Customizing Instruction Set Extensible Reconfigurable Processors using GPUs

Unmesh D. Bordoloi[1], Bharath Suri[1], Swaroop Nunna[2], Samarjit Chakraborty[2], Petru Eles[1], Zebo Peng[1]
[1]Linköpings Universitet, Sweden [2]TU Munich, Germany
[1]E-mail:{ bhasu733@student.liu.se}, {unmbo, petel, zebpe}@ida.liu.se
[2]E-mail: {swaroop.nunna, samarjit.chakraborty}@rcs.ei.tum.de

***Abstract*—Many reconfigurable processors allow their instruction sets to be tailored according to the performance requirements of target applications. They have gained immense popularity in recent years because of this flexibility of adding custom instructions. However, most design automation algorithms for instruction set customization (like enumerating and selecting the optimal set of custom instructions) are computationally intractable. As such, existing tools to customize instruction sets of extensible processors rely on approximation methods or heuristics. In contrast to such traditional approaches, we propose to use GPUs (Graphics Processing Units) to efficiently solve computationally expensive algorithms in the design automation tools for extensible processors. To demonstrate our idea, we choose a custom instruction *selection* problem and accelerate it using CUDA (CUDA is a GPU computing engine). Our CUDA implementation is devised to maximize the achievable speedups by various optimizations like exploiting on-chip shared memory and register usage. Experiments conducted on well known benchmarks show significant speedups over sequential CPU implementations as well as over multi-core implementations.**

I. Introduction

Instruction set extensible reconfigurable processors have become increasingly popular over the last decade. Their popularity is driven by the fact that they strike the right balance between the flexibility of general purpose processors and the performance of ASICs. The existing instruction cores of extensible processors may be extended with *custom instructions* to meet the performance requirements of the target application.

Our contributions: Design automation problems for customizing instruction sets are computationally intractable (NP-hard) [17]. In this paper, we propose the use of GPUs to accelerate the running times of design automation tools for customizable processors. To show the applicability of GPUs in instruction set customization algorithms, we choose a custom instruction *selection* problem and accelerate it using CUDA (Compute Unified Device Architecture), NVIDIA's parallel computing architecture based on GPUs [12]. Our contribution is interesting because we show how the custom instruction selection problem can be engineered to exploit on-chip memory on GPUs and other CUDA features. We choose custom instruction *selection* problem because (i) of its intractability (see Section II) and (ii) it has received lot of attention in recent years (see Section I-A). We would like to note that in contrast to traditional approaches (like approximation schemes [2]), our GPU-based technique provides *optimal* solutions.

Our contribution is also practically relevant because instruction set customization techniques are incorporated into compilers [18] and such compilers are invoked repeatedly by designers. Typically, a designer would choose the values of certain system parameters (e.g., processor frequency, deadlines) once an implementation version of the application has been fixed and then invoke the compiler to determine whether the constraints (like performance and area) are satisfied. If the compiler returns a negative answer, then some of the parameters are modified (e.g., an optimized version of the implementation is chosen or processor frequency is scaled) and the compiler is invoked once again. Thus, the designer iteratively interacts with the tools to adjust the parameters and functionalities till the performance constraints are satisfied. If each invocation of the tool takes long time to run to completion, the interactive design sessions become tedious affecting the design productivity. Hence, by bringing down the running times of the tools by significant margins using GPUs, the usability of such tools may be improved. Further, this comes at no additional cost because most desktop/notebook computers today are already equipped with a commodity GPU.

Finally, given that the combinatorial optimization problem mapped to GPU in this paper is a variant of the knapsack problem, our results might be meaningful to a wider range of problems in the design automation domain.

Overview of the problem: Given a library of custom instruction candidates the goal is to select a subset of instructions such that the performance is enhanced while keeping the area costs at minimum. In such a scenario, conflicting tradeoffs are inherent because while the performance of a system may be improved by the use of custom instructions, the benefits come at the cost of silicon area. Hence, a designer is not interested in identifying one solution which meets the performance requirements, but would rather like to identify all the conflicting tradeoffs between performance and area. The designer can then inspect all solutions and pick one which suits his/her design.

