lenguaje de programación C. En él se puede ver que a partir de la observaciones (*a*) recibidas, se realiza el cálculo de las tangentes hiperbólicas, así como la posterior generación de las emociones. Por motivos de simplicidad, los factores de traslación y de reescalado, no han sido incluidos en el desarrollo y además cada emoción estará formada siempre por 6 funciones de contribución.

```
//tangente hiperbólica
for(i=0;i<n;i++){
        fci[i] =
                (exp(2*a[i])-1)/
                (exp(2*a[i])+1);
}
```

```
//agrupación
for (i=0;i<n/6;i++){
        for(j=0; j<6; j++){
                acum += fci[6*i+j];
        }

        m[i]= acum;
}
```

2. HERRAMIENTAS DE PARALELIZACIÓN

En esta sección vamos a hablar de las técnicas de paralelización que se han empleado, estas técnicas son las siguientes: paralelización a través de hilos, utilizando unidades de procesamiento gráfico (GPUs) y las instrucciones SIMD.

Para la paralelización a nivel de hilos se empleará una API denominada Open MP (OMP), esta API soporta la memoria compartida entre procesadores y está disponible en los lenguajes de programación C, C++ y fortran. En el caso de las GPUs se empleará un lenguaje de programación, CudaC, creado por Nvidia que nos permitirá emplear la capacidad de cómputo de estas unidades. Por último implementaremos el sistema de emociones utilizando un set de instrucciones que se introdujeron en 1999 en todos los procesadores, este set recibe el nombre de Single Instruction Multiple Data(SIMD) y permiten el cálculo de una operación a conjuntos de datos a la vez.

Por cada propuesta paralela hay una sección de implementación en la que podemos ver parte del código empleado. En las subsecciones de implementación, veremos como se ha de modificar el código original para cada implementación en paralelo. Si se desea ver el código completo, en el anexo Códigos completos, podrá acceder a ellos.

En las siguientes subsecciones se profundizará más en las herramientas empleadas y en la implementación de nuestro sistema basado en emociones usando estas herramientas.

2.1 Procesadores multinúcleo

La mayoría de procesadores actuales son multinúcleos, es decir, el procesador de estos ordenadores posee dos o más microprocesadores (unidades de procesamiento o CPU) independientes, estos procesadores reciben su nombre en función de la cantidad de microprocesadores que tienen, es decir, un procesador con dos microprocesadores recibirá el nombre de doble núcleo. Los procesadores multinúcleo permiten que un dispositivo tenga la capacidad para ejecutar un proceso en varios microprocesadores a la vez, este tipo de paralelismo thread-level parallelism (TLP), o lo que es lo mismo paralalismo a nivel de hilo. Cada microprocesador se encarga de la realización de una parte del proceso original.

Los procesadores fueron creados originalmente con un solo núcleo, este núcleo se encargaba de ejecutar todos los procesos de forma secuencial, sin embargo, a partir del siglo XXI se empezaron a introducir los primeros procesadores con más de un núcleo. En la actualidad existen procesadores con más de 50 CPUs (por ejemplo el procesador Xeon phi de Intel), aunque estos están reservados para servidores. Como ya hemos comentado anteriormente y como se podrá ver en la sección 3.3, un solo núcleo no es suficiente para implementar un sistema basado en emociones, así que hemos decidido comprobar si, al utilizar más núcleos (dos y cuatro núcleos) los resultados obtenidos son suficientes, es decir, si nos permiten obtener la capacidad de cálculo requerida por el sistema. En las pruebas realizadas utilizaremos hasta cuatro núcleos trabajando en paralelo.

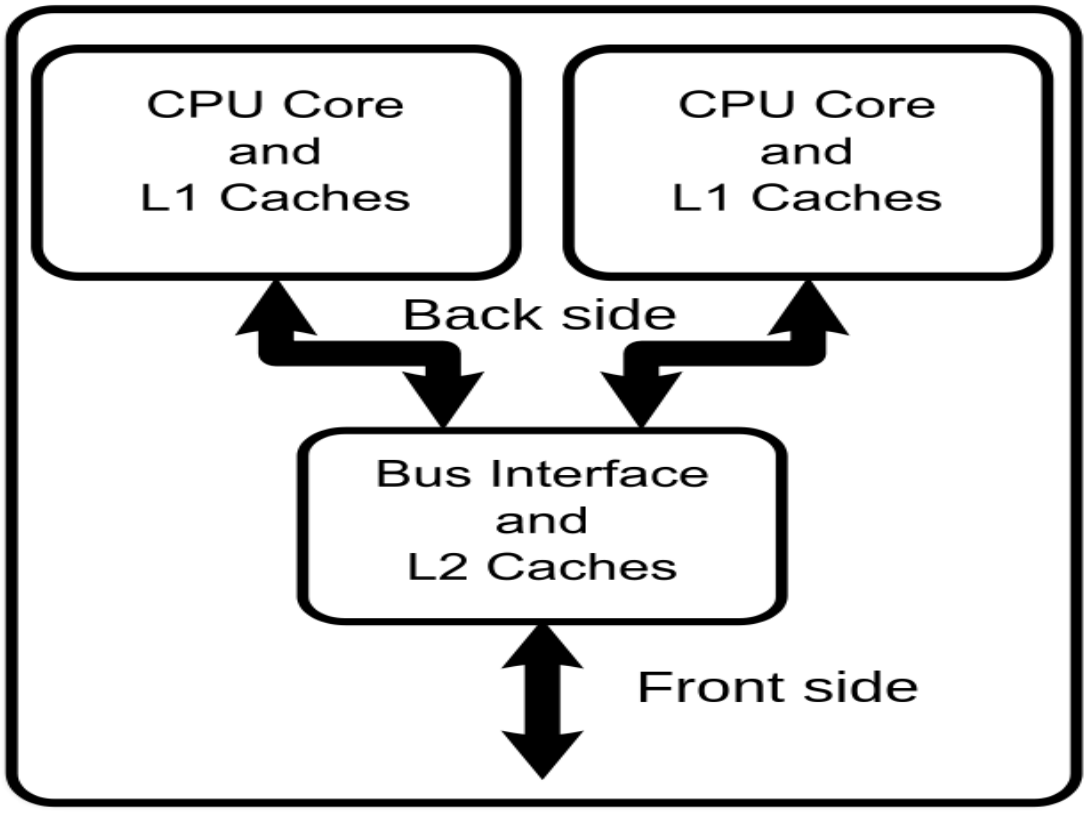

Figura 4: Diagrama de un procesador genérico de dos núcleos.

La primera implementación en paralelo se ha realizado utilizando procesadores multinúcleo. Se eligió este método debido a que la mayoría de procesadores de hoy en día poseen más de un núcleo, lo que permite que se puedan ejecutar varios hilos en a la vez, se ha empleado la API Open MP (OMP) para conseguir paralelismo a nivel de hilos. La API OMP modifica el comportamiento en tiempo de ejecución, esto permite obtener paralelismo a nivel de hilos. Su uso se basa en el empleo de directivas, éstas permiten al programador comunicarse con el compilador y gracias a esta comunicación, el compilador sabe que secciones del código se han de ejecutar en paralelo y cuáles no, además permite emplear otra serie de capacidades, entre dichas capacidades se encuentran: identificar que variables son compartidas entre todos los hilos y cuáles no, permite indicar que orden de ejecución que deben llevar estos hilos, el tipo de reparto del trabajo entre los hilos... Esta API, con el fin de paralelizar proceso deseado, divide el hilo maestro en un número

específico de hilos esclavos, los hilos esclavos reciben parte más pequeña de la tarea que tenía que hacer el hilo maestro. Si la paralelización se ha hecho bien, cada hilo esclavo puede llevar a cabo su tarea a la vez que el resto, sin que esto afecte de forma negativa al resultado.

2.2 Implementación multinúcleo

La implementación utilizando OMP no es muy diferente de la que se utilizó en el algoritmo secuencial, aun así hay que tener en cuenta diversos factores. Debido a que OMP declara todas las variables públicas (compartidas entre los hilos) por defecto, hemos de declarar explícitamente que las variables j, acum son privadas dado que queremos evitar a toda costa las condiciones de carrera, por ejemplo si no declarásemos que acum es privada, podría darse el caso en el que los valores de acum se vieran alterados debido a los cálculos que se están llevando a cabo en los otros hilos, generando un resultado erróneo. Para indicar que el bucle en cuestión se va a realizar en paralelo, utilizamos la directiva *#pragma omp parallel for*, para el preprocesador, justo antes del bucle, además añadimos *private (j,acum)* para declarar esas variables privadas. A continuación podemos ver el código empleado para implementar el bucle principal.

```
#include <omp.h>
...
#pragma omp parallel for private (j,acum)
For (i=0;i<n/6;i++){

        acum = 0;
        for (j=0; j<6; j++){

                fci[6*i+j]=
                        (exp(2*a[6*i+j])-1)/
                        (exp(2*a[6*i+j])+1);

                acum += fci[6*i+j];
        }

        m[i] = acum;
}
...
```

La API empleada está desactivada por defecto, así que para poder emplearla tenemos que indicárselo al compilador. En nuestro caso dado que hemos empleado en compilador gcc nos basta con añadir opción *–fopenmp*, es decir, si llamamos a nuestro programa tfg.c para compilarlo usaríamos *gcc tfg.c –o tfg –fopenmp*, y para ejecutarlo en un entorno Linux tendríamos que ejecutar *./tfg. S*e puede controlar cuantos hilos vamos a lanzar mediante la directiva *export OMP_NUMTHREADS = X,* donde X es el número de hilos que se van a lanza a la vez. Esta directiva se puede incluir en el código paralelizado antes de la directiva #pragma, o bien, se puede indicar en la consola de sistema (en Linux) antes de lanzar la ejecución basta

2.3 Unidades de procesamiento gráfico

La unidad de procesamiento gráfico o GPU (siglas de la palabra en inglés graphic processing unit) es un coprocesador especializado, éste ha sido diseñado para manejar rápidamente la creación de imágenes, además su memoria esta modificada para que acceda a ella a mayor velocidad, por lo tanto los accesos y los guardados en la memoria son más rápidos. Las GPUs modernas son muy eficientes con el manejo de los gráficos por ordenador, además su estructura está diseñada para sacar un gran rendimiento en las tareas de paralelización. Las GPUs se encuentran presentes en los ordenadores de sobremesa, normalmente se encuentran incluidas en las tarjetas gráficas, aunque también las podemos encontrarlas integradas en la placa base o en algunos procesadores, como es el caso de los procesadores de la familia Core i de Intel.

La GPU es un coprocesador gráfico que en un principio estaba dedicado al procesamiento de gráficos, sobretodo en el campo de los videojuegos, no obstante, actualmente también se emplea para el cálculo de operaciones en coma flotante, normalmente se usa para aligerar la carga de trabajo del procesador central, esto permite que se pueda centrar en otros procesos. Dado que en este proyecto las operaciones que predominan son en coma flotante, se ha decidido que realizar una implementación del sistema utilizando GPUs, porque esto podría generar buenos resultado, además debido a que hoy en día los ordenadores comunes tienen una GPU, no se necesita invertir en hardware adicional, lo que es uno de los objetivos principales de este trabajo. Además debido al buen rendimiento obtenido en la pruebas, las GPUs podrían ser un buen sustituto para las FPGAs, las cuales tenían un coste adicional importante.

La principal diferencia entre CPUs y GPUs, reside en la especialización, mientras que las CPUs no están especializadas, las GPUs se han especializado en el cálculo de operaciones en coma flotante, esto se debe a que este tipo de operaciones predominan en las operaciones gráficas en 3D y por lo tanto cumplen lo que perseguían, acelerar el cálculo de los gráficos para permitir el uso de herramientas gráficas más potentes.

Otra de las diferencias con las CPUs radica en su arquitectura. A diferencia del procesador central que normalmente tiene una arquitectura von Neumann, la GPU se basa en el Modelo Circulante, este modelo facilita el procesamiento en paralelo. Como se puede ver en la figura 5 la GPU tiene muchas más unidades de cálculo que la CPU, pese a que son más simples y trabajan más lentas, dan mejores resultados debido a su cantidad, otra diferencia que podemos ver es el tamaño de caché, la caché es menor en las GPUs dado que no tiene tanta importancia como en las CPUs.

