Abstract—
Over the last few years, we have witnessed the proliferation of GPU devices on HPC environments. Manufacturers produce new versions of their devices every few years, though, posing a new problem for scientists and engineers using their technology: is it worth the time and effort spent optimizing the codes for the current version? Or is it better to wait until a new architecture appears? In this paper, we present a comparison of various CUDA versions, in order to compare their architectures, and optimize codes for each version.

This work would require a tremendous coding effort if done manually. However, using fast prototyping tools, like mtic, this is an effortless process. Applying loop optimization techniques, we evaluate three different algorithms. With each one, we apply a set of optimization techniques, showing the performance benefit or penalty, in three CUDA architecture versions, including Fermi.

The results of these techniques will guide developers on the right path towards efficient code optimization. Preliminary results show that some optimizations recommended for older CUDA architectures may not be useful in Fermi.

Keywords—mti, GPGPU; CUDA; OpenMP; compiler; code performance; automatic parallelization; productivity;

I. INTRODUCTION

The world of HPC (High Performance Computing) is undergoing rapid changes. Computer architectures capable to achieve high performance [1] have broadened.

HPC systems can be built with a reasonable cost per GFLOP. Several compute nodes, where each node consists of multiple many-core processors, are common nowadays. The irruption of computational accelerators, like FPGAs [2] and GPUs [3] is adding even more performance while maintaining low cost per GFLOP.

However, this generalization of massive parallel machines, with complex processor hierarchies, needs experienced developers capable of taking advantage of its raw power. The theoretical performance is rarely achieved with current tools and libraries. In addition, most HPC users are not expert developers, but scientists and engineers from a wide variety of fields (physics, chemistry, medicine, …). These people, the main consumers of GFLOP’s, will not be able to exploit new architectures unless simple but powerful tools are provided.

Expert developers may not be ready to exploit complex processor hierarchies either. Hardware accelerators have different instruction sets, and even different programming paradigms, which developers need to learn for each device.

Each version may need some recoding in order to take benefit from architectural improvements.

From our point of view, MPI or OpenMP, the prevailing tools for programming parallel systems, are not acceptable and new tools and languages are clearly needed if we want to take advantage of new hardware capabilities.

The OpenCL [4] standard, which many manufacturers have joined, represents an effort to create a common programming interface for heterogeneous devices. However, it is still immature, and its programming model is not simple.

CUDA [5] is a more mature and widespread approach, although currently it only supports NVIDIA devices. It offers a programming interface (mostly C with a small set of extensions). This framework allows HPC users to reimplement their codes using GPU devices. Despite of the relative simplicity of building a code using this framework, it is hard to achieve a good performance rate given the huge coding and optimization efforts required to obtain the maximum performance from the architecture.

This language diversity, and the difficulties associated with different programming languages, schemes and libraries, has paved the way for the emergence of the DARPA HPCS (High Productivity Computer Systems, [6]) program. HPCS seeks to create Petaflop systems but, rather than focusing of theoretical performance, they focus on decreasing the time-to-solution. In order to achieve this target, new tools and languages need to focus on productivity, and development time need to be taken into account.

The aim of this paper is to use our language, mtic, and its compiler, mtic, as fast-prototyping tools, in order to study different cases in the parallelization of loops. The performance impact of these optimizations is used to compare three different CUDA architectures. Our results can be used as a guide for developers seeking to improve performance with new CUDA architectures, raising their productivity by reducing coding effort.

The remainder of the paper is organized as follows. We begin with a brief explanation and description of mtic as an example of a fast-prototyping tool in Section II. A comparison of the different CUDA architectures is presented in Section III. Different optimization techniques for loop parallelization, commonly used in CUDA codes, are studied in Section IV. We guide our explanations through the use of three applications implemented in mtic for which computational results are presented. Finally, we offer a few
concluding remarks and consider future work in Section V.

II. **MTI** LANGUAGE AND ITS COMPILER

Our proposal to increment HPC productivity is **mti**. **mti** is a high level parallel language where parallelism is expressed through the use of compiler directives that follow the OpenMP syntax. The performance of the MPI and hybrid MPI+OpenMP code generated by the **mti** compiler has been presented in previous papers.

**mti** offers the HPC programmer a simple and well known language that hides the hardware complexity. Examples of **mti** sources are shown in section IV.