Note that in the above setup, part of the application is implemented as software on a programmable processor and the rest in hardware as custom instructions on a sea of FPGA. In this setting, a good metric for performance is the *processor utilization* because it is a measure of the load on the processor. Moreover, in this paper, we assume that the processor is running hard real-time tasks and processor utilization is a well known metric that is used to capture the feasibility of such systems (for details, see Section II). Formally, let (c, u) denote the hardware cost c, arising from the

use of custom instructions and the corresponding utilization u, of the processor. We are then interested in generating the Pareto-optimal curve [4] $\{(c_1, u_1), \ldots, (c_n, u_n)\}$ in a multi-objective design space. Each (c_i, u_i) in this set has the property that there does not exist any implementation choice with a performance vector (c, u) such that $c \leq c_i$ and $u \leq u_i$, with at least one of the inequalities being strict. Further, let $\mathcal{S}$ be the set of performance vectors corresponding to all implementations choices. Let $\mathcal{P}$ be the set of performance vectors $\{(c_1, u_1), \ldots, (c_n, u_n)\}$ corresponding to all the Pareto-optimal solutions. Then for any $(c, u) \in \mathcal{S} - \mathcal{P}$ there exists a $(c_i, u_i) \in \mathcal{P}$ such that $c_i \leq c$ and $u_i \leq u$, with at least one of these inequalities being strict (i.e., the set $\mathcal{P}$ contains *all* performance tradeoffs). The vectors $(c, u) \in \mathcal{S} - \mathcal{P}$ are referred to as *dominated solutions*, since they are "dominated" by one or more Pareto-optimal solutions.

A. Related Work

Note that in this work we focus on the custom instruction *selection* phase. In recent years, lot of research has been devoted to custom instruction selection techniques so as to optimize either performance or hardware area [2], [9]. In this paper, we have considered a more general problem formulation by focusing on multi-objective optimization instead of optimizing for a single objective.

Custom instruction selection techniques assume that a library of custom instruction candidates is given. Such a library of custom instructions may be *enumerated* by extracting frequently occurring computation patterns from the data flow graph of the application [14], [17]. We believe our paper would motivate researchers to explore the possibility of deploying GPUs in this phase as well.

Motivation for using GPU: It should be mentioned in this section that our paper has been motivated by the recent trend of applying GPUs to accelerate non-graphics applications. Applications that have harnessed the computational power of GPUs span across numerical algorithms, computational geometry, database processing, image processing, astrophysics and bioinformatics [13]. Of late, there has also been lot of interest in accelerating computationally expensive algorithms in the computer-aided design of electronic systems [7], [3], [5]. There are many compelling reasons behind exploiting GPUs for such non-graphics related applications. First, modern GPUs are extremely powerful. For example, high-end GPUs, such as the NVIDIA GeForce GTX 480 and ATI Radeon 5870, have 1.35 TFlops and 2.72 TFlops of peak single precision performance, whereas a high-end general-purpose processor such as the Intel Core i7-960, has a peak performance of 102 Gflops. Additionally, the memory bandwidth of these GPUs is more than $5\times$ greater than what is available to a CPU, which allows them to excel even in low compute intensity but high bandwidth usage scenarios. Finally, GPUs are now commodity items as their costs have dramatically reduced over the last few years. The attractive price-performance ratios of GPUs gives us an enormous opportunity to change the way design automation tools like compilers for instruction set customization perform, with almost no additional cost.

However, implementing general purpose applications on a GPU is not trivial. The GPU follows a highly parallel computational paradigm. Since many threads run in parallel, it must be ensured that they do not have arbitrary data dependency on each other. Hence, the challenge is to correctly identify the data parallel segments so that dependency constraints of the application mapped to the GPU are not violated. Secondly, in order to exploit the high bandwidth on-chip shared memory, it is important to identify the frequently accessed data structures so that they can be pre-fetched in the shared memory.

II. Problem Description

In this section, we discuss our system model and formally present the multi-objective optimization problem.

System Model: We assume a multi-tasking hard real-time system. Formally, we use the sporadic task model [1] in a preemptive uniprocessor environment. Thus, we are interested in selecting custom instructions for a task set $\tau = \{T_1, T_2, \ldots, T_m\}$ consisting of m hard real-time tasks with the constraint that the task set is schedulable. Any task T_i can get triggered independently of other tasks in τ. Each task T_i generates a sequence of jobs; each job is characterized by the following parameters:

- *Release Time*: the release time of two successive jobs of the task T_i is separated by a minimum time interval of P_i time units.
- *Deadline*: each job generated by T_i must complete by D_i time units since its release time.
- *Workload*: the worst case execution requirement of any job generated by T_i is denoted by E_i.