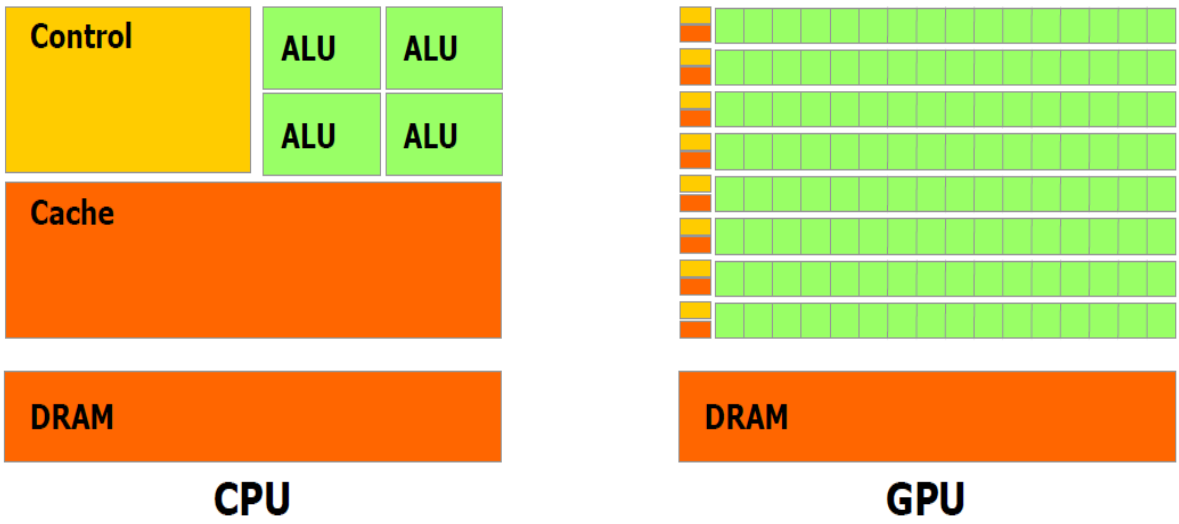

Figura 5: La GPU dedica más transistores al procesamiento de datos

CUDA es el modelo de programación para GPUs empleados, es proveido por Nvidia a los programadores que usan sus tarjetas gráficas. Este lenguaje deriva del lenguaje de programación C. La estructura que se utiliza en este modelo está definida por un grid o parilla, este grid está formado por bloques y cada uno de los bloques contiene una cantidad idéntica de hilos, esta cantidad es modificable por el programador. Estos hilos que conforman los bloques se ejecutan concurrentemente en un SM (Streaming Processor), esto es idóneo para la ejecución en paralelo, pues la unidad se encuentra dividida en muchas unidades de procesamiento y cada una de estas unidades puede realizar un cálculo a la vez que el resto, por lo que el cálculo de las tangentes hiperbólicas, así como el posterior agrupamiento, se hace de forma mucho más rápida. Durante el proceso, los datos se dividen entre los distintos hilos que conforman los bloques del grid. Debido a que la GPU se encuentra en el dispositivo (tarjeta gráfica) los datos han de ser enviados al dispositivo antes de poder ser utilizados, este proceso de transferencia se realiza a través de los buses PCI express y de normal puede suponer un cuello de botella sino se trata con cuidado, este retraso es provocado porque la GPU muy rápido, tanto que muchas veces trata los datos a mayor velocidad que a la que el PCI se los suministra. En este trabajo se ha considerado la peor situación y es que todas las observaciones cambian a cada iteración, por lo que cada vez que el robot debe tomar una decisión, se debe enviar todas las observaciones a través del bus y una vez calculadas las emociones, el host ha de recibir todas las emociones calculadas por la GPU, por lo que los resultados obtenidos pueden ser considerados pesimistas, pero aun así superan al resto de propuestas. Hay diversas técnicas que pueden reducir el impacto, por ejemplo se podría enviar simplemente las observaciones que han cambiado, debido a que la toma de decisiones se realiza cada muy poco tiempo, algunas de las observaciones no tendrían que haber cambiado, por ejemplo el terreno podría ser el mismo o la distancia al objetivo podría no haber variado significativamente, aunque esta técnica supondría añadir un pequeño sobrecargo al controlador, pues éste tendría que decidir que observaciones han sido modificadas, además habría que identificar su posición en el array de observaciones, dado que es vital modificar el valor correspondiente o los

resultados podría ser erróneos. Otro método para mejorar estos resultados podría ser el solapamiento del envío de las observaciones con el cálculo de las emociones, para más información consultarse [16].

Las arquitecturas de las tarjetas gráficas de Nvidia han ido cambiando y mejorando, estos avances han permitido al lenguaje de programación CUDA crecer y añadir nuevas funcionalidades. Estas nuevas funcionalidades no están disponibles en arquitecturas anteriores, por lo tanto algunos métodos pueden no estar disponibles para nuestra GPU, por esta razón tenemos que tener claro qué funciones podemos emplear y cuales no. La característica de la gráfica que nos indica su arquitectura es la capacidad de computo (compute capability) y por lo tanto tenemos que tener en cuenta que versión tenemos. Algunas de las características mencionadas son: la posibilidad de emplear operaciones en coma flotante de doble precisión, la cantidad de hilos que se pueden lanzar por bloque, permitir la ejecución de instrucciones atómicas en las secciones paralelas, paralelismo dinámico... En nuestro ejemplo hemos decidido emplear funciones utilizables en todas las arquitecturas.

2.4 Implementación GPU

Esta era la primera vez que empleábamos una GPU para paralelizar código, por supuesto tampoco sabíamos nada del lenguaje de programación CUDA, debido a esto hemos encontrado algunos problemas a la hora de implementar el sistema, los más destacados han sido debimos al a la capacidad de cómputo y de las limitaciones que este genera, por ejemplo: la cantidad de bloques y de hilos por bloque se ve afectada por esta capacidad, además si superamos las posibilidades de la unidad gráfica, el compilador no nos avisa y termina el proceso sin incidencias, por lo que si medimos el tiempo de ejecución el resultado es excesivamente pequeño, pues aborta la operación, esto generó confusión con la cantidad de operaciones por segundo que podía realizar, los resultados eran confusos y poco fiables. Otro problema con el que nos encontramos fue la falta de prints, resulta que no podemos imprimir por pantalla directamente des del dispositivo, a no ser que se tenga una gráfica con una capacidad de cómputo alta, lo que provocó que no se pudieran mostrar los resultados intermedios, o que fuese costoso obtenerlo, esto se tradujo en que el método principal de corrección de errores se centró en el análisis teórico del código y en la interpretación de los errores.

A continuación podemos ver parte del código empleado para implementar el sistema basado en emociones en GPUs usando CUDA (para ver el código completo vaya al anexo códigos completos):

```
#include <cuda.h>

__global__ void hyperbolicTangent(float *dev_a, int *dev_n, float *dev_fci) {
        int tid = threadIdx.x + blockIdx.x * blockDim.x;

        while (tid < *dev_n) {
```

```
                dev_fci[tid] = tanh(dev_fci[tid]);
                tid += blockDim.x * gridDim.x;
        }

}

__global__ void reduction (float *dev_fci, float *dev_m, int *dev_n) {

        int tid = 6*threadIdx.x + blockIdx.x * blockDim.x;
        int tidemotion = threadIdx.x + blockIdx.x * blockDim.x;
        int cont;

        while (tid < *dev_n) {

                cont = 0;
                dev_m[tidemotion] = 0;

                while (cont <= 5) {

                        dev_m[tidemotion] += dev_fci[tid+cont];
                        cont ++;
                }

                tid += 6 * blockDim.x * gridDim.x;
                tidemotion += blockDim.x * gridDim.x;
        }
}

int main (...) {

        float *a, *fci, *m;
        float *dev_a, *dev_fci, dev_m;
...

        fci = (float*)malloc( n * sizeof(float) );
        a = (float*)malloc( n * sizeof(float) );
        m = (float*)malloc( n/6 * sizeof(float) );

        cudaMalloc( (void**)&dev_a, n * sizeof(float) );
        cudaMalloc( (void**)&dev_fci, n * sizeof(float) );
        cudaMalloc( (void**)&dev_m, n/6 * sizeof(float) );
        cudaMalloc( (void**)&dev_n, sizeof(int) );

        cudaMemcpy( dev_a, a, n * sizeof(float),
        cudaMemcpyHostToDevice );
        cudaMemcpy( dev_n, &n, sizeof(int),
        cudaMemcpyHostToDevice );

        hyperbolicTangent<<<blocks, threads>>>(dev_a, dev_n, dev_fci);

        reduction<<<blocks, threads>>>( dev_fci, dev_m, dev_n);

        cudaMemcpy( m, dev_m, (n/6) * sizeof(float),cudaMemcpyDeviceToHost );

...

        cudaFree( dev_a );
        cudaFree( dev_fci );
```

```
        cudaFree( dev_m );
        cudaFree( dev_n );
}
```

Antes de procesar las observaciones hemos de enviarlas a la memoria del dispositivo, pues recordemos que hace falta transferir la información al dispositivo antes de poder usarla, para esto, al igual que ocurre en C, tenemos que reservar memoria en la GPU, para conseguirlo tenemos que utilizar la función para realizar la reserva de memoria, *cudaMalloc((void**) & dev_app, n * sizeof(float))*, está función actúa de forma muy similar a *malloc*, empleada en el lenguaje C, el primer parámetro es un puntero a la variable para la que vamos a realizar la reserva, y como segundo parámetro tenemos la cantidad de bytes que vamos a reserva. Una vez que tenemos hecha la reserva podemos proceder con la transferencia de los datos, los datos irán desde la memoria principal hasta el dispositivo, para esto se ha empleado la función *cudaMemcpy* (un ejemplo sería *cudaMemcpy(dev_app, app, n * sizeof(float), cudaMemcpyHostToDevice*)), esta función recibe tres parámetros, el primero, que indica la variable donde van a ser copiados los datos, el segundo parámetro hace referencia a la variable desde donde se van a enviar y por último, el tercer parámetro indica la dirección en la que se va a hacer la transferencia, es decir, indicia si los datos van desde la memoria principal hasta la memoria del dispositivo o si van en sentido contrario, desde la memoria del dispositivo hasta la principal, por lo que también utilizaremos esta función para recuperar los valores finales, estos valores son los que serán utilizados por el robot para tomar decisiones.

Una vez tenemos las observaciones cargadas en la memoria del dispositivo, podemos pasar a ejecutar el cálculo de la tangentes hiperbólicas, para hacerlo utilizaremos la función *hyperbolicTangent*, esta función ha sido implementada por nosotros, se ejecutará en el dispositivo pero será invocada desde el controlador del robot y se encargará de tratar las observaciones, la invocación se realiza de una forma similar a la de cualquier función en C, pero hay una diferencia y es que se utiliza un parámetro más, este parámetro se especifica entre los signos *"<<<", ">>>"* y está compuesto por dos valores, el primero indica la cantidad de bloques que tiene el grid y el segundo marca la cantidad de hilos que tiene cada bloque, a la hora de elegir el valor de estos campos se suele hacer empíricamente, aunque Nvidia recomienda que se haga según la ocupación del bus, como se indica en el trabajo de Justin Luitjens y Dr. Steven Rennich [17], en nuestro caso hemos seguido la recomendación del fabricante, no obstante hay que tener en cuenta, que si estamos empleando tarjetas gráficas antiguas, existe un límite en la cantidad de hilos por bloque, como ya hemos mencionado antes. Respecto a las funciones *hyperbolicTangent* y *reduction*, como ocurre con todas funciones que se ejecutan en el dispositivo, hay que declarar el tipo de función que es, en este caso tanto el cálculo de la tangente, como la posterior agrupación en emociones son funciones global, esto indica que estas funciones se llaman desde el host, pero que se ejecutarán en el dispositivo, para especificarlo tenemos que usar el atributo *__global__* delante de su declaración. Durante la ejecución del cálculo de las tangentes hiperbólicas, así como la agrupación de la observaciones, necesitan un parámetro adicional para que a cada iteración de cada hilo, se acceda a la posición correcta del array y que

ninguno de los hilos acceda a una posición incorrecta del array, ya sea porque no exista esa posición o bien porque ya haya sido calculada anteriormente, para conseguir esto emplearemos tres parámetros, estos parámetros son característicos de CUDA y son inherentes al lenguaje*, threadIdx*, cuyo valor es único para cada hilo y lo identifica*, blockIdx*, nos indica a que bloque pertenece ese hilo y *blockDim* que nos indica cual es el tamaño de dicho bloque, la combinación de estos tres valores nos va a dar un resultado único, de esta forma no se accederá dos veces a la misma posición, además según el tamaño de la rejilla, a cada iteración cada hilo incrementará su propio valor, se hace para poder acceder al resto de posiciones que le corresponden a ese hilo. Debido a que este valor es irrepetible, podemos emplearlo tranquilamente para acceder a las posiciones de los vectores con los datos, no obstante tenemos que tener en cuenta que este valor nunca ha de superar el número total de observaciones y utilizamos la guarda del bucle while, de esta forma ese identificador interno nunca se saldrá de los límites.

Una vez que ambas funciones han sido ejecutadas, recuperamos las emociones que han sido calculadas en el dispositivo, para esto, volvemos a emplear *cudaMemcpy*, pero esta vez el último parámetro es *cudaMemcpyDeviceToHost,* esto indica que la transferencia va des del dispositivo hasta la memoria principal.

Una vez que la transferencia ha terminado y que ya tenemos la emociones en la memoria principal, pasamos a liberar la memoria que ha sido reservada en el dispositivo, para esto CUDA tiene una función muy similar a *free* del lenguaje de programación C, *cudaFree*, esta función recibe un parámetro, este parámetro no es más que un puntero a una variable para la que se ha reservado memoria en el dispositivo.

Esta vez no podemos emplear el compilador gcc, por esta razón vamos a emplear el compilador ofrecido por Nvidia, se llama nvcc y se puede compilar de la misma forma que se compila con el gcc, solo que el código generado estará listo para interactuar con la GPU. Un ejemplo de compilación y ejecución podría ser el siguiente *nvcc –o tfg tfg.cu*, una vez compilado podemos ejecutarlo desde un terminal de sistema de la siguiente forma ./tfg, para más información ver el código completo en el anexo.

2.5 Instrucciones SIMD

La última propuesta paralela es el uso de las instrucciones SIMD (Single Instruction Multiple Data), estas instrucciones forman parte del conjunto de instrucciones ISA (del inglés Instruction Set Architecture), es decir, del conjunto de instrucciones que una CPU de un ordenador puede entender y ejecutar. Fueron introducidas 1999 por INTEL, el set introducido por aquél entonces recibió el nombre de MMX. Para facilitar la comparación en secciones posteriores, la implementación de este sistema se ha hecho en un solo núcleo. Es posible combinar las instrucciones SIMD con los procesadores multinúcleo, esta técnica combinada está cogiendo cada vez más fuerza, por ejemplo la última versión de Open MP (la versión 4.0) permite esta combinación [18], como es de esperar la combinación de estas dos técnicas

generará mejores resultados que ambas por separado, aunque muy probablemente no dé como resultado la suma exacta. Podemos ver algunos ejemplos de su uso en el trabajo de Anoop Madhusood que se encuentra en la zona de desarrolladores de Nvidia [19].