Hardware architects are sometimes forgotten when compilers are designed. However, we think that the capability of a fast backend building process is the key feature for the adoption of a compiler. If a new compiler backend can be written in a few weeks, the hardware can be tested in less time with an increased set of codes. This will facilitate the adoption of the architecture among users.

To provide a fast backend building process, we present templates, representing the most common parallel patterns, where we can introduce optimized versions without too much effort.

The software architecture, conformed by a set of powerful intermediate representation tools, and the information extracted from the source code, allows **mtic**, the **mti** compiler, to produce efficient source code for the destination architecture.

**mtic** is a source to source compiler that translates C code annotated with **mti** directives into high-level parallel code. Although **mtic** supports the most common parallel patterns - **forall** sections, pipelines and task queues -, the current version of the new CUDA **mtic** backend only supports parallel loops (**forall**).

Since all OpenMP directives and clauses are recognized by **mtic**, from a single source code we can produce different binaries (sequential or parallel) depending on the compiler selected to translate the target code produced by **mtic**. However, **mtic** is currently under heavy re-factoring, so some of the OpenMP constructs are silently ignored.

**mtic** works by translating the abstract syntax tree (AST) corresponding to the input source code into an internal representation (IR) based on a class hierarchy. Those parts of the IR corresponding to sequential code in the source are written in the target code with no previous transformation. The compiler searches in the AST for specific patterns using what we call a Filter. These patterns correspond to different high-level parallel constructs. The compiler has a filter class hierarchy to address this search task. Once a pattern is located in the AST, we can apply different mutators to achieve the desired translation. **Mutators** produce local modifications in the AST where they insert the high-level code corresponding to the desired translation. After all **Mutators** have been applied, the new AST is processed by the **CudaWriter** module to write the target CUDA code.

Code generation in **mtic**, as in the former version of the compiler, uses the code pattern concept. A code pattern is an abstraction that represents a specific task in the context of the translation. **mtic** uses two kind of code patterns: static and dynamic. The simplest code patterns are implemented using code templates, while the most complex cases require the implementation of a Mutator.

A code template is a code fragment in the target language that will be modified according to certain input parameters. This code is interpreted and translated into to the IR and afterwards it is grafted in the AST. The design of the backend using code templates will ease the implementation of new future backends.

Operations such as initialization or local data allocation, are common for all kinds of hardware devices. Each of these tasks identifies a pattern, and each pattern is implemented through a code template. To manipulate these code templates and insert them into the IR, **mtic** defines a set of operations that are collected in a tool library and exhibit a common facade.

Current implemented patterns are:

- **Device initialization**: contain tasks like device identification, environment set up and resource allocation.
- **Compute kernel invocation**: the code to be parallelized is extracted from the **mti** source, and the compiler injects it wherever specified by the hardware architect.
- **Reduction operation**: a common parallel operation for each device.
- **Environment cleaning**.

As mentioned earlier, our intention is to allow hardware developers to build a compiler backend for fast prototyping by simply implementing these patterns in code templates.

In order to achieve this, we are using a highly expressive template engine, which can fill and modify annotated destination code. The code in Listing I shows an example of a CUDA kernel template. **mtic** will inject the optimized code into the specified placeholders.

```sql
Listing 1: **mtic** template for a CUDA kernel

```
The aim of mtic is not to produce high performance CUDA code, but to demonstrate the potential of the architecture, and to produce efficient code with low coding effort.

Although a hand-written CUDA implementation will always yield better performance, it may be more productive to write an mti source rather than a CUDA source. Furthermore, mti source can be easily ported to a wide variety of architectures. If the developer needs more performance, the code produced by mtic can be manually-optimized to achieve better results, with less coding effort than a fully hand-written implementation.

III. CUDA ARCHITECTURE EVOLUTION

NVIDIA has been releasing a new CUDA architecture version almost every year. Each architecture revision, although sharing the same computational paradigm, has different hardware resources (i.e. number of multiprocessors or memory available). Optimizing codes for a specific version of the architecture does not guarantee optimal results for the next revision. Fine-tuning is necessary in order to get most out of the platform.

The first attempt to use GPU processors for non-graphical applications was made in 2003 [7], though it was not until February 2007, with the appearance in the market of the first CUDA devices, that GPGPU became widespread. Before CUDA, the programmability of GPUs was a major drawback, and the lack of a general purpose language posed major problems for developers who wanted to implement non-graphical algorithms [8].