Throughout this paper, we assume the underlying scheduling policy to be the earliest deadline first (EDF). Assuming that for all tasks T_i, $D_i \geq P_i$, the schedulability of the task set τ can be given by the following condition $(U = \sum_{i=1}^{m} \frac{E_i}{P_i}) \leq 1$, where U is the processor utilization due to τ [1].

Problem Statement: For a given processor P, let each of the tasks T_i have n_i number of custom instruction choices which can be implemented in hardware. For simplicity of exposition, assume that the processor P's clock frequency is constant and all the execution times of the tasks are specified with respect to this clock frequency. The objective is to minimize P's utilization (by mapping certain custom instructions onto hardware) and at the same time also minimize the total hardware cost. In other words, our goal is to compute the *cost-utilization* Pareto curve $\{(c_1, u_1), \ldots, (c_n, u_n)\}$ for a prespecified clock frequency of P. Note that it is possible that the given task set (without utilizing custom instructions) is already schedulable on the processor, i.e., $U < 1$. In these cases, the Pareto curve reveals how the utilization can be further reduced at the cost of hardware area. This is interesting because the designer can then use the processor for soft real-time tasks or clock the processor at a lower frequency to save power. On other hand, if the original task set is not schedulable, the Pareto curve reveals the hardware costs at which the task set becomes schedulable. Note that the designer can also choose to clock the processor

Algorithm 1 Custom Instruction Selection

Require: The task set τ, and a set S_i for each task T_i.
1: $U_{0,0} \leftarrow \sum_{i=1}^{m} E_i/P_i$
2: **for** $j \leftarrow 1$ to mC **do**
3: $\quad U_{0,j} \leftarrow \infty$
4: **end for**
5: **for** $i \leftarrow 1$ to m **do**
6: $\quad$ **for** $j \leftarrow 0$ to mC **do**
7: $\quad\quad$ For each pair $(\delta_{i,k}, c_{i,k})$ that belongs to the set S_i
8: $\quad\quad U_{i,j} \leftarrow \min\{U_{i-1,j}, U_{i-1,j-c_{i,k}} - \delta_{i,k}/P_i\}$
9: $\quad$ **end for**
10: **end for**

at a higher frequency to make the task set schedulable. By revealing the utilization points for $U > 1.0$, our results will expose the higher frequencies at which the processor may be clocked for the system to be schedulable.

Each of the n_i choices of the task T_i is associated with a certain hardware cost. Choosing the jth implementation choice for the task T_i lowers its execution requirement on P from E_i to $e_{i,j}$. Equivalently, the amount by which the execution requirement of T_i gets lowered on P is $\delta_{i,j} = E_i - e_{i,j}$. Hence, for each task T_i we have a set of choices $S_i = \{(\delta_{i,1}, c_{i,1}), \ldots, (\delta_{i,n_i}, c_{i,n_i})\}$, where $c_{i,j}$ is the hardware cost associated with the jth implementation choice. Let $x_{i,j}$ be a Boolean variable that is assigned 1 if the jth implementation choice for the task T_i is chosen and is assigned 0, otherwise. In this setup, the objective is to minimize the utilization $U(S) = \sum_{i=1}^{m} \frac{E_i - x_{i,j}\delta_{i,j}}{P_i}$ and the cost $C(S) = \sum_{i=1}^{m} c_{i,j}x_{i,j}$, where S is the chosen implementation among the various available options.

NP-hardness: The NP-hardness of the problem can be shown by transforming the knapsack problem [10] into a special instance of this problem. Towards this, corresponding to each item in the knapsack problem, we have a task with performance gain equal to the profit and the hardware cost equal to the weight of the item. A complete proof is omitted due to space constraints.

Algorithm: An algorithm to compute optimally the Pareto curve described above consists of two parts. First, a dynamic programming algorithm (Algorithm 1) computes the minimum utilization that might be achieved for each possible cost. The second part finds all undominated solutions (*cost-utilization* Pareto curve) from the entire solution set found by the dynamic programming algorithm. We denote this part as 'Retain Undominated' — a straightforward sequential implementation on CPU. Below, we discuss Algorithm 1.