Las instrucciones SIMD permiten explotar el paralelismo a nivel de datos, pero no a nivel de concurrencia, este paralelismo consiste en realizar la misma operación a distintos datos a la vez, es decir, que los operadores permitirán aplicar la misma operación a más datos a la vez, en el mismo instante. Los datos han de ser de tipo uniforme, por ejemplo se pueden hacer cuatro sumas distintas de variables tipo entero, sin embargo si algún operando pasa a ser de tipo decimal ya no se puede hacer. La unidad básica de las SIMD es el vector, el vector no es más que una fila de números o escalares. Normalmente las CPUs solo realizan una operación a la vez, sin embargo, las instrucciones SIMD permiten la ejecución de una operación a todos los componentes del vector a la vez, existen varios juegos de instrucciones SIMD, así que por lo tanto el tamaño de este vector dependerá del juego que se emplee, por ejemplo suponiendo que trabajamos con operandos en coma flotante de simple precisión (cada operando ocupa 32 bits), si el vector fuese de 128 bits se podrían hacer hasta cuatro operaciones a la vez [20, 21], por lo tanto podemos ver que, cuanto más grande sean los vectores más aceleración se obtendrá, sin embargo hay un límite, resulta que estas operaciones necesitan que el hardware del procesador esté preparado, pues no se pueden emplear las mismas ALUs (del inglés arithmetic logic unit), unidades que se encargan de realizar las operaciones, que realizan las operaciones normales, sino que se requieren ALUs especiales, capaces de trabajar con estos vectores de mayor tamaño, no obstante estas ALUs ya vienen incluidas en los procesadores de hoy en día y lo único que hay que tener en cuenta es la compatibilidad con cada juego de instrucciones, sobre todo con aquellos de diferente tamaño. Como hemos visto el tamaño del vector y el tamaño de las variables afectan al número de operaciones que se pueden procesar a la vez.

El desarrollo de las instrucciones SIMD, pese a que tuvo su origen para el mercado de los primeros superordenadores, se desarrolló debido a su introducción en 1995 a los ordenadores de sobremesa, aunque fue introducido por Sun Microsystems, el primer juego de instrucciones que se extendió fue el set MMX, estas instrucciones fueron introducidas por Intel para la arquitectura x86, esto ocurrió durante el año 1996. Otras compañías como IBM o Motorola hicieron sus propias versiones, esto provocó que hubiera una dura competencia entre las compañías, esto siguió así hasta que en 1999, Intel sacó un nuevo set de instrucciones SIMD denominado SSE, este juego de instrucciones introdujo, por primera vez, la capacidad de trabajar con vectores de hasta 128 bits, dejando el resto opciones muy retrasadas en este campo [22]. En este trabajo se han empleado dos juegos distintos de instrucciones SIMD, el set de instrucciones SSE y el set AVX, las SSE (del inglés Streaming SIMD Extensions) permiten trabajar con vectores de hasta 128 bits, lo que nos permite ejecutar 4 operaciones en coma flotante a la vez, recordemos que las operaciones en coma flotante son la base del sistema basado en emociones, en lo que respecta a las instrucciones AVX

(Advanced Vector Extensions) pueden trabajar con vectores de hasta 256 bits, lo que nos permite realizar hasta 8 operaciones con variables de 32 bits, el juego de instrucciones AVX fue introducido en marzo del 2008 y por lo tanto, necesitamos procesadores con arquitecturas de ese año, o posteriores, para poder utilizarlas, en el caso de los procesadores Intel, basta con que su arquitectura sea Sandy Bridge o posterior, en lo referente a los procesadores AMD, una arquitectura Bulldozer o más moderna es suficiente para poder emplearlas. Cabe destacar que la aceleración obtenida por este sistema no es directamente proporcional al número de operaciones que podemos ejecutar en paralelo, a diferencia de lo que ocurre en la paralelización a nivel de hilos, que suele obtener aceleraciones similares a la cantidad de hilos empleados, en este caso depende más del tipo de operaciones que estemos y de lo lentas que sean estas operaciones, aunque por supuesto, el tamaño de los vectores, así como la cantidad de operaciones que se ejecutan a la vez, afectan a la aceleración, cuanto mayores sean esos campos, mejores serán los resultados.

El uso de las instrucciones SIMD esta desactivado por defecto, para poder emplearlas debemos indicárselo al compilador indicarle el juego de instrccuiones a emplear. En nuestro caso vamos a emplear dos juegos y el compilador gcc, podemos habilitar las instrucciones SSE añadiendo la opción *-msseX,* esta opción se ha de añadir durante la compilación. Para activar las instrucciones AVX, es necesario incluir la opción *-mavX* durante la compilación. En ambos casos la X, empleada al final de cada opción, es un indicador de la versión de cada set que vamos a emplear, es recomendable hacerlo, pues cada versión introduce nuevas instrucciones y por lo tanto, si indicamos al compilador la versión empleada, facilitaremos su trabajo.

2.6 Implementación SIMD

La implementación utilizando instrucciones SIMD ha sido muy costosa, porque, al contrario que ocurría con las otras dos opciones, la cantidad de ejemplos y de información sobre esto está muy limitada, lo que ha provocado que el aprendizaje fuera lento y costoso, utilizando como herramienta principal la técnica de prueba y error. Una vez comprendidas la bases de su funcionamiento, el desarrollo se basaba en buscar, en la documentación del procesador, la función que se acoplaba más a nuestras intenciones, no obstante, muchas veces no existía la función que hiciese exactamente lo que queríamos, por lo tanto nos tocaba realizar una implementación alternativa, ésta tenía que dar el mismo resultado pero pasando por otro camino. Uno de estos problemas ocurrió a la hora de realizar el agrupamiento, necesitábamos una forma de realizar las sumas de los elementos de un vector y al final usamos las sumas horizontales para solucionarlo. Otro caso lo solucionamos añadiendo librerías externas, estas librerías nos daban la funcionalidad que buscábamos, éste fue el caso del cálculo de la tangente hiperbólica, pues no existía ninguna función que calculase la exponencial. Resumiendo esta implementación necesitó mucho tiempo de documentación e investigación.

El siguiente código muestra la implementación de la arquitectura basada en emociones utilizando las instrucciones SIMD, en concreto el set de

instrucciones SSE, para ver el código completo consulta el anexo códigos completos.

```
#include <emmintrin.h>
#include <mmintrin.h>
...

int main (...) {

	__m128 div, ptrPos, ptrNeg, ptr;
	__m128 ptr2, ptrEm, one, two,zero;

	...

	posix_memalign((void**)&fci,16,n * sizeof(float));
	posix_memalign((void**)&a, 16, n * sizeof(float));
	posix_memalign((void**)&m, 16, n/6 * sizeof(float));

...

	for (i=0; i<n/4; ++i) {

		ptr = _mm_load_ps(&a[4*i]);
		ptr = _mm_mul_ps(two, ptr);
		ptr = fmath::exp_ps(ptr);
		ptrNeg = _mm_sub_ps(ptr, one);
		ptrPos = _mm_add_ps(ptr, one);
		div =_mm_div_ps(ptrNeg,ptrPos);
		_mm_store_ps(fci+4*i, div);

	}

	j=0;

	for (i=0; i<n/6; ++i) {

		ptr = _mm_load_ps(&fci[4*j]);
		ptr2 =_mm_load_ps(&fci[4*(j+1)]);

		if (i % 2 == 0) {

			ptr2 = _mm_movelh_ps( zero, ptr2 );
		}

		else {

			ptr = _mm_movehl_ps( zero, ptr );
			j++;
		}


		ptrEm = _mm_hadd_ps(ptr,ptr2);
		ptrEm =_mm_hadd_ps(ptrEm,zero);
		ptrEm =_mm_hadd_ps(ptrEm,zero);
		_mm_store_ps(&aux2[0], ptrEm);
		m[i] = aux2[0];
```

```
            j++;
        }

        …
}
```

Como se ha indicado en el apartado anterior, tenemos que indicarle al compilador que vamos a emplear las instrucciones SIMD, para esto tenemos que indicar las correspondientes opciones de compilación y además, debemos de incluir las distintas librerías que se van a emplear y que contienen las instrucciones SIMD, en el caso del juego de instrucciones SSE, tenemos que realizar dos inclusiones, uno para la librería *emmintrin* y el otro para *mmintrin*, cuando empleamos el set AVX tenemos que añadir la librería *immintrin*. En este proyecto hemos empleado el juego de instrucciones 3.0 para las SSE y el 1.0 para las AVX.

Vamos a proceder a explicar el código empleado para la implementación utilizando las instrucciones del set SSE y una vez terminado explicaremos las diferencias con el juego de instrucciones AVX. A la de declarar las variables tenemos que indicar que estas van a ser registros de 128 bits, es por esta razón que las variables sobre las que se va a operar son de tipo *_m128*, no obstante las variables donde se almacenan las observaciones (*a*) y donde se almacenarán las emociones generadas (*m*) siguen siendo de tipo float, esto se debe a que se ha intentado abstraer esta función del proceso total, permitiendo al procesador principal actuar como si esta función no necesitase ningún tratamiento especial. Para el correcto funcionamiento de los vectores a la hora de reservar memoria para las observaciones y emociones está tiene que estar alineada, pues sino pueden ocurrir errores de lectura y de acceso, para alinear la memoria basta con emplear la función de reserva de memoria *posix_memalign*. Una vez tenemos el tema de la memoria claro, tenemos que pasar la información que está almacenada en el vector de observaciones a, hacía los registros de 128 bits, para realizar esta carga de información se puede utilizar la instrucción *_mm_load_ps()*, esta función recibe como parámetro una dirección de memoria y carga en el registro los siguientes 128 bits empezando desde la posición del puntero enviada, es decir, los que cargará los siguientes cuatro floats a partir del índice de acceso del vector utilizado. Como podemos ver en nuestro código, esta instrucción se emplea por primera vez para cargar 4 valores del vector de observaciones *a* a cada iteración del bucle, guardándolos en el registro *ptr*.

Una vez realizada la carga de los valores, comienza el cálculo de las tangentes hiperbólicas, como el set instrucciones SSE no incluye todas las operaciones, hemos necesitado la ayuda de una librería externa, esta librería ha sido creada por Shigeo [22] y entre otras funciones incluye el cálculo de la exponencial, esta operación es necesaria para calcular la tangente hiperbólica, como se indica en la sección 1 arquitectura emocional, esta librería se llama *fmath*. De forma similar a lo que ocurría con los valores de entrada, el resultado obtenido tras el cálculo ha de ser guardado en el vector de observaciones tratadas, para esto utilizamos la función *_mm_store_ps*, esta instrucción nos permite, dado un registro de 128 bits como parámetro de entrada, guardar valores en una posición de memoria, esta posición de

memoria suele pertenecer a un array y se emplea como primer parámetro de la función de guardado. Hemos de tener en cuenta que la duración del bucle ya no es *n*, siendo n el número de observaciones totales, sino que es *n/4*, esto se debe a que a cada iteración del bucle calculamos 4 tangentes hiperbólicas a la vez, y por lo tanto solo se necesitan *n/4* iteraciones.

Una vez terminado el cálculo de todas las tangentes hiperbólicas, comienza el proceso de generación de las emociones, como sabemos, en este trabajo se ha tomado la decisión de agruparlas las observaciones de seis en seis, sin embargo el tamaño de los registros empleados es de cuatro floats, por lo tanto necesitamos un tratamiento especial para generar las emociones. Se necesitan dos variables del tipo *_m128*, pero de esta forma cargamos ocho valores en lugar de los 6 necesarios, por eso utilizaremos una instrucción de desplazamiento, esta instrucción combinará el contenido del registro a modificar con el de otro registro, este registro auxiliar estará cargado con el valor cero, de esta forma dos de los ocho valores pasarán a ser cero y por lo tanto no afectarán al valor de nuestra emoción, las instrucciones empleadas son *_mm_movelh_ps()* y *_mm_movehl_ps()*, la primera cambia los valores que ocupan la tercera y la cuarta posición del primer registro con los valores que ocupan la primera y segunda posición del segundo registro, la segunda instrucción hace lo contrario, los valores que ocupan la primera y segunda posición del primer registro cambian su posición con los que ocupan la tercera y la cuarta posición del segundo registro, esto se debe a que en las iteraciones impares trabajamos con los seis primeros valores y en las pares con los seis últimos evitando así el solapamiento, veámoslo con un ejemplo el array *a* contiene los siguientes valores [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12], los dos registros empleados cogerán en la primera iteración los valores *reg* = [1, 2, 3, 4] y *reg2* = [5, 6, 7, 8] con la instrucción *_mm_movehl_ps(reg2, zero)* el registro dos pasará a valer [5, 6, 0, 0], con lo que ya no causará ningún problema en la suma, en la segunda iteración *reg* = [5, 6, 7, 8] y *reg2* =[9, 10, 11, 12], como estamos en una iteración par, el registro uno es el que es modificado para quedarnos con los seis últimos valores, así pues, empleamos la instrucción *_mm_movelh_ps(reg,zero)* y obtenemos *reg* = [0, 0, 7, 8] y *reg2* =[9, 10, 11, 12]. El contador *j* se utiliza de forma especial para evitar que se lea varias veces el mismo vector de cuatro de elementos y se incrementa dos veces en las iteraciones impares evitando así solapamientos. Cuando tenemos ambos registros con los valores que necesitamos pasamos a hacer la suma de sus valores, es decir, la agrupación de las observaciones ya tratadas, esta suma de registros se hace a través de tres sumas horizontales, la suma horizontal se realiza mediante la suma de pares, con el siguiente ejemplo podremos ver cómo funciona exactamente, dados dos registros *a* = [A3, A2, A1, A0] y *b* = [B3, B2, B1, B0] obtenemos como resultado el registro *c* =[B3 + B2, B1 + B0, A3 + A2, A1 + A0]. Tras la realización de las tres sumas horizontales tendremos en el primer valor del registro final la emoción de cada iteración, que como hemos explicado tendrá que ser guardada en el vector *m*, para realizar la suma horizontal utilizamos la instrucción *_mm_hadd_ps ()*.