The arrival of the first CUDA-enabled device, started a race to increase the performance and programmability of these devices. Table I shows a comparison among four CUDA capable devices. The Table indicates that a new revision appears almost every year.

The second generation of CUDA began with the GT200 processors, and the widely-known Tesla C1060 boards. This generation defined the compute capability 1.3, with double precision floating point operations and an increased number of streaming processor cores (128 to 240). The number of registers was increased and hardware coalescing was added, improving memory access efficiency.

This presented a major difference with previous versions of CUDA. In devices with compute capability lesser than 1.3, memory coalescence needed to be resolved manually by the programmer [9]. This required a huge coding and design effort in order to take advantage of the device’s processing power. However, all the effort was for naught since it was useless on 1.3 or later devices, where coalescence problems were solved by hardware.

Fermi is the latest CUDA architecture revision, 2.0. NVIDIA claims to have applied a new approach to the design and implementation. Architectural differences are evident, with improved double precision performance, ECC support for professional environments, increased frequency and more cores, among other changes.

From our point of view, the main advantage of Fermi architecture is the use of a true cache hierarchy. Some applications adapt seamlessly to the previous memory hierarchy. However, there are memory-bound applications that are not able to take advantage of local storage, leading to poor performance in CUDA.

To avoid this, Fermi presents a configurable L2 cache, transparent to the developer. The only parameter that needs to be manually specified is the cache size, which can be 16Kb or 48Kb. If the programmer wants to use shared memory instead of cache, she can specify a 48Kb shared memory, so the cache will be 16Kb. On the contrary, if the programmer wants to use the transparent cache, she will specify 16Kb of shared memory.

The theoretical performance of this new board seems impressive, but it is important to take into account that some of the optimization work done for previous architectures will not be useful in this new board. For example, the automatic cache seems to obviate the need for local storage (as we will study in Section IV-A).

Regardless of the CUDA architecture chosen, it is necessary to properly adjust the number of blocks and threads per block so as to maximize performance. An improperly chosen combination can cause a noticeable drop in performance. This value needs to be adjusted manually by the user. Although recommended values may be provided for a set of boards, the kernel launch configuration is heavily dependent on the problem characteristics and the board used (see Figure 1). mtic features a compile-time macro that users can set to the recommended value for their platform.

![Figure 1: Comparison of different numbers of threads in the three architectures, using the Mandelbrot set computation implemented in mtic. An incorrect choice in the kernel configuration may hinder performance. Each architecture has its own optimal kernel launch configuration.](image-url)
**IV. CASE STUDIES**

In order to study different loop optimization techniques and their effect on the CUDA architecture, we present the mtci implementations of three well-known algorithms: a simple LU-reduction, the solution of a finite difference equation using the Jacobi iterative method and a Molecular Dynamic simulation.

With each code we will focus our attention on different aspects that offer opportunities to optimize the code produced by mtic. The source code for all the algorithms is available at the mtic project home page.

The computational experience we present here was carried out using three different systems, detailed on Table II. Note that each system is completely different. Our comparisons will focus on CUDA running time, not including memory transfer times unless necessary.

However, some of our computational results compare the performance of OpenMP code executed in the host CPU against the execution in the GPU. Our purpose is to showcase the improvement obtained just by adding a GPU card to a small multicore computer. In addition, the peak power consumption of such a system is much lower than that of an equivalent homogeneous system.

**A. LU reduction**

The LU reduction algorithm, involving LU decomposition, is usually used in HPC as a benchmarking algorithm. LU reduction presents a special parallelized version of a LU decomposition.

With this simple source code, whose mtic implementation is shown in Listing 2, we aim to demonstrate the increased productivity of using mtic. From this source code, mtic is capable of producing different translations, like MPI, Hybrid MPI-OpenMP or CUDA, without modifications.

In order to use the CUDA device, the programmer needs only to specify the target directive (line 1). Furthermore, if the programmer uses the mtic capabilities to specify memory transfers, detailed in Section IV-B, the compiler can take advantage of this information to optimize the code.