Let $U_{i,j}$ be the minimum utilization that might be achieved by considering only a subset of tasks from $\{1, 2, \ldots, i\}$ when the cost is exactly j. If no such subset exists we set $U_{i,j} = \infty$. Let the maximum cost be represented by C i.e. $C = max_{(i=1,2,\ldots,n;j=1,2,\ldots,n_i)} c_{i,j}$. Clearly, mC is an upper bound on the total cost that might be incurred. Lines 1 to 4 of Algorithm 1 initialize $U_{0,0}$ to $\sum_{i=1}^{m} E_i/P_i$, and $U_{0,j}$ to ∞ for $j = \{1, 2, \ldots, mC\}$. The values $U_{i,j}$ for $i = 1$ to $i = m$ are computed using the iterative procedure in lines 5 to 10. Thus, any non-infinity value $U_{n,j}$ for $j = \{1, 2, \ldots, mC\}$ implies that there exists a design choice of the task set with utilization $U_{n,j}$ and cost j. It can be easily verified that the running time of Algorithm 1 is $O(nmC)$, where $n = \sum_{i=1}^{m} n_i$, and its space complexity is $O(m^2C)$. The algorithm runs in pseudo-polynomial time, and hence, turns out to be a computationally expensive kernel. In this paper, we accelerate the running times of this algorithm by mapping it to the GPU and obtain optimal and exact solutions as described in Section IV.

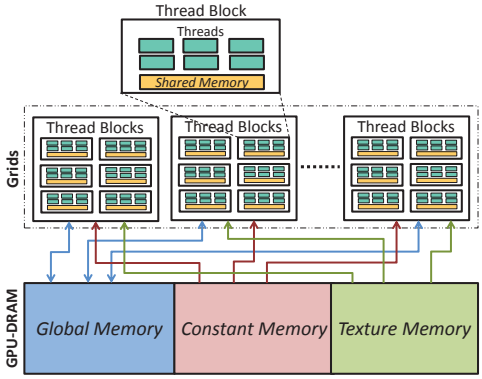

Fig. 1. CUDA programming model

III. CUDA

In this section, we provide a brief description of CUDA. For a complete description, we refer the reader to NVIDIA's guide [12]. CUDA abstracts the GPU as a powerful multi-threaded coprocessor capable of accelerating data-parallel, computationally intense operations. The data parallel operations, which are similar computations performed on *streams* of data, are referred to as *kernels*. Essentially, with its programming model and hardware model, CUDA makes the GPU an efficient streaming platform.

In CUDA, *threads* execute data parallel computations of the kernel and are clustered into blocks of threads referred to as thread blocks. These thread blocks are further clustered into grids. During implementation, the designer can configure the number of threads that constitute a block. Each thread inside a block has its own registers and local memory. The threads in the same block can communicate with each other through a memory space shared among all the threads in the block and referred to as *Shared Memory*. The *Shared Memory* space of the thread block and is typically in the order of KB. However, an explicit communication and synchronization between threads belonging to different blocks is only possible through GPU-DRAM. GPU-DRAM is the dedicated DRAM for the GPU in addition to DRAM of the CPU. It is divided into *Global Memory, Constant Memory* and *Texture Memory*. We note that the *Constant* and *Texture Memory* spaces are read-only regions whereas *Global Memory* is a read-write region. Figure 1 illustrates the above described CUDA programming model. In case a memory location being accessed, by a CUDA memory instruction, resides in GPU-DRAM, i.e., either in *Global*, *Texture* or *Constant Memory* spaces, the memory instruction consumes an additional 400 to 600 cycles. On the other hand, if the memory location resides on-chip in the registers or *Shared Memory*, there will be almost no additional latencies in the absence of memory access conflicts. Note that in contrast to the GPU-DRAM, the *Shared Memory*

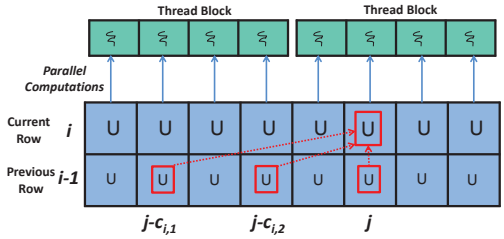

Fig. 2. Data dependency graph for Algorithm 1

region is a on-chip memory space. The additional latencies on GPU-DRAM might obscure the speedups that can be achieved due to parallelization and hence the on-chip shared memory must be judiciously exploited.