Para la implementación de las AVX basta con utilizar funciones equivalentes que nos permiten trabajar con vectores de 256 bits. Veamos a continuación algunos de los cambios que hay que realizar:

- Durante la declaración de tipos, en lugar de declarar que usaremos registros __*m128*, emplearemos registros __*m256*.

- La función de carga pasa a ser _mm256_load_*ps* en lugar de *_mm_load_ps*, que es la función empleada con el juego de instrucciones SSE.

- Las operaciones empleadas para el emular el cálculo de la tangente hiperbólica son las mismas, pero las funciones empleadas son las equivalentes para 256 bits y al igual que ha ocurrido en los casos anteriores basta con indicar detrás de *m* el *256.*

- El cálculo de exponencial es un poco distinto, recordemos que esta operación no se encontraba en el set inicial de operaciones incluidas por estas instrucciones, por fortuna para nosotros existe una versión para vectores de 256 bits para la librería empleada con las SSE.

- La agrupación de las observaciones ya tratadas tiene pequeñas alteraciones en el algoritmo empleado, pues la estrategia empleada con los vectores de 128 bits ya no se sirve, esto se debe a que ahora trabajamos con ocho floats a la vez y no con grupos de cuatro, por lo que el control de las superposiciones no es el mismo. En este caso, tendremos un array auxiliar de 8 floats que se encargará de cargar los 6 valores a evaluar este ciclo, a parte tendrá dos ceros ya precargados en las últimas dos posiciones, la suma se hará con tres sumas horizontales. A continuación el código de esta parte

```
for (i=0; i<n/6; ++i) {

    for( j=0; j<6; j++){

            aux [j] = fci[i+j];
    }

    ptr = _mm256_load_ps(&auxVec);


    ptrEm = _mm256_hadd_ps(ptr,ptr2);
    ptrEm =_mm256_hadd_ps(ptrEm,zero);
    ptrEm =_mm256_hadd_ps(ptrEm,zero);

    _mm256_store_ps(&m[i], ptrEm);
 }
```

3. EVALUACIÓN

3.1 Aplicación del robot

A un robot, con la capacidad de moverse, se le pueden asignar muchas tareas, por ejemplo el transporte de materiales, la limpieza, la vigilancia, la realización de diagnósticos sobre terrenos entre otros. En algunos casos puede verse obligado a realizar varias operaciones a la vez, lo que podría requerir una gran potencia de cálculo, por lo que la aplicación de un sistema basado en emociones, recordemos que su arquitectura está pensada para ser paralelizada, podría darnos la potencia requerida, permitiendo que el robot tome la decisión más acertada a tiempo.

En este trabajo no se ha querido perder de vista la alcanzabilidad, por lo que no solo se ha buscado una solución que de buenos resultados, sino que además, se ha intentado que esté al alcance de la mayoría y que no necesite una gran inversión en hardware específico, por esta razón las herramientas que se han empleado se encuentran en los ordenadores actuales.

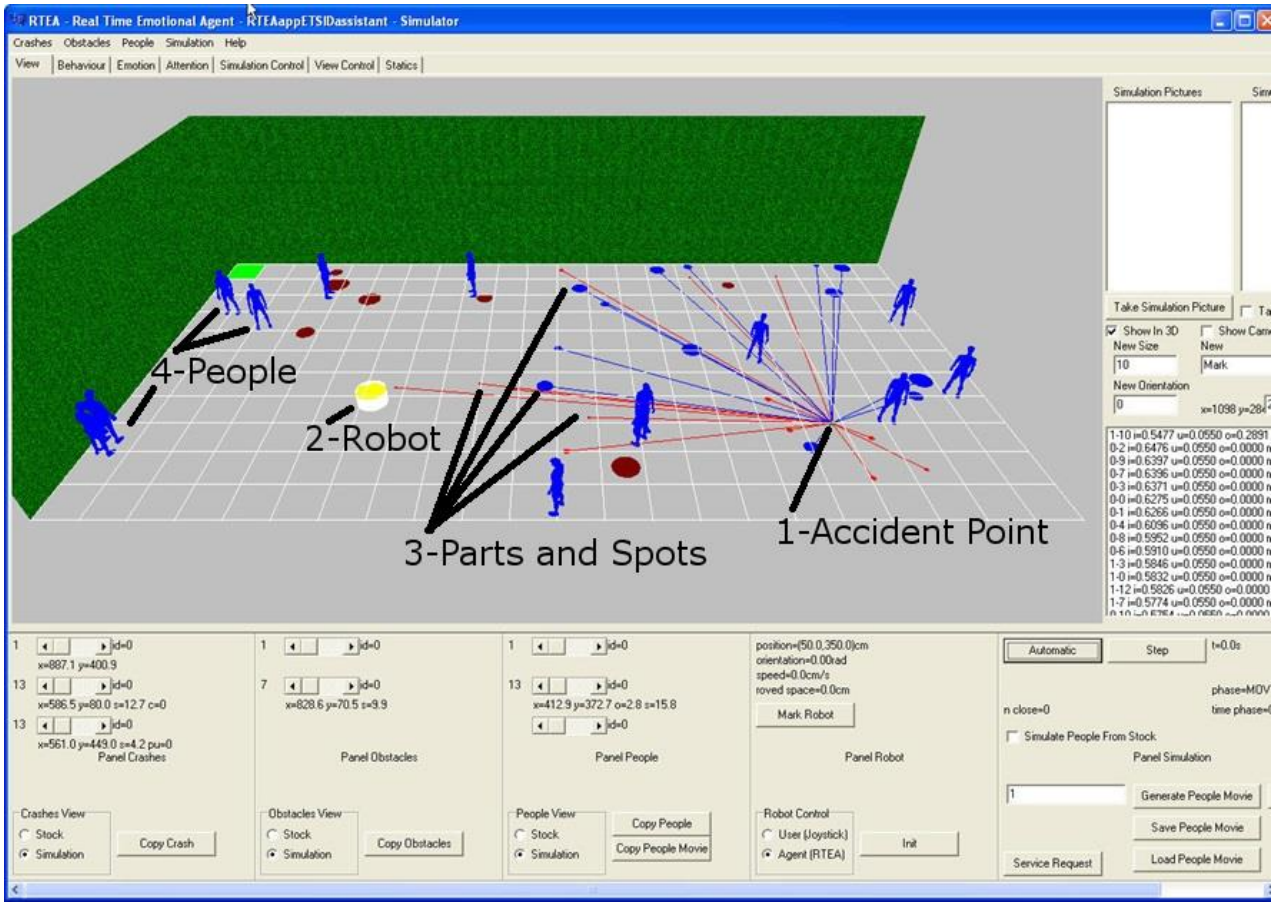

Figura 6: Robot resolviendo un accidente

Para especificar la carga de trabajo de un sistema emocional y para facilitar el estudio del sistema emocional, nos vamos a basar en una de las aplicaciones citadas, esta aplicación consiste en un simulador de entornos (ver la figura 6). El simulador genera una gran cantidad de escenarios diferentes, estos escenarios distintos nos permitirán poner a prueba las capacidades del robot. La figura 6 nos muestra un ejemplo de entre los

distintos escenarios generados, en esta figura podemos ver el resultado de un accidente y al robot intentado solucionar el problema, el accidente ha desperdigado por el entorno de representación fragmentos de objetos y además ha ensuciado ciertos puntos del espacio, para solucionarlo (1) el robot (2) define una serie de subproblemas (3). El objetivo del robot será recoger los fragmentos de los objetos y limpiar las áreas ensuciadas por la colisión. El sistema emocional del robot se encargará de motivar todos los subproblemas, para esto considerará las distintas observaciones para la situación de los subproblemas: la importancia, la probabilidad de conseguirlo, la urgencia y la oportunidad. El sistema de atención del robot utiliza el valor de estas motivaciones para aplicar su política de atención y decidir qué acción debe de realizar a continuación. Además, cabe destacar que el robot no está solo en este escenario, sino que hay personas que caminan por el entorno (4), el simulador define su conducta dependiendo del movimiento y de la zona del accidente donde se encuentren, la idea es que esta gente, y otros obstáculos, puedan interferir en las actividades del robot, así que para garantizar la seguridad y evitar colisiones, el robot debe de realizar las reparaciones ajustándose al entorno, pero el entorno estará cambiando constantemente.

La velocidad a la que se desplaza el robot, el número de objetos en el área de operación y el riesgo de colisión son factores que determinan el ciclo de atención del robot, T_a, este ciclo de atención varia dentro del rango de valores comprendido entre 0.1s y 0.5s. La tabla 1 muestra los valores que puede tomar la velocidad del robot.

TABLA 1

VELOCIDAD DEL ROBOT

Seguro	Normal	Peligroso
0.1m/s	1m/s	2m/s

La tabla 2 muestra los valores que hemos asumido según la complejidad de la aplicación, estos valores se miden en millones de operaciones por ciclo de atención (Mopc). Para obtener estos valores, se han ejecutado en el simulador diferentes aplicaciones, cada aplicación poseía distintas complejidades, esto ha permitido calcular el número de emociones envueltas en cada una de las aplicaciones. Un problema simple requiere la ejecución de unas 0.5 Mopc, mientras que las más complejas requieren cuatro veces más, exactamente 2Mopc.

TABLA 2

COMPLEJIDAD DE LOS PROBLEMAS

Nmero de Operaciones Emocionales por Ciclo de Atencin

Simple	Normal	Complejo
0.5 Mopc	1 Mopc	2 Mopc

El estado emocional del robot representa un ratio, el ratio entre el tiempo empleado para ejecutar el proceso emocional y el ciclo de atención (T_a). En nuestro proyecto se han considerado tres estados emocionales, un estado relajado, un estado normal y uno estresado. En el estado relajado, el robot dedica menos tiempo al proceso emocional y más tiempo a resolver los problemas, pero cuando el robot está estresado ocurre justo lo contrario, el tiempo que dedica al proceso emocional es mayor y el tiempo a resolver los problemas se ve reducido. En el modo relajado, el tiempo para calcular las emociones es menor al 10% del T_a, en el modo normal se asume que este tiempo ronda entre el 10% u el 25%, mientras que si el robot se encuentra estresado, el rango de valores va des del 25% hasta el 40%, estos valores se pueden ver en la tabla 3. Hay que tener en cuenta que una carga de trabajo mayor al 40% no sería aceptable, esto se debe a que si se dedica demasiado tiempo al proceso emocional, la resolución del problema podría sufrir grandes retrasos e incluso podría llegar a pararse, esto se debe a que el robot no poseería el tiempo suficiente para realizar la acción que ha decido, por lo tanto dejaría esta acción a mitad y se volvería a realiza el proceso de motivación, por esta razón, es muy importante que la capacidad de cálculo del robot sea lo más grande posible, de esta forma se evita que pase más del 40% del tiempo pensando, por lo que evitaremos que se quede atascado y no cumplir con sus objetivos nunca.

TABLA 3

ESTADOS EMOCIONALES

Relajado	Normal	Estresado
< 0.1	$[0.1, 0.25]$	$[0.25, 0.4]$

Combinando las diferentes complejidades de los problemas, las distintas velocidades a la que puede ir el robot y los estados emocionales a los que puede estar sometido, se ha elaborado una tabla (ver tabla 4) con el poder computacional requerido, este requisito esta medido en MOPS (ver la sección 1, arquitectura emocional) y nos servirá de plantilla a la hora de evaluar las distintas soluciones propuestas, de esta manera se facilitará la comparación de la propuestas y se desvelará con mayor facilidad los objetivos han sido alcanzados.