**Listing 2: LU reduction in mtic**

```c
#pragma omp target device(cuda) copy_in(M2,L)
copy_out(M2)
#pragma omp parallel shared(M2, L, size)
private(i, j, k)
for(k=0; k&lt;size-1; k++) {
  #pragma omp for
  for (i=k+1; i&lt;size; i++) {
    L[i][k] = M2[i][k] / M2[k][k];
    for (j=k+1; j&lt;size; j++) {
      M2[i][j] = M2[i][j] - L[i][k]*M2[k][j];
    }
  }
}
```

**Table I: CUDA Architecture comparison**

<table>
<thead>
<tr>
<th>GPU</th>
<th>G80</th>
<th>G92a</th>
<th>GT200</th>
<th>Fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>1.0</td>
<td>1.1</td>
<td>1.3</td>
<td>2.0</td>
</tr>
<tr>
<td>Reference board</td>
<td>8800GTX</td>
<td>9800GT</td>
<td>C1060</td>
<td>C2050</td>
</tr>
<tr>
<td>Transistors</td>
<td>681 million</td>
<td>754 million</td>
<td>1.4 billion</td>
<td>3.0 billion</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>128</td>
<td>112</td>
<td>240</td>
<td>512</td>
</tr>
<tr>
<td>Double precision FP capability</td>
<td>None</td>
<td>None</td>
<td>30 FMA ops/clock</td>
<td>256 FMA ops/clock</td>
</tr>
<tr>
<td>Single precision FP capability</td>
<td>128 MAD ops/clock</td>
<td>112 MAD ops/clock</td>
<td>240 MAD ops/clock</td>
<td>512 MAD ops/clock</td>
</tr>
<tr>
<td>Special Function Units (SFUs)/SM</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Warp Schedulers</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Shared Memory per SM</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>48 or 16</td>
</tr>
<tr>
<td>L1 Cache per SM</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>48 or 16</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>768 Kb</td>
</tr>
<tr>
<td>Load/Store address width</td>
<td>32 bit</td>
<td>32 bit</td>
<td>32 bit</td>
<td>64 bit</td>
</tr>
<tr>
<td>Clock rate</td>
<td>513Mhz</td>
<td>600Mhz</td>
<td>1600Mhz</td>
<td>3000Mhz</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>60GB/s</td>
<td>57.6GB/s</td>
<td>102.4GB/s</td>
<td>144GB/s</td>
</tr>
</tbody>
</table>

**Table II: Experimental platforms**

<table>
<thead>
<tr>
<th>Name</th>
<th>Ilion</th>
<th>Peco</th>
<th>Zape</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manufacturer</td>
<td>Intel</td>
<td>Intel</td>
<td>AMD</td>
</tr>
<tr>
<td>Model</td>
<td>Q9500</td>
<td>E5520</td>
<td>Phenom 9550</td>
</tr>
<tr>
<td># of cores</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td># of threads</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>2 * 3Mb</td>
<td>4 * 256kb</td>
<td>4 * 512Kb</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>-</td>
<td>8Mb</td>
<td>2048Kb</td>
</tr>
<tr>
<td>Clock rate</td>
<td>2.83Ghz</td>
<td>2.26 Ghz</td>
<td>2200Mhz</td>
</tr>
<tr>
<td>CUDA board</td>
<td>9800GT</td>
<td>Tesla C1060</td>
<td>GeForce GTX 280</td>
</tr>
<tr>
<td>Memory Available</td>
<td>3Gb</td>
<td>4Gb</td>
<td>4Gb</td>
</tr>
</tbody>
</table>
The serial version of this code contains 56 lines. The OpenMP implementation only adds one line (the `omp parallel for`). The mti version of the code includes some modifications to the OpenMP version, intended to avoid unnecessary memory transfers, and thus containing 59 lines of code. Note that the mti code version is compatible with the OpenMP implementation. The CUDA source produced by the current mtic version has 154 lines of code, including error checking, CUDA memory manipulation, etc. The kernel extracted from the annotated loop is shown in Listing 3. This represents a tremendous reduction in the coding effort, as developers can focus on algorithms and high level implementation rather than working on low level code, with CUDA pointers, memory transfers or kernel parameterization.