IV. Proposed Framework on CUDA

As described in Section II, the computation of the *cost-utilization* Pareto curve to expose the design tradeoffs at custom instruction selection phase involves a pseudopolynomial algorithm (Algorithm 1). In this section, we present our CUDA based framework to implement Algorithm 1 to accelerate its running times. This involves the following major challenges. First, we need to identify and isolate the data parallel computation of the algorithm so that they may be compiled as the *kernels*. Recall that kernels are executed by data parallel threads on CUDA. Secondly, we must devise the algorithm such that it can exploit the on-chip *Shared Memory* and registers to enhance the achievable speedups. Finally, thread block size must be appropriately configured. In light of these challenges, we now provide a systematic implementation of Algorithm 1 in the following.

Identifying data parallelism: As mentioned above, our first goal is to identify the data-parallel portions (*kernels*) in Algorithm 1 which can be computed by CUDA threads in a SIMD fashion. The kernels must not have any data dependencies (on each other) because they will be executed by threads running in parallel. Towards this we first identify the data dependencies in Algorithm 1. Algorithm 1 (lines 5 - 10) builds a dynamic programming (DP) matrix. The i-th row of the matrix corresponds to the i-th task T_i in the task set described in Section II. Each cell in the i-th row represents the value $U_{i,j}$ where $j = \{0, 1, 2, \ldots, mC\}$. According to Algorithm 1 (line 8), the computation of these values depends only on the values present in the previously computed rows. Figure 2 illustrates this for the cell $U_{i,j}$. This implies that the values of the cells of the same row in the DP-based matrix can be computed independently of each other by using different CUDA threads in SIMD fashion. Therefore, we isolate (line 8 of Algorithm 1) as the *kernel* of our CUDA implementation. In the following, we explain the effective usage of the on-chip share memory.

Memory usage: We store the DP-matrix in the *Global Memory* space (GPU-DRAM). Note that we use *Global Memory* space instead of *Constant* or *Texture Memory* because *Constant* and *Texture Memory* are read-only regions. During the computation of our DP-matrix we need to perform both read (to fetch values from the previous rows computed earlier) and write (to update the DP-matrix with the values of the row computed in the current iteration) operations which can only be done explicitly with *Global Memory*. Also, note that we have so far not used the on-chip *Shared Memory* because the size of the *Shared Memory* is typically quite small (see Section III) and the DP-matrix cannot fit into it.

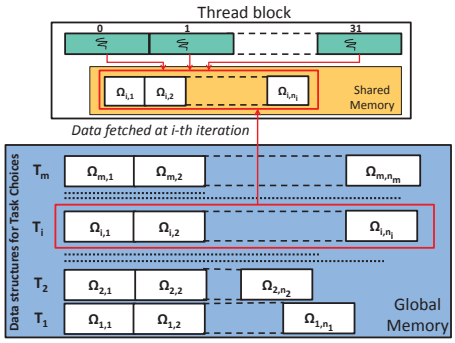

Fig. 3. Data fetched into shared memory at i-th iteration of the algorithm

However, the on-chip *Shared Memory* can be exploited to store other frequently accessed data structures. To identify such data structures, we once again focus on the kernel operations of our algorithm (line 8 of Algorithm 1). We note that the computation of each of the $U_{i,j}$ values corresponding to the task T_i (i.e., the i-th row of our DP-based matrix) needs the values of all the n_i hardware implementation choices of T_i. Now let us denote the choice tuple $(\delta_{i,k}, c_{i,k})$ by $\Omega_{i,k}$ for $k = \{1, 2, \ldots, n_i\}$. Thus, from line 8 of Algorithm 1, the computation of the i-th row in the DP-matrix requires the values $\Omega_{i,1}, \Omega_{i,2}, \ldots, \Omega_{i,n_i}$.

This set, $\{\Omega_{i,1}, \Omega_{i,2}, \ldots, \Omega_{i,n_i}\}$, is essentially a subset of the overall specification of the task set. Also, in iteration i of computing the DP-matrix this set of required data structure remains constant, i.e., information about the other parts of the task set is not required. This set changes only at the next iteration (iteration $i + 1$) because this iteration corresponds to a different task in the task set which might have a different set of hardware implementation choices. This observation provides an opportunity to significantly reduce the GPU based execution times by loading these values $\{\Omega_{i,1}, \Omega_{i,2}, \ldots, \Omega_{i,k}\}$ to the on-chip *Shared Memory* at the beginning of each iteration. Compared to the DP-matrix, this set of values is much smaller and can fit into the on-chip *Shared Memory*. Figure 3 illustrates our scheme of prefetching the required data structure from *Global Memory* to *Shared Memory* at the start of each iteration. The figure shows a thread block (which consists of 32 threads) fetching the required data from the *Global Memory* at the i-th iteration.