TABLA 4
PODER DE PROCESAMIENTO NECESITADO(MOPS)

		Velocidad del robot (m/s)		
Problema	Estado	0.1	1	2
Simple	Estresado	3	5	11
Simple	Normal	4	8	17
Simple	Relajado	13	25	51
Normal	Estresado	6	11	19
Normal	Normal	8	17	33
Normal	Relajado	26	49	99
Complejo	Estresado	9	21	39
Complejo	Normal	17	33	68
Complejo	Relajado	51	99	200

3.2 Hardware empleado

La implementación en paralelo de nuestro sistema ha sido evaluada en distintas plataformas y, como ya se ha mencionado con anterioridad, estas plataformas se pueden encontrar en muchos ordenadores actuales. La versión basada en el uso de procesadores multinúcleo, ha sido ejecutada en un procesador Intel core i7, este procesador tiene cuatro núcleos, la velocidad de estos núcleos es de 2.93 GHz, pero puede alcanzar los 3.6 GHz si se emplea el modo turbo. Para las instrucciones SIMD, las dos versiones, SSE y AVX, han sido utilizadas, como se ha explicado en la sección sobre las instrucciones SIMD, estas van incluidas en el juego de instrucciones del procesador, por lo tanto no necesitan ningún hardware adicional, sin embargo, las instrucciones AVX fueron introducidas posteriormente y por lo tanto, necesitan un procesador más moderno. En lo relativo a las GPUs, se han empleado dos tarjetas gráficas distintas la Nvidia 9800 GTX y la Nvidia 670 GTX, la idea de usar dos tarjetas gráficas distintas es comparar, se busca comparar distintas gamas de NVIDA para tener una mayor visión de su efectividad. La primera tarjeta pertenece a la gama baja de Nvidia, mientras que la segunda pertenece a la gama media alta. La 9800 GTX salió en el 2008, hace más de seis años, mientras que la 670 GTX en el 2012, por lo que es moderna. Para mejorar la comparación, los resultados del algoritmo secuencial y los resultados de la ejecución en dos núcleos se ha añadido a la comparativa, además, vamos a incluir los resultado que se obtuvieron durante las pruebas con FPGAs, pues recordemos que se trataba de un trabajo similar y se ha visto conveniente añadirlos a la comparativa, de esta forma obtenemos más más elementos con los que realizar la comparación. La tabla 5 muestra algunas de las características de las distintas gráficas empleadas.

TABLA 5
Características de las GPUs

	modelos	
Características	9800 GTX	670 GTX
núcleos cuda	128	1344
Frecuencia del procesador (MHz)	675	980
Ancho de banda de la memoria (GB/sec)	70.4	192.2

El sistema basado en emociones ha sido ejecutado bajo diferentes condiciones de entorno, distintos estados emocionales para el robot. El objetivo ha sido realizar una evaluación sobre estas ejecuciones, esta evaluación se ha centrado en el análisis de las herramientas empleadas, midiendo su actuación y comparándola con las otras alternativas. Esta actuación ha sido medida en MOPS, recordemos que en este caso cada operación consiste en el cálculo de seis tangentes hiperbólicas, realizadas a partir de seis observaciones, y su posterior suma creando una emoción.

3.3 Resultados

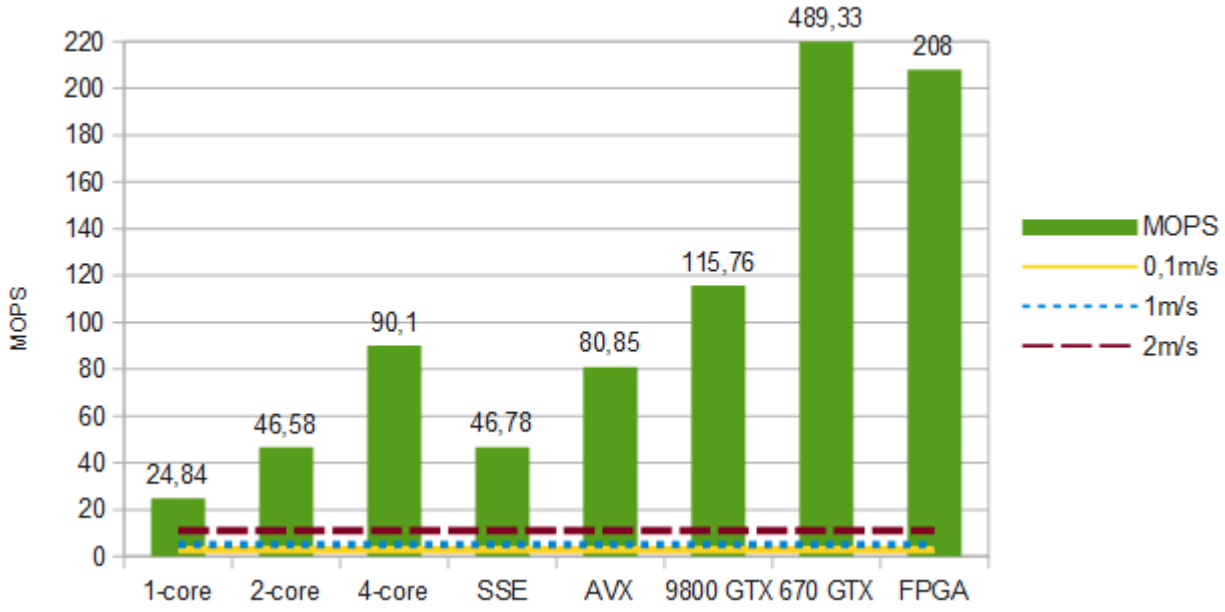

Figura 7: Problema simple con robot estresado

Las figuras de la 7 a la 15 muestran los resultados de las evaluaciones, estas pruebas se han realizado a las distintas implementaciones, se han considerado 3 niveles de complejidad (simple, normal y complejo), tres posibles estados emocionales del robot (estresado, normal y relajado), para tres posibles valores en la velocidad del robot (0.1m/s, 1m/s and 2m/s).

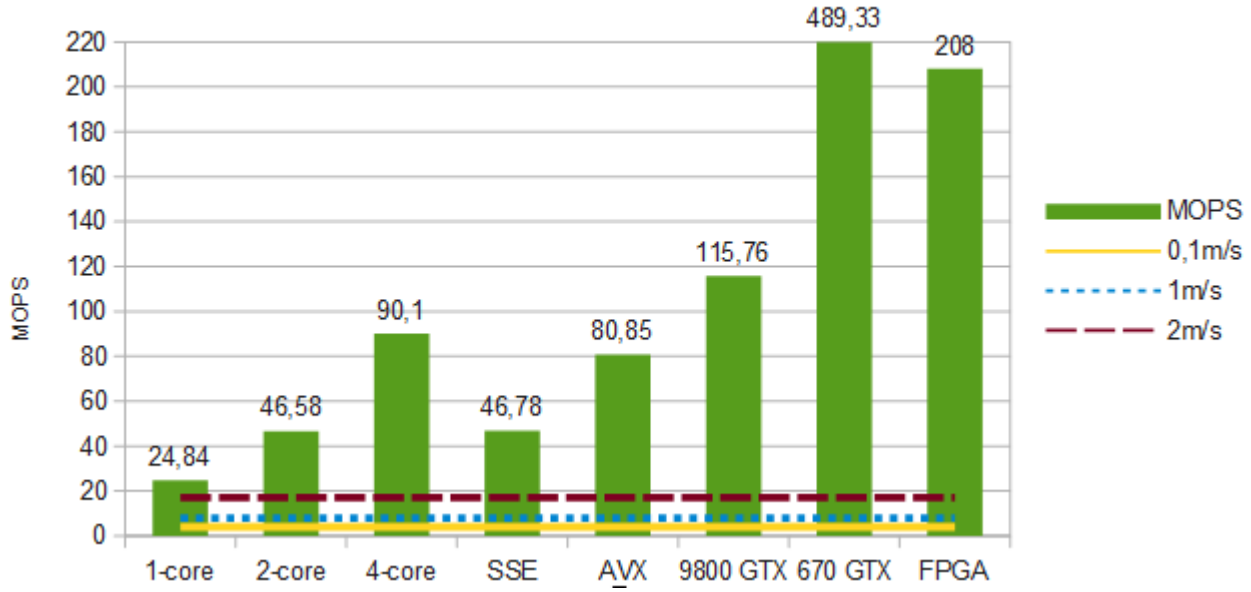

Figura 8: Problema simple con robot normal

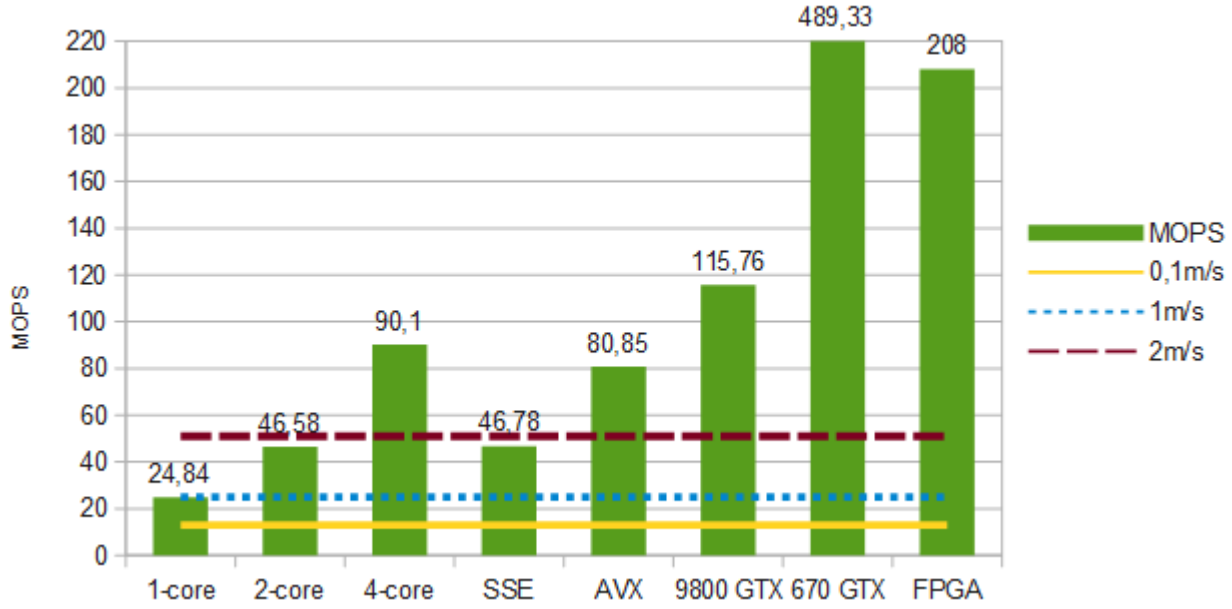

Figura 9: Problema simple con robot relajado

En cada figura podemos ver unas barras, estas barras representan la capacidad computacional en MOPS que cada procesador, juego de instrucciones SIMD, GPU o FPGA pueden conseguir, este valor se ha calculado como la media de varias ejecuciones, éstas han sido realizadas sobre situaciones muy parecidas. Para cada par, complejidad del problema, estado emocional, la velocidad del robot impone un mínimo a la capacidad de computo necesaria, esto se representa como líneas horizontales en las figuras, cada una de las figuras tendrá tres líneas, cada línea representará los valores mínimos para una velocidad determinada, por ejemplo en el caso de un problema simple con el robot relajado (figura 9), si la velocidad es de 2m/s, la capacidad mínima de computación requerida por el sistema emocional es de 51 MOPS, por debajo de esta cifra el robot tendrá problemas para resolver la aplicación, sin embargo, si el robot considera que debe de ir a una velocidad de 0.1m/s, la capacidad de computo requerida es de 13 MOPS

tal y como se muestra en la tabla 4. En general, para el mismo tipo problemas, los requisitos computacionales del procesador aumentan, si aumenta la velocidad con la que se desplaza el robot, esto se debe a que recorre la misma distancia en menos tiempo y por lo tanto, el tiempo de procesamiento disponible es menor. Respecto a la complejidad del problema, cuanto mayor sea ésta, mayor será el requisito computacional para resolverlo. Por último, el estado emocional del robot también afecta a dicho requisito, resulta que si el estado el robot es el de estresado, los requisitos para resolver el problema decrecen, recordemos que en este estado se dispone de más tiempo para el cálculo, pero menos para la realización de la acciones, en caso contrario, cuando el robot se encuentra en un estado relajado, dispone de menos tiempo para el proceso emocional, y por lo tanto, requiere hacer más operaciones por segundo.