Listing 3: LU reduction, automatically generated kernel

```c
__global__ void CM_0_loopKernel0(double *M2_cu, double *L_cu, int size, int k)
{
    int i = (blockIdx.x * blockDim.x) + threadIdx.x + (k + 1);
    int j;
    if ((i < size)) {
        L_cu[(k * 4096) + i] = M2_cu[(k * 4096) + i] / M2_cu[(k * 4096) + k];
        for (j = k + 1; j < size; j++) {
            M2_cu[(j * 4096) + i] = M2_cu[(j * 4096) + i] - (L_cu[(k * 4096) + i] * M2_cu[(j * 4096) + k]);
        }
    }
}
```

This kernel is obviously memory-bound, since each thread’s main task is to read from or write to memory. On CUDA, using local memory is a common technique for optimizing performance of memory-bound kernels. With minor modifications to the code produced by mtic, we can use local storage to improve the performance of the kernel. Using local storage for the L matrix avoids consequent references to global memory, reducing latency in memory access. Each thread accesses only one shared memory position, avoiding bank conflicts.

As is shown in Figure 2 using local storage in older architectures yields a significant performance boost. However, the same technique used in Fermi only provides a minimal benefit. The new cache hierarchy used to cache global memory accesses makes the manual implementation of this local storage unnecessary.

The implementation of the construct mti cache, which automatically produces code memory access caching, is currently in progress and planned for the next compiler release.

This kernel is obviously memory-bound, since each thread’s main task is to read from or write to memory. On CUDA, using local memory is a common technique for optimizing performance of memory-bound kernels. With minor modifications to the code produced by mtic, we can use local storage to improve the performance of the kernel. Using local storage for the L matrix avoids consequent references to global memory, reducing latency in memory access. Each thread accesses only one shared memory position, avoiding bank conflicts.

As is shown in Figure 2 using local storage in older architectures yields a significant performance boost. However, the same technique used in Fermi only provides a minimal benefit. The new cache hierarchy used to cache global memory accesses makes the manual implementation of this local storage unnecessary.

The implementation of the construct mti cache, which automatically produces code memory access caching, is currently in progress and planned for the next compiler release.

B. The Jacobi method

A key issue for enhancing performance in the CUDA architecture is reducing data transfers between host and device.

Listing 4: Iterative loop in Jacobi, implemented in mti/OpenMP

```c
while ((k < maxit) && (error > tol)) {
    error = 0.0;
    #pragma omp target device (cuda) copy_in(uold, f, u) copy_out(u)
    #pragma omp parallel shared(uold, u, ...) private(i, j, resid)
    {
        #pragma omp for reduction(+:error )
        for (i = 0; i < m; i++)
            for (j = 0; j < n; j++)
                uold[i][j] = u[i][j];
        #pragma omp for reduction(+:error )
        for (i = 0; i < (m - 2); i++)
            for (j = 0; j < (n - 2); j++)
                resid = ...
                error += resid * resid;
        k++;
        error = sqrt(error) / (double) (n * m);
    }
}
```

The code in Listing 4 is the iterative loop in the Jacobi method both in mti and OpenMP. In line 3, with syntax taken from [10], we specify the target device for the parallel loops in lines 6 and 10. The `copy_in` and `copy_out` clauses in the directive at line 3 state the memory positions to be transferred to and from the device. Figure 3 measures the performance hit of this language feature by comparing a pure OpenMP implementation (8 threads, one per core) with CUDA code generated by mtic specifying (label CUDA v2) the memory transfers with these clauses and without them.
In our translation strategy, at the end of each parallel region we synchronize host and device memories. Inside a parallel region we assume that memory locations allocated in the host remain unchanged. The programmer has to use the OpenMP \texttt{flush} construct in order to synchronize host and device for the case where access to variables computed in the device in a previous parallel loop is needed inside the parallel region. The insertion of the \texttt{flush} construct is not required in the case of function calls because they are automatically translated into device code.

The performance of the nested loop can be improved using loop collapsing techniques. A \texttt{COLLAPSE} clause is included in the new OpenMP 3.0 standard. \texttt{COLLAPSE} takes a constant positive integer expression as its parameter. The parameter specifies how many loops the compiler should collapse into one loop before parallelizing the resulting loop.

When the \texttt{mtic} CUDA backend finds the collapse clause, it generates a 2D kernel. The \texttt{x} coordinates represent the first loop and the \texttt{y} coordinates represent iterations of the second loop. This implementation then produces lighter CUDA threads, reducing memory access conflicts and increasing granularity. However, a correct kernel launch configuration must be chosen in order to increase performance, as we can see in Figure 4.