Register usage: The threads of CUDA access registers (used to store the local variables) which have very low access latencies like the *Shared Memory*. If the total number of required registers is greater than that available in the processor for the current set of thread blocks, then CUDA will schedule less thread blocks simultaneously to cope with the situation. This will decrease the degree of parallelism offered by CUDA. In Algorithm 1 there is a division operation (line 8) that is known to contribute to high

register usage. Hence, in our implementation, we convert it into a multiplication operation to optimize the register usage.

Thread block: We recall from Section III that the on-chip memory is shared only between the threads within a single block. Hence, configuring the thread blocks to an appropriate size is also important to effectively exploit the GPU on-chip memory. For example, if we choose a very small thread block size, then the computation of each row in our DP-based matrix will involve lot of thread blocks. However, only the threads within a thread block share the same chunk of on-chip memory. This implies that data from the *Global Memory* to *Shared Memory* will have to be transfered for a large number of thread blocks, inspite of the fact that all the threads in a single iteration need the same data structures - $\{\Omega_{i,2}, \ldots, \Omega_{i,k}\}$, as described above.

We note, however, that thread block size cannot be increased arbitrarily to increase performance. As an example, consider the Tesla GPU from NVIDIA that allows a maximum of 1024 threads in a thread block. Interestingly, the total number of threads that can be active simultaneously is 1536, as limited by the hardware. If we set thread block size to 1024, only one thread block (i.e., 1024 threads) will run in parallel. This is because it is not possible for the GPU to run only some threads of a thread block. On the other hand, if we set thread block size as 768, two thread blocks (i.e., 1536 threads in total) can run in parallel because all the threads can be activated simultaneously. Hence, for Tesla, we choose 768 as the thread block size. Under certain conditions (like register spillage, *Shared Memory* capacity overrun), it is possible that a thread block size of 1024 delivers a better performance than with 768. Our optimizations on *Shared-Memory* and register usage, as described above, ensure that such scenarios do not occur for the problem addressed in this paper.

V. Experimental Results

In this section, we report the experimental results that were obtained by running our CUDA-based implementation on 5 different task sets that were constructed using well known benchmarks. We compared these results with those obtained by running sequential CPU-based implementation as well as multi-core implementations.

Experimental Setup: We created 5 task sets with number of tasks between 8 and 12. These task sets comprise of 5 benchmarks (*compress, jfdctint, ndes,edn, adpcm*) from WCET [15], 3 benchmarks (*aes, sha, rijndael*) from MiBench [8], 3 benchmarks (*g721encoder, djpeg, cjpeg*) from MediaBench [11] and one benchmark (*ispell*) from Trimaran [16]. Table I shows the combination of benchmarks incorporated in each of the task sets and the sizes of the task sets.

We chose the Xtensa [6] processor platform from Tensilica for our experiments. Xtensa is a configurable processor core allowing application-specific instruction-set extensions. The custom instruction configurations from the benchmarks were obtained by using the XPRES compiler from Tensilica. First, the workload E_i is computed for each task T_i which refers to the workload without any custom instruction enhancement. Assuming Tensilica identifies n_i custom instructions for each

Task Set	Benchmarks	Size
1	aes, djpeg, g721decode, rijndael, adpcm jfdctint, cjpeg, edn, ispell, sha, ndes, compress	12
2	djpeg, g721decode, rijndael, adpcm jfdctint, cjpeg, edn, ispell, sha, ndes, compress	11
3	aes, djpeg, g721decode, rijndael jfdctint, cjpeg, edn, ispell, sha, ndes	10
4	adpcm, rijndael, cjpeg, ispell sha, ndes, djpeg, compress, edn	9
5	cjpeg, ispell, edn, sha g721decode, djpeg, compress, ndes	8