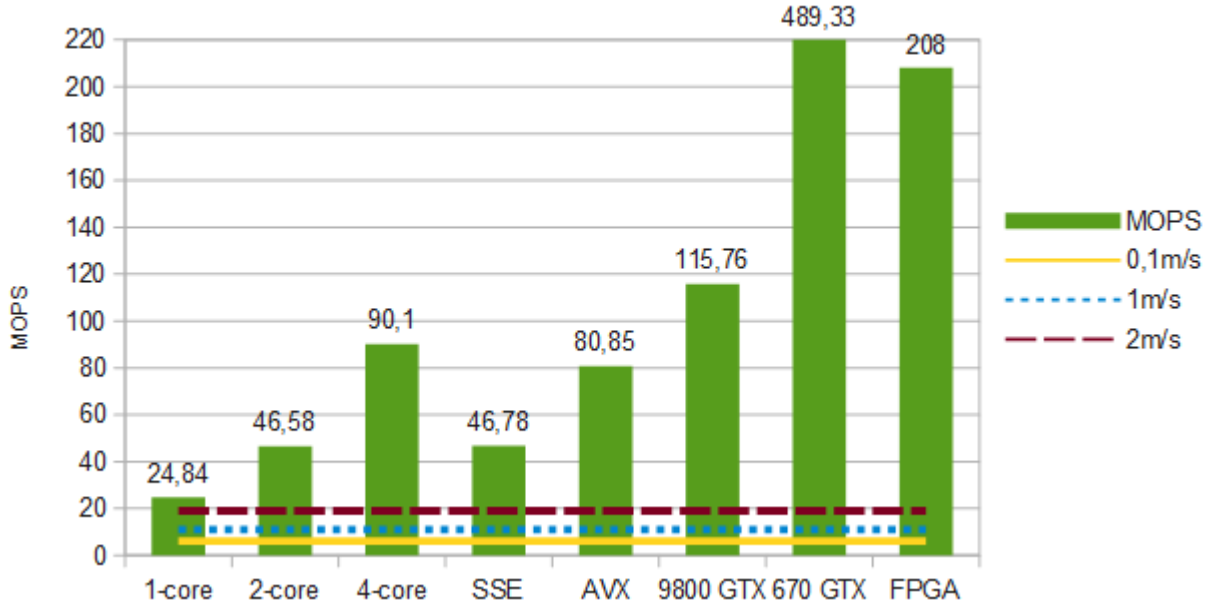

Figura 10: Problema normal con robot estresado

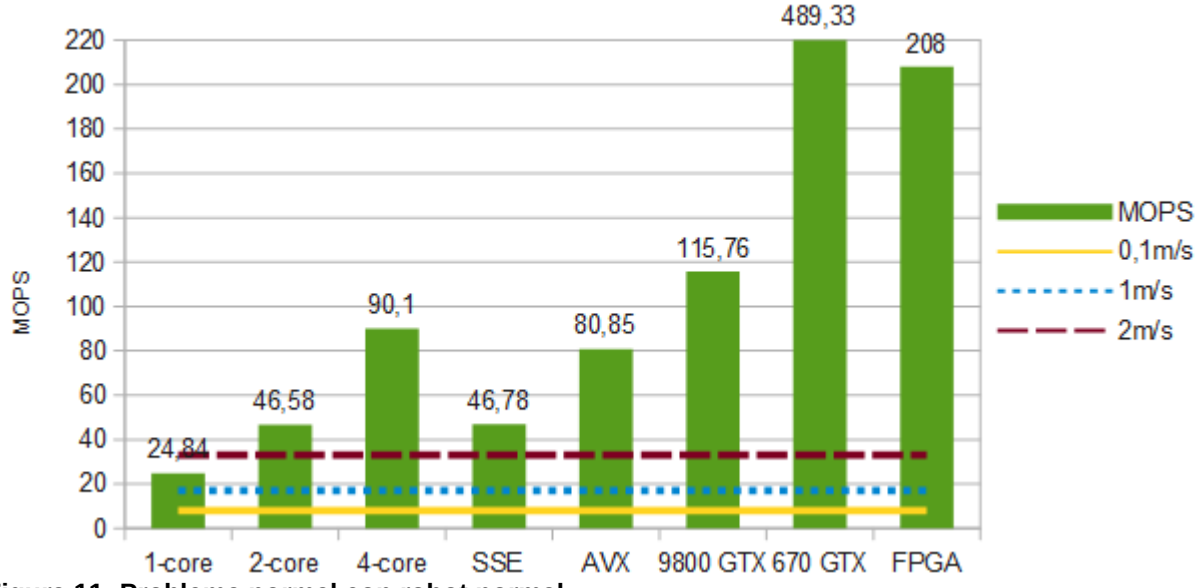

Figura 11: Problema normal con robot normal

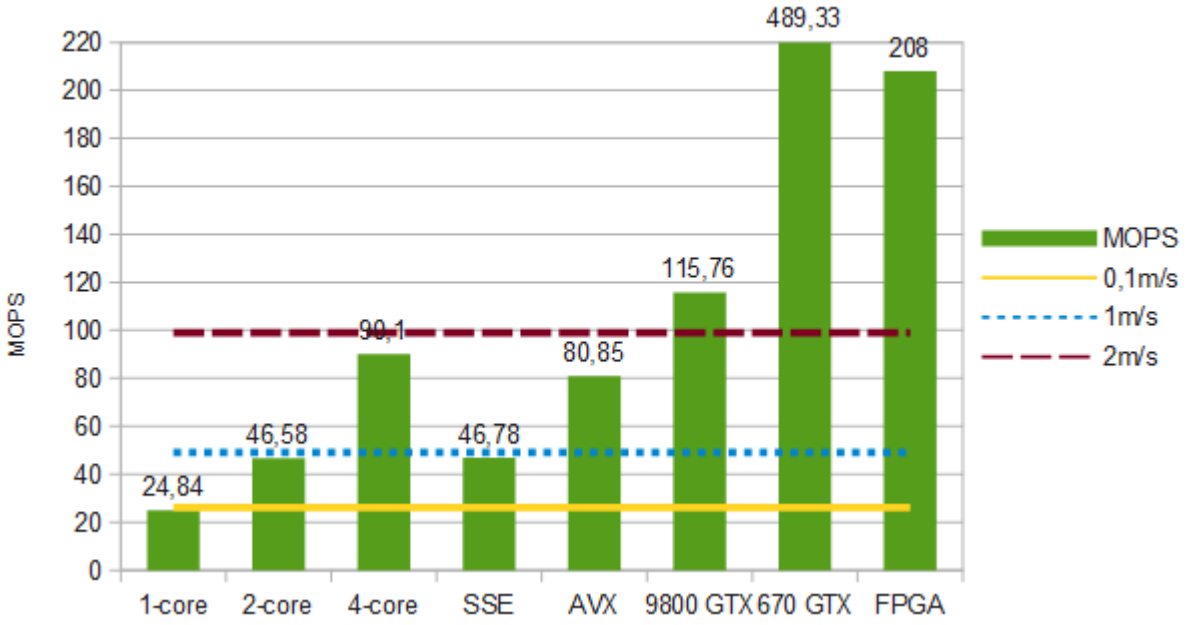

Figura 12: Problema normal con robot relajado

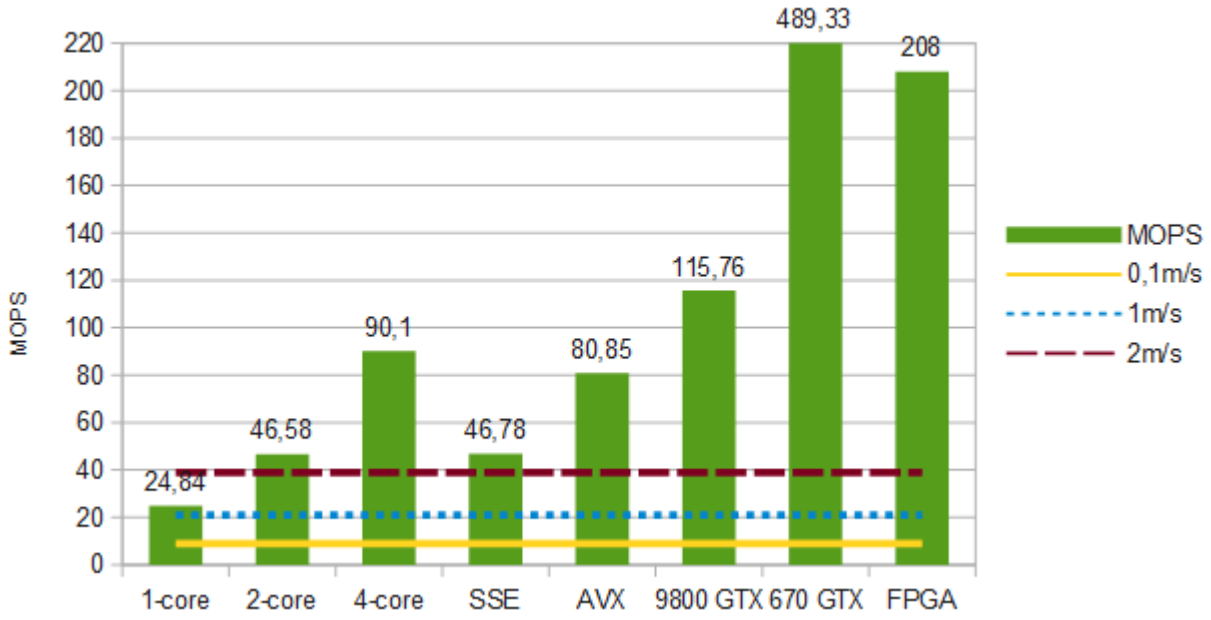

Figura 13: Problema complejo con robot estresado

Ahora que ya sabemos cómo afectan los tres factores principales, pasemos a analizar los datos mostrados en las figuras. Para un problema simple y un robot que se encuentra estresado (ver figura 7), cualquiera de las implementaciones, incluida la secuencial, es capaz de cumplir perfectamente con los requisitos, esto también ocurre en el caso de que el robot se encuentre normal (ver figura 8), en estos casos no importa la velocidad a la que se desplace el robot, no obstante, si el robot pasa a un estado relajado (ver figura 9), la implementación secuencial del sistema emocional no es suficiente para cumplir con todas la velocidades del robot, de hecho si este necesita desplazarse a máxima velocidad, 2m/s, un solo núcleo no sería suficiente y por lo tanto el robot tendría que realizar parones para conseguir terminar el cálculo, obviamente este resultado no es aceptable debido a que

el entorno puede volver a cambiar y la decisión tomada no sea la adecuada para la nueva situación.

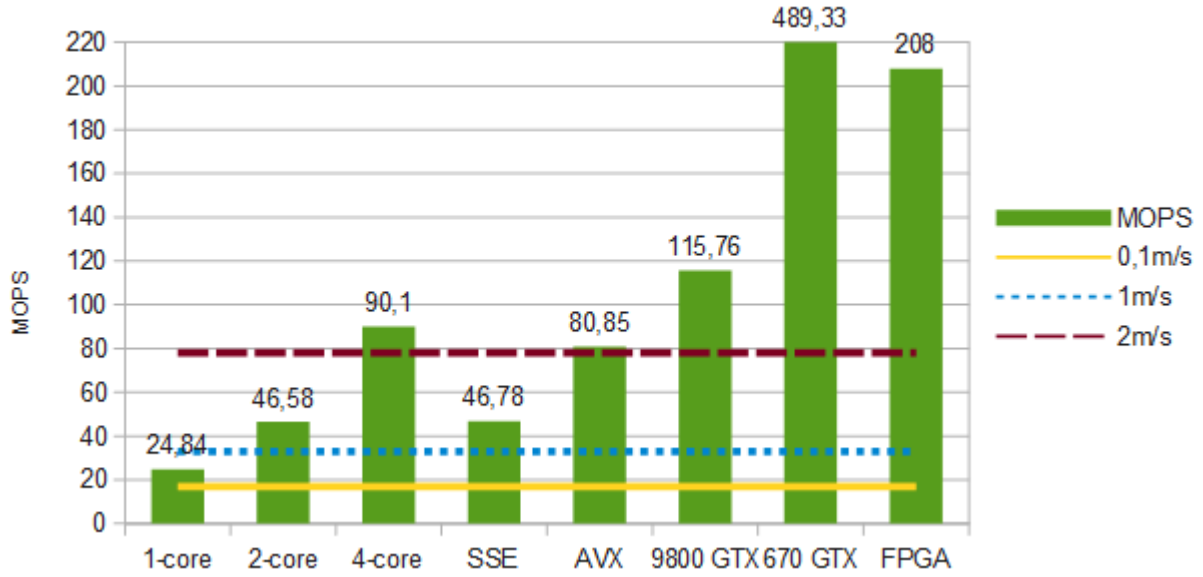

Figura 14: Problema complejo con robot normal

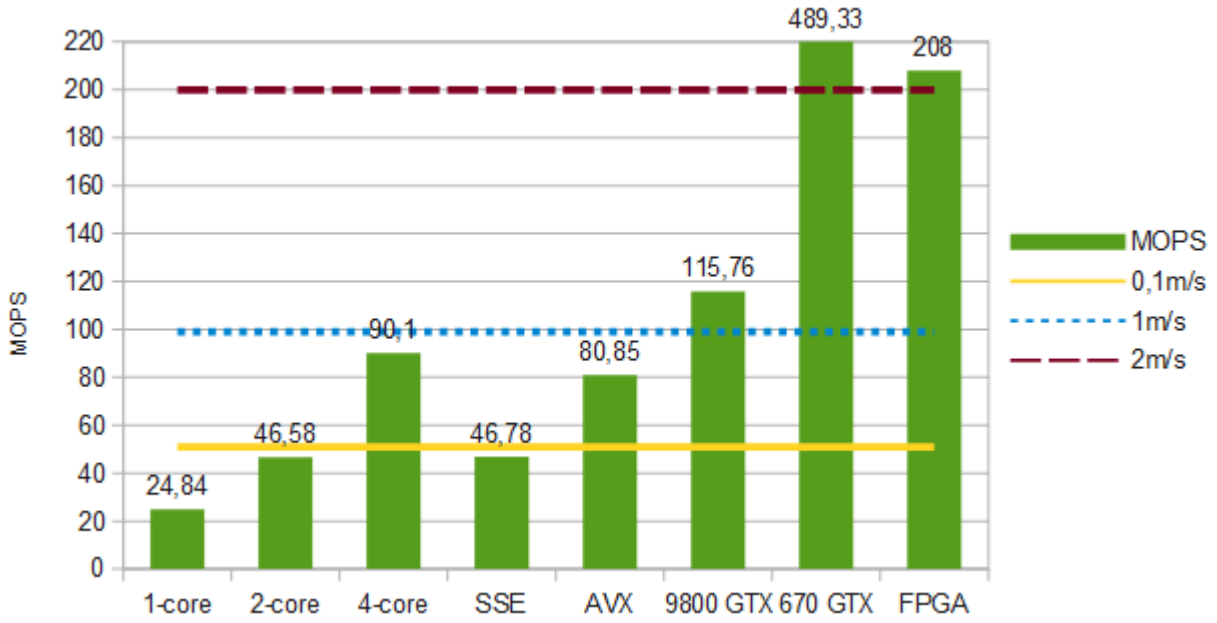

Figura 15: Problema complejo con robot relajado

Para un problema de dificultad normal con un robot estresado, la implementación secuencial funciona correctamente (ver figura 10), por lo tanto, el resto de implementaciones lo consiguen, sin importar a la velocidad a la que se desplace el robot. Si el estado del robot se relaja un poco y alcanza el valor normal (ver figura 11), la versión secuencial no es suficiente, si el robot va a máxima velocidad no realiza los cálculos necesarios a tiempo, por otro lado, cualquier otra implementación alcanza el mínimo computacional requerido. Estos resultados cambian cuando, para la misma complejidad del problema, el robot se relaja (ver figura 12) pues solo las implementaciones basadas en GPUs y en la FPGA Statrix IV pueden alcanzar los requisitos de cálculo cuando el robot circula a su mayor velocidad. Cuando el robot circula

a 1 m/s tanto el procesador de 4 núcleos como la implementación en AVX alcanzan los objetivos, al igual que todas las GPUs y la FPGA. Cuando el robot circula a la velocidad mínima, solo la ejecución secuencial es incapaz de cumplir los objetivos, aunque sí que es cierto que 24.84 MOPS es una media y que en algunas ejecuciones se han obtenido 25 MOPS, en la mayoría de casos no se ha alcanzado esa cantidad y por lo tanto los resultados no son aceptables.