C. Molecular Dynamic simulation

Given positions, masses and velocities of \(np\) particles, the routine shown in Listing 5 computes the energy of the system and the forces on each particle. The code is an \texttt{mti} implementation of a simple Molecular Dynamics (MD) simulation. It employs an iterative numerical procedure to obtain an approximate solution whose accuracy is determined by the time step of the simulation.

In each simulation step, the algorithm performs two basic operations: \texttt{compute} (shown in Listing 5) and \texttt{update}. The \texttt{update} operation is simply a for loop that runs over the particles, updating their positions, velocities and accelerations. From a computational point of view, \texttt{compute} is more intensive than \texttt{update}.

With this algorithm we want to study the best combination of GPU/CPU to target the parallel code. Let us denote the CPU by \(C\) and GPU by \(G\). We measured four different versions of the code:

- \texttt{CC}: both routines in the CPU (pure OpenMP code)
- \texttt{GG}: both routines in the GPU (pure CUDA code)
- \texttt{GC}: \texttt{compute} in the GPU and \texttt{update} in the CPU
- \texttt{CG}: \texttt{compute} in the CPU and \texttt{update} in the GPU

Figure 5 shows the speedup obtained for three different problem sizes (number of particles). The best performance is obtained when both routines are placed in the GPU. For the hybrid OpenMP/CUDA codes, the best choice is to allocate...
Listing 5: Molecular Dynamic code simulation in mti

void compute(int np, int nd, double *box, vnd_t *pos, ...) {
    double x, d, pot, kin;
    int i, j, k;
    vnd_t rij;
    pot = kin = 0.0;
    #pragma omp target device(cuda) copy_in(f,vel,pos,box) copy_out(f)
    #pragma omp parallel for default(shared) private(i, j, k, rij, d)
        reduction(+ : pot, kin)
    for (i = 0; i < np; i++) { /* Pot. energy and forces */
        for (j = 0; j < nd; j++)
            f[i][j] = 0.0;
        for (j = 0; j < np; j++) {
            if (i != j) {
                d = dist(nd, box, pos[i], pos[j], rij);
                pot = pot + 0.5 * v(d);
            }
        }
        kin = kin + dotr8(nd, vel[i], vel[i]); /* kinetic energy */
    }
    kin = kin * 0.5 * mass;
    *pot_p = pot;
    *kin_p = kin;
}

V. CONCLUSIONS AND FUTURE WORK

As we have shown, the CUDA architecture has evolved rapidly over the last few years, and the changes in each architecture revision have enhanced performance. However, these architectural changes have rendered some of the optimization effort useless.

For newcomers to the world of GPGPU, acquiring a Fermi board is better than wasting time optimizing code for old architectures, because most of the effort will not be reusable in subsequent revisions.

Those who work in GPGPU may want to concentrate on advanced optimization techniques, that may be usable in future versions of the architecture. In particular, using 2D kernels for matrix operations appears to maintain performance benefits in new architectures, thus, stencil kernel optimization seems worth the effort.

This paper presents an evaluation of different optimization techniques implemented (or scheduled to be implemented) in the new backend of our mtic compiler. These techniques, allow developers to quickly test different optimization strategies for their code. When a correct optimization strategy is chosen, using mtic generated code as a start point eases the development process, and increases programmer productivity.

Our compiler is designed to be flexible and portable, and we pretend that, with the experience achieved in the development of the CUDA backend, incorporation of new target languages (OpenCL, for example) should not require an unaffordable effort.

Work in progress within the framework of this project includes the following:

- To increase the number of algorithms parallelized using our compiler, paying particular attention to commercial applications.
• To compare the performance in platforms with a larger number of CPU cores.
• To study and implement additional compiler optimizations that will enhance the performance of the target code.
• To study the generation of hybrid CUDA+OpenMP code.

Some of the compiler optimizations that are currently under study or development are the following:
• To improve locality through a better use of the memory hierarchy.
• To use the texture memory to store read-only data.
• To enhance the translation of nested loops by taking advantage of the architecture design.
• To achieve an intelligent load balance between host and device.

With our approach, the performance loss with respect to a direct CUDA implementation is clearly compensated by the significantly lower development effort. Taking this into account, we conclude that mti is appropriate to implement certain classes of parallel applications.

REFERENCES