TABLE I
Task sets

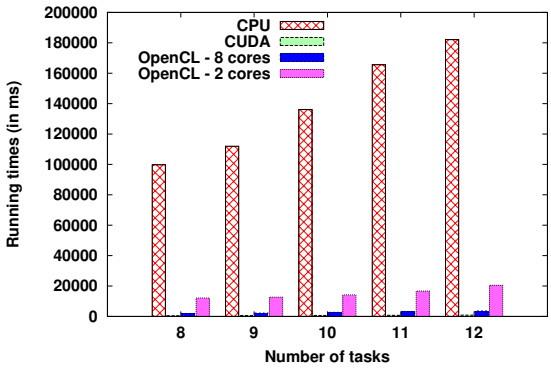

Fig. 4. Comparison of all implementations.

task T_i, we compute — (i) the performance improvement $\delta_{i,j}$ and (ii) the area cost $c_{i,j}$ — for each of the jth custom instruction configurations. The workload is in terms of Multiply-Accumulate (MAC) operation's cycles and the hardware area is in terms of number of adders.

We set P_i for the tasks, such that the $U = \sum_{i=1}^{N} \frac{E_i}{P_i}$ is 0.80, 1.00, 1.05, 1.08 and 1.10 for the 5 different tasks set. The GPU used for evaluating our experiments was a NVIDIA Tesla M2050 GPU. This GPU was connected via on-board PCI express slot to the host machine with 2 Xeon E5520 CPUs, each with 4 cores, i.e., 8 cores overall and each core ran at 2.27GHz. We compared the performance of our CUDA implementation against an OpenCL implementation on the multi-core host. We also compared the results with an OpenCL implementation on a dual core laptop, each core running at 2.1 GHz. We also implemented (in C) a sequential version of the algorithm that was run on a single core of the Xeon host machine. Thus, we have four implementations overall — CUDA, OpenCL 8-core, OpenCL 2-core and a sequential CPU.

Results: To illustrate the benefits of our CUDA implementation, we compared the running times for computing the Pareto

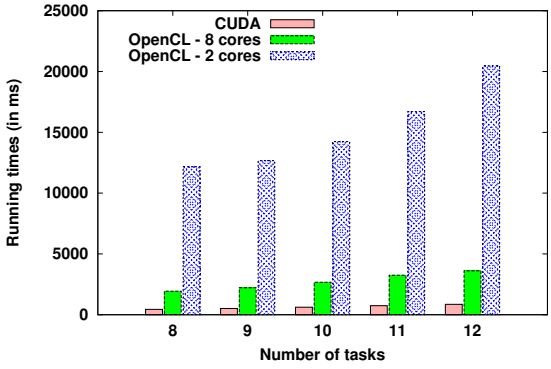

Fig. 5. Comparison of OpenCL multi-core and CUDA implementations.

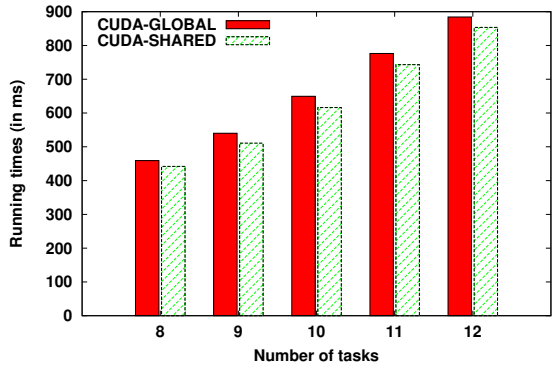

Fig. 6. Running times of CUDA-Shared and CUDA-Global implementations.

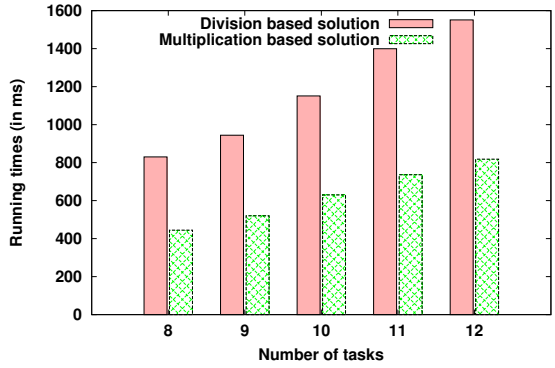

Fig. 7. The speedup obtained after minimizing register spillage using multiplication operation instead of division.

curve for the different task sets for all four implementations discussed above. Figure 4 plots the running times for these implementations. Our CUDA implementation is 220× faster than the CPU implementation (on single processor). Due to such tremendous speedups, the bar graph showing the CUDA implementation almost co-incides with the x-axis. To better illustrate the speedups when compared to the multi-cores, Figure 5 shows only the CUDA implementation along with the multi-core implementations. As seen in this figure, even compared to a dual-core (on a laptop) and 8-core implementations (on Intel Xeon), our GPU-based implementation is 24× and 8× faster, respectively.