En lo relativo a los problemas de mayor complejidad, si el robot se encuentra estresado (ver figura 13), la implementación secuencial puede completarlos sí el robot no va a máxima velocidad, es decir, si se desplaza a 0.1 m/s o a 1 m/s, cualquier implementación tomada en cuenta puede conseguirlo, sin embargo, a máxima velocidad la versión secuencial no es suficiente, pero en lo referente al resto de implementaciones no tienen mayores problemas. Cuando el robot se encuentra en estado normal (ver figura 14), ninguna implementación en paralelo tiene problemas para alcanzar los objetivos, siempre y cuando el robot no vaya máxima velocidad. La versión secuencial solo sirve si el robot circula a mínima velocidad, para el resto de casos no es suficiente. Cuando nos fijamos en los requisitos computacionales del robot desplazándose a máxima velocidad, solo cuatro implementaciones (las dos GPUs, la FPGA y los 4 núcleos) cumplen con los requisitos limpiamente. En lo referente al set de instrucciones AVX, podría tener algunas dificultades, en la mayoría de caso no habría ninguna complicación, sin embargo podrían existir ocasiones en las que no fuese suficiente, hay que tener en cuenta que el parón no sería muy importante. Cuando nos encontramos en un problema complejo, con la necesidad de que el robot vaya relajado (ver figura 15), la implementación secuencial no sirve, pues ni siquiera puede cumplir con los requisitos cuando el robot va a 0.1m/s, pero esto no solo le ocurre a la secuencial, sino que ocurre lo mismo con las implementaciones que de dos núcleos y en las que se usa el set SSE. Cabe destacar que el resto de implementaciones no tienen ningún problema. Cuando el robot decide ir a 1m/s, las cosas se vuelven más complicadas y solo tres de las alternativas cumplen los requisitos, lo que demuestra el alto nivel de requisitos del sistema basado en emociones. Las implementaciones que superan los requisitos son las dos GPUs y la FPGA, que son justamente las propuestas que mayor potencia de paralelismo poseen. Cuando el robot va a 2m/s, siendo la situación con mayor grado de restricción, solo dos versiones pueden con los requisitos, la GPU de la tarjeta gráfica Nvidia 670 GTX y la FPGA Statrix IV, el resto de versiones ni siquiera se acercan al objetivo, por lo que ni si quiera se debería considerar su empleo bajo estas condiciones, no obstante, si se juntasen las capacidades de los ordenadores multinúcleo y las instrucciones AVX se podrían alcanzar resultados muy prometedores.

Como hemos visto, solo la Nvidia 670 GTX y la FPGA Statrix IV pueden solucionar todos los problemas, incluidos aquellos con los requisitos más altos. Sin embargo, el caso de la tarjeta gráfica 670 GTX es una mejor elección, esto no se debe solo a que supera los resultados de la FPGA Statrix IV, sino que además no es un hardware tan específico que haya que comprar adrede, las tarjetas gráficas están incluidas en todos los ordenadores de sobremesa, además cabe destacar que no son tan caras como la Statrix IV.

La Nvidia 9800 GTX se sitúan como la tercera solución más prometedora, esta propuesta puede ser muy útil si el robot ha de solucionar problemas con menores requisitos, debido a que pertenece a la gama baja de Nvidia la inmensa mayoría de gráficas actuales de esta marca deberían de dar buenos para problemas menos exigentes. El procesador de cuatro y el set de instrucciones AVX, han obtenido resultados muy similares, y por supuesto se encuentran en la mayoría de ordenadores de sobremesa actuales, y como hemos visto solucionan la gran mayoría de las situaciones propuestas, aunque existe problemas complejos que superan sus capacidades(por ejemplo: un problema complejo con un robot en un estado normal). El set de instrucciones SSE y la implementación para dos núcleos han obtenido resultados iguales prácticamente, estas propuestas no son suficiente para solucionar problemas cuando el robot se encuentra en un estado relajado, y tampoco cuando este necesita ir a velocidades altas, por lo que si no se tiene acceso a mejores herramientas, éstas se pueden emplear en aplicaciones más sencillas que no requieran tanta capacidad de computo.

Un factor a destacar es que la mayoría de las propuestas descritas a lo largo de este trabajo, se pueden encontrar en ordenadores comunes actuales, dado que se consideran piezas fundamentales de estos ordenadores y todos ellos vienen con procesadores de varios núcleos o con gráficas que poseen GPUs listas para realizar este tipo de cálculos, aparte de que los procesadores actuales tienen acceso al set de instrucciones SSE y aquellos con una arquitectura posterior a la arquitectura Sandy Bridge en el caso de los procesadores INTEL y para los de AMD la arquitectura Bulldozer, podrán emplear el juego de instrucciones AVX.

4. CONCLUSIONES

La arquitectura basada en emociones es una de las soluciones más prometedoras para implementar los robots del futuro. Sin embargo, esta arquitectura tiene unos requisitos computacionales muy altos. Por esto, la capacidad de cómputo que posee un solo núcleo es insuficiente y necesitamos buscar otras alternativas. Para solucionar este problema, se han explotado las capacidades paralelas inherentes a esta arquitectura. Para ello se han realizado distintas propuestas: los procesadores multinúcleo, las GPUs y los juegos de instrucciones SIMD. Para comprobar su efectividad, se ha empleado una aplicación para el robot, la cual emula distintos entornos, distintos estados emocionales del robot y distintas velocidades de desplazamiento, que nos permiten medir la efectividad de estas propuestas.

En el caso de las GPUs, hemos usado dos tarjetas gráficas distintas. Ambas son de la marca Nvidia, pero pertenecen a distintas gamas, estás tarjetas son la 9800 GTX y la 670 GTX. Los resultados se han comparado con los obtenidos por un procesador de cuatro núcleos (Intel i7 CPU 870 2.93GHz), para el que se han hecho pruebas con dos y cuatro núcleos. Otras herramientas utilizadas en la comparación son los juegos de instrucciones SSE y AVX. Por último, hemos añadido en la comparación los resultados obtenidos utilizando FPGAs (Statrix IV) y la implementación, la

implementación que mejores resultados ha dado ha sido la que emplea GPUs. Aunque la Nvidia 9800 GTX no ha podido resolver todos los problemas, hemos de tener en cuenta que forma parte de la gama baja de las tarjetas de Nvidia y que cuando empleamos una gama media alta, como la 670 GTX, los resultados han superado sin grandes problemas todos los problemas propuestos, incluso han superado al resto de implementaciones. La Statrix IV ha resuelto todos los problemas igualmente, sin embargo, su alto coste hace que otras propuestas, como los procesadores multinúcleo, puedan ser más adecuadas para el desarrollo del sistema emocional, pues los resultados de los multinúcleo son prometedores y éstos son muy comunes en los procesadores. Las pruebas se han realizado sobre cuatro y dos núcleos, sin embargo podemos encontrar, al alcance del público general, procesadores con hasta 6 núcleos que podrían obtener mejores resultados. En el caso de instrucciones SIMD, pese a obtener resultados un poco peores que los procesadores multinúcleo, son una posible alternativa a estos, pues hemos de destacar que no añaden coste alguno al de procesador y que la mayoría de ellos pueden utilizarlas sin problemas. Un punto muy importante a tener en cuenta es que las capacidades paralelas de los procesadores multinúcleo y las instrucciones SIMD pueden ser combinadas y por lo tanto, podemos obtener incluso mejores resultados que los aquí mostrados. No obstante este campo es relativamente nuevo y algunas librerías como la OMP, usada en este trabajo, lo han añadido recientemente (Open MP lo ha añadido en su versión 4.0), así que sería conveniente tenerlo en cuenta para trabajos futuros.

Los objetivos buscados en este trabajo han sido completados, hemos encontrado distintas implementaciones para el sistema emocional, que además se pueden encontrar en la mayoría de ordenadores actuales y que no son tan costosas como las soluciones que se habían propuesto anteriormente. Por lo que sería muy interesante seguir desarrollando aplicaciones en esta línea, al menos en lo referente a la implementación de robots, pues a priori parece ser una línea consistente para el desarrollo de futuras aplicaciones.

Fruto del trabajo realizado en este TFG, se ha publicado un artículo en el 6th international Workshop on Multicore and Multithreaded Architectures an Algorithms, celebrado junto a la 16th IEE International Conference on High Performance Computing and Comunications (HPCC 2014) [24]. También hemos preparado un segundo artículo que enviaremos a una revista especializada, para una revisión y posible aceptación.

BIBLIOGRAFÍA

[1] M. Malfaz and M.A. Salichs (2010). Using MUDs as an experimental platform for testing a decision making system for self-motivated autonomous agents. Artificial Intelligence and Simulation of Behaviour Journal, Vol. 2, No. 1, pp.21-44.

[2] L. Damiano and L. Cañamero(2010). Constructing Emotions.

Pistemological groundings and applications in robotics for a synthetic approach to emotions. AI-Inspired Biology (AIIB) Symposium, Leicester, UK.

[3] N. Hawes, J. Wyatt and A. Sloman(2009). Exploring design space for an integrated intelligent system. Knowledge-Based Systems, Vol. 2m2, No.7, pp. 509-515.

[4] A. Sloman(2009). Some Requirements for Human-Like Robots: Why The Recent Over-Emphasis on Embodiment Has Held Up Progress. Creating Brain-Like Intelligence, 248-277.

[5] H. Moravec(2009). Rise of the Robots{The Future of Artificial Intelligence. Scientic American.

[6] L. Moshkina, R. C. Arkin (2009) Beyond Humanoid Emotions: Incorporating Traits, Attitudes and Moods. IEEE International Conference on Robotics and Automation.

[7] iRobot industrial robots website: http://www.irobot.com/gi/ground/

[8] C. P. Lee-Johnson and Dale A. Carnegie (2010). Mobile Robot Navigation Modulated by Artificial Emotions. IEEE Transactions On Systems, Man, And Cybernetics PART B: CYBERNETICS, VOL. 40, NO. 2.

[9] C. Domínguez, H. Hassan, A. Crespo (2009). Real-time Emotional Agent Architecture Application on Service Mobile Robot Control The 2009 International Conference on Artificial Intelligence. Las Vegas, Nevada, USA.

[10] E. Daglarli, H. Temeltas, M. Yesilogly (2009). Behavioral task Processing for cognitive robots using artificial emotions Neurocomputing 2009.

[11] R. Ventura, C. Pinto-Ferreira (2009). Responding efficiently to Relevant stimuli using an emotion-based agent architecture Neurocomputing 2009.

[12] Altera FPGA devices Stratix III and Stratix IV, http://www.altera.com/devices/fpga/stratix-fpgas/stratix-iii/st3-index.jsp

http://www.altera.com/devices/fpga/stratix-fpgas/stratixiv/ stxivindex. Jsp

[13] Intel Processor Core i5 660, http://ark.intel.com/products/43550/Intel-Core-i5-660-Processor-(4MCache- 3 33-GHz)

[14] San Jose Convention Center. Introduction to Cuda C (2010), http://www.nvidia.com/content/GTC-2010/pdfs/2131 GTC2010.pdf

[15] Nvidia official website http://www.nvidia.es/object/cuda-parallel-computing-es.html

[16] Nvidia developer zone (2012), http://devblogs.nvidia.com/parallelforall/how-overlap-data-transferscuda-cc/

[17] Dr. Justin Luitjens and Dr. Steven Rennich, CUDA Warps and Occupancy(2011) http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf

[18] Ramses van Zon (septiembre 2013), OpenMP 4 - What's New?, http://wiki.scinethpc.ca/wiki/images/9/9b/Ds-openmp.pdf

[19] Anoop Madhusood (diciembre, 2013), Enabling SIMD in program using OpenMP4.0, https://software.intel.com/en-us/articles/enabling-simd-in-program-using-openmp40

[20] J. Stokes (2000),http://arstechnica.com/features/2000/03/simd/2/

[21] C. Lomont, Introduction to Intel Advanced Vector Extensions, http://software.intel.com/en-us/articles/introduction-to-intel-advancedvector-extensions

[22] David A. Patterson and John L. Hennessey, Computer Organization and Design: the Hardware/Software Interface, 2nd Edition, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1998, p.751

[23] M. Shigeo (2010). http://homepage1.nifty.com/herumi/soft/fmath.html

[24] F. Almenar, C. Domínguez, J. Martínez, H. Hassan, P. López, Embedded Multicore Processors and SIMD Instructions for Emotional-based Mobile Robotic Agents. Proceedings of the 16th International Conference on High Performance Computing and Communications. IEEE Computer Society, ISBN nº978-1-4799-6123-8

ANEXO: CÓDIGOS COMPLETOS

Código secuencial

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define e 2.718281828459

int main( int argc, char *argv[] ) {
	float aux, time, *a, *fci, *m;
	int N, i, j, count,number;
	FILE* data;
	clock_t t_ini, t_fin;

	if(argc != 2){
		printf("Wrong usage, expected: N");
	}

	j = 0;

	number=30;

	N = atoi(argv[1]);

	fci = (float*)malloc( number*N * sizeof(float) );
	a = (float*)malloc( number*N * sizeof(float) );
	m = (float*)malloc( number*(N/6) * sizeof(float) );

	//initializing values on the CPU
	data=fopen("data.txt","r");

	fscanf(data,"%f",&aux);

	for(i=0;i<N;i++){
		for(count=0; count<number;count++){
			a[j]=aux;
			j++;
		}
		fscanf(data,"%f",&aux);
	}
	fclose(data);

	N=number*N;

	printf("Data: %d\n", N);

	t_ini = clock();
	for(i=0;i<N;i++){
		fci[i] = tanh (a[i]);
	}

	int acum = 0;

	for(i=0;i<N/6;i++){
```

```
                for(j=0;j<6;j++){
                        acum = fci[6*i+j];
                }
                m[i] = acum;
        }

        t_fin = clock();

        time = (float)(t_fin-t_ini);

        printf("Tiempo en CPU:%f ms\n", time/(double)CLOCKS_PER_SEC*1000);

        free(a);
        free(fci);
        return 0;
}
```

Código multinúcleo

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define e 2.718281828459

int main( int argc, char *argv[] ) {
        float aux, *a, *fci, *m;
        int N, i, j, count, number, tt;
        double t1, t2, acum;
        FILE* data;

        if(argc != 2){
                printf("Wrong usage, expected: N");
        }

        j = 0;
        number = 30;

        N = atoi(argv[1]);

        //Allocate memory on CPU
        fci = (float*)malloc( number*N * sizeof(float) );
        a = (float*)malloc( number*N * sizeof(float) );
        m = (float*)malloc( number*(N/6) * sizeof(float) );

        //initializing values on the CPU
        data=fopen("data.txt","r");

        fscanf(data,"%f",&aux);

        for(i=0;i<N;i++){
                for(count=0; count<number;count++){
                        a[j]=aux;
                        j++;
                }
```

```
            fscanf(data,"%f",&aux);
        }
        fclose(data);

        N = number * N;
        acum = 0;
        printf("Data: %d\n", N);

        t1 = omp_get_wtime();

        //hyperbolic tangent calculations
        #pragma omp parallel for private(j, acum)
        for(i=0;i<N/6;i++){

            acum = 0;
            for(j=0; j<6; j++){

                fci[6*i+j] = (exp(2*a[6*i+j])-1)/(exp(2*a[6*i+j])+1);
                acum += fci[6*i+j];
            }
            m[i] = acum;
        }
        t2 = omp_get_wtime() - t1;

        printf("Tiempo OMP:%f ms\n", t2*1000);

        //free memory
        free(a);
        free(fci);
        return 0;
}
```

Código GPU

```
#include <cuda.h>
#include <math.h>
#include <stdio.h>

__global__ void hyperbolicTangent (float *dev_app, int *dev_N, float *dev_fci) {

        int tid = threadIdx.x + blockIdx.x * blockDim.x;

        //Hyperbolic tangent
        while(tid < *dev_N){
            dev_fci[tid] = tanh(dev_app[tid]);
            tid += blockDim.x * gridDim.x;
        }
}

__global__ void reduction (float *dev_fci, float *dev_emotions, int *dev_N){

        int tid = 6*threadIdx.x + blockIdx.x * blockDim.x;
        int tidemotion = threadIdx.x + blockIdx.x * blockDim.x;

        int cont;
```

```
	while(tid < *dev_N){
		cont = 0;
		dev_emotions[tidemotion] = 0;

		while(cont <= 5){
			dev_emotions[tidemotion] += dev_fci[tid+cont];
			cont ++;
		}
		tid += 6 * blockDim.x * gridDim.x;
		tidemotion += blockDim.x * gridDim.x;
	}
}

int main( int argc, char *argv[] ) {
	float *a, *fci, *m, aux, time;
	float *dev_a, *dev_fci, *dev_m;
	int N, *dev_N, i, j, count, number;
	int blocks = 128;
	int threads = 1024;
	FILE* data;
	cudaEvent_t start, stop;


	if(argc != 2){
		printf("Wrong usage, expected: N");
	}

	j = 0;
	number=30;
	N = atoi(argv[1]);

	//allocate memory on the CPU
	fci = (float*)malloc( number*N * sizeof(float) );
	m = (float*)malloc( ( number*N / 6 ) * sizeof(float) );
	a = (float*)malloc( number*N * sizeof(float) );

	//creating time events
	cudaEventCreate(&start);
	cudaEventCreate(&stop);

	//Reading values
	data = fopen("data.txt","r");
	fscanf(data,"%f",&aux);
	for(i=0;i<N;i++){
		for(count=0; count<number;count++){
			a [j]=aux;
			j++;
		}
		fscanf(data,"%f",&aux);
	}
	fclose(data);

	N=number*N;
	printf("Data: %d\n", N);

	//allocate memory on the GPU
	cudaMalloc( (void**)&dev_a, N * sizeof(float) );
	cudaMalloc( (void**)&dev_fci, N * sizeof(float) );
```

```
	cudaMalloc( (void**)&dev_m, ( N / 6 ) * sizeof(float) );
	cudaMalloc( (void**)&dev_N, sizeof(int) );

	//copy values from CPU to GPU
	cudaMemcpy( dev_a, a, N * sizeof(float), cudaMemcpyHostToDevice );
	cudaMemcpy( dev_N, &N, sizeof(int), cudaMemcpyHostToDevice );


	cudaEventRecord(start, 0);

	hyperbolicTangent<<<blocks, threads>>>(dev_a, dev_n, dev_fci);

	reduction<<<blocks, threads>>>( dev_fci, dev_m, dev_n);

	cudaEventRecord(stop, 0);
	cudaEventSynchronize(stop);

	cudaEventElapsedTime(&time, start, stop);

	cudaEventDestroy(start);
	cudaEventDestroy(stop);

	printf("Tiempo parcial:%f ms\n", time);

	//Copy emotions back from the GPU to the CPU
	cudaMemcpy( m, dev_m, ( N / 6 ) * sizeof(float), cudaMemcpyDeviceToHost );

	//Free GPU memory
	cudaFree( dev_a );
	cudaFree( dev_fci );
	cudaFree( dev_m);
	cudaFree( dev_N );

	//Free CPU memory
	free ( fci );
	free ( m );
	free ( a );


	return 0;
}
```

Código SSE

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <emmintrin.h>
#include "fmath.hpp"
#include <mmintrin.h>

#define e 2.718281828459

int main( int argc, char *argv[] ) {
	float aux, time, *a, *m, *fci, *ones, *twos, *ceros, *masks, *aux2;
```

```
int N, i, j, count, number;
FILE* data;
clock_t t_ini, t_fin;
__m128 div, one, two, cero, ptrPos, ptrNeg, ptr, ptr2, ptrEm;

if(argc != 2){
        printf("Wrong usage, expected: N");
}

j = 0;
number=30;
N = atoi(argv[1]);

// Allocating memory
posix_memalign((void**)&fci, 16,  number * N * sizeof(float));
posix_memalign((void**)&a, 16,  number * N * sizeof(float));
posix_memalign((void**)&m, 16,  number * (N/6.0) * sizeof(float));
posix_memalign((void**)&ceros, 16,  4 * sizeof(float));
posix_memalign((void**)&ones, 16,  4 * sizeof(float));
posix_memalign((void**)&twos, 16,  4 * sizeof(float));
posix_memalign((void**)&masks, 16,  4 * sizeof(float));
posix_memalign((void**)&aux2, 16,  4 * sizeof(float));

//initializing values on the CPU
data=fopen("data.txt","r");
fscanf(data,"%f",&aux);
for(i=0;i<N;i++){
        for(count=0; count<number;count++){
                a[j]= aux;
                j++;
        }
        fscanf(data,"%f",&aux);
}
fclose(data);

N=number*N;
printf("Data: %d\n", N);

ceros[0] = 0.0;
ceros[1] = 0.0;
ceros[2] = 0.0;
ceros[3] = 0.0;

ones[0]=1.0;
ones[1]=1.0;
ones[2]=1.0;
ones[3]=1.0;

twos[0]=2.0;
twos[1]=2.0;
twos[2]=2.0;
twos[3]=2.0;

//converting ones and twos into __m128
cero = _mm_load_ps( &ceros[0] );
one = _mm_load_ps( &ones[0] );
two = _mm_load_ps( &twos[0] );
```

```
	//hyperbolic tangent with sse 2.0
	t_ini = clock();

	for(i=0; i<N/4; ++i){
		ptr = _mm_load_ps( &a[4*i] );
		ptr = _mm_mul_ps(two, ptr);
		ptr = fmath::exp_ps(ptr);
		ptrNeg = _mm_sub_ps(ptr, one);
		ptrPos = _mm_add_ps(ptr, one);
		div = _mm_div_ps(ptrNeg, ptrPos);
		_mm_store_ps(fci+4*i, div);
	}

	for(i=0; i<N/6; ++i){
		// loading values
		ptr = _mm_load_ps( &fci[4*i] );
		ptr2 = _mm_load_ps( &fci[4*(i+1)] );

		//mask
		ptr2 = _mm_movelh_ps( ptr2, cero);

		//reduction
		ptrEm = _mm_hadd_ps ( ptr, ptr2);
		ptrEm = _mm_hadd_ps ( ptrEm , cero);
		ptrEm = _mm_hadd_ps ( ptrEm , cero);

		_mm_store_ps(&aux2[0], ptrEm);

		m[i] = aux2[0];
	}
	t_fin = clock();

	time = (float)(t_fin-t_ini);
	printf("Tiempo en CPU:%f ms\n", time/(double)CLOCKS_PER_SEC*1000);

	//Free memory
	free( fci );
	free( a );
	free( ones );
	free( twos );
	return 0;
}
```

Código AVX

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <emmintrin.h>//needed?
#include <immintrin.h>
#include "avx_mathfun.h"
#include <mmintrin.h>

int main( int argc, char *argv[] ) {
	float aux, time, *a, *fci, *ones, *twos, *pos, *neg, auxVect;
```

```
int N, i, j, count, number;
FILE* data;
clock_t t_ini, t_fin;
__m256 div, one, two, ptrPos, ptrNeg, ptr;

if(argc != 2){
	printf("Wrong usage, expected: N");
}

j = 0;
number=70;
N = atoi(argv[1]);

// Allocating memory
posix_memalign((void**)&fci, 32,  number * N * sizeof(float));
posix_memalign((void**)&a, 32,  number * N * sizeof(float));
posix_memalign((void**)&neg, 32,  number * N * sizeof(float));
posix_memalign((void**)&pos, 32,  number * N * sizeof(float));
posix_memalign((void**)&ones, 32,  8 * sizeof(float));
posix_memalign((void**)&twos, 32,  8 * sizeof(float));
posix_memalign((void**)&auxVect, 32,  8 * sizeof(float));

//initializing values on the CPU
data=fopen("data.txt","r");
fscanf(data,"%f",&aux);
for(i=0;i<N;i++){
	for(count=0; count<number;count++){
		a[j]= aux;
		j++;
	}
	fscanf(data,"%f",&aux);
}
fclose(data);

N=number*N;
printf("Data: %d\n", N);

ones[0]=1.0;
ones[1]=1.0;
ones[2]=1.0;
ones[3]=1.0;
ones[4]=1.0;
ones[5]=1.0;
ones[6]=1.0;
ones[7]=1.0;

twos[0]=2.0;
twos[1]=2.0;
twos[2]=2.0;
twos[3]=2.0;
twos[4]=2.0;
twos[5]=2.0;
twos[6]=2.0;
twos[7]=2.0;

//converting ones and twos into __m256
one = _mm256_load_ps(&ones[0]);
two = _mm256_load_ps( &twos[0] );
```

```
	//hyperbolic tangent with avx
	t_ini = clock();
	for(i=0; i<N/8; ++i){
		ptr = _mm256_load_ps( &a[8*i] );
		ptr = _mm256_mul_ps(two, ptr);
		ptr = exp256_ps(ptr);
		ptrNeg = _mm256_sub_ps(ptr, one);
		ptrPos = _mm256_add_ps(ptr, one);
		div = _mm256_div_ps(ptrNeg, ptrPos);
		_mm256_store_ps(fci+8*i, div);
	}

	auxVect[6] = 0;
	auxVect[7] = 0;


	for (i=0; i<n/6; ++i) {

		for( j=0; j<6; j++){

			auxVect [j] = fci[i+j];
		}

		ptr = _mm256_load_ps(&auxVect);

		ptrEm = _mm256_hadd_ps(ptr,zero);
		ptrEm =_mm256_hadd_ps(ptrEm,zero);
		ptrEm =_mm256_hadd_ps(ptrEm,zero);

		_mm256_store_ps(&m[i], ptrEm);
	 }

	t_fin = clock();

	time = (float)(t_fin-t_ini);
	printf("Tiempo en CPU:%f ms\n", time/(double)CLOCKS_PER_SEC*1000);

	//Free memory
	free( fci );
	free( a );
	free( pos );
	free( neg );
	free( ones );
	free( twos );
	return 0;
}
```