To illustrate the benefits of *Shared-Memory* usage and register size optimization, we conducted further experiments. Figure 6 shows the running times of CUDA-Global (where we do not utilize *Shared-Memory*) and CUDA-Shared (where we exploit the on-chip *Shared-Memory* as discussed in Section IV). Using the on-chip shared memory leads to an improvement of 6% on an average. Similarly, our optimization to manage the register spillage (based on the conversion of the division operation as a multiplication) also yields significant speedups (on average around 85%) as shown in Figure 7. Finally, in Figure 8 we illustrate the Pareto curve that was obtained for task set 1. Note that that the solution space is significantly huge, but for clarity of illustration, we have plotted only a part of the graph and the x-axis is truncated when cost is 4000.

We would like to mention that all the running times reported here include the time taken for transfer of data from the host machine to the GPU and vice-versa. Also, recall that computing the Pareto curve involves a straightforward algorithm

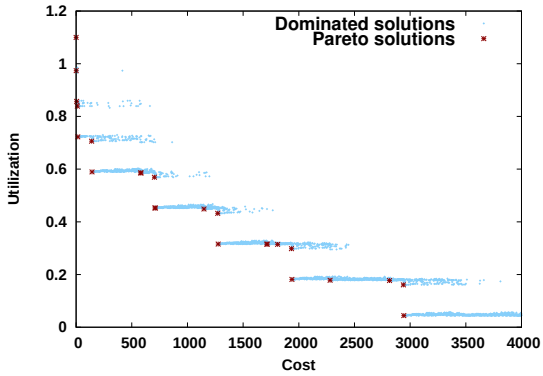

Fig. 8. The Pareto curve for task set 1. The points on the Pareto curve are highlighted with an asterisk.

to retain the undominated solutions after Algorithm 1 (see Section II). This part is implemented in the CPU because it is not amenable to parallelization and its running time is significantly less that Algorithm 1 (always less than 10 milliseconds). However, for accuracy its running times has also been included in the running times reported here.

VI. Concluding Remarks

We presented a technique to implement a custom instruction selection algorithm on GPUs. To the best of our knowledge, this is the first paper on instruction set customization using GPUs. Our technique exploits not just the parallelism but also the shared memory features offered by GPU architectures in order to achieve significant speed ups.

References

[1] S. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *IEEE RTSS*, 1990.
[2] U. D. Bordoloi, H. P. Huynh, S. Chakraborty, and T. Mitra. Evaluating design trade-offs in customizable processors. In *DAC*, 2009.
[3] D. Chatterjee, A. De Orio, and V. Bertacco. GCS: High-performance gate-level simulation with GP-GPUs. In *DATE*, 2009.
[4] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, 2001.
[5] J. Feng, S. Chakraborty, B. Schmidt, W. Liu, and U. D. Bordoloi. Fast schedulability analysis using commodity graphics hardware. In *RTCSA*, 2007.
[6] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.
[7] Kanupriya Gulati and Sunil P. Khatri. Towards acceleration of fault simulation using graphics processing units. In *DAC*, 2008.
[8] M. R. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE Annual Workshop on Workload Characterization*, 2001.
[9] H. P. Huynh and T. Mitra. Instruction-set customization for real-time embedded systems. In *DATE*, 2007.
[10] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer, 2004.
[11] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO*, 1997.
[12] NVIDIA. CUDA Programming Guide version 1.0, 2007.
[13] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
[14] N. Pothineni, A. Kumar, and K. Paul. Application specific datapath extension with distributed i/o functional units. In *VLSI Design*, 2007.
[15] F. Stappert. WCET benchmarks. http://www.c-lab.de/home/en/download.html.
[16] Trimaran:. An infrastructure for research in backend compilation and architecture exploration. http://www.trimaran.org.
[17] A. K. Verma, P. Brisk, and P. Ienne. Rethinking custom ISE identification: a new processor-agnostic method. In *CASES*, 2007.
[18] P. Yu. Design methodologies for instruction-set extensible processors. *PhD Thesis, C.Sc. Dept., National University of Singapore*, Jan. 2009.