Autotuning GPU-accelerated QAP Solvers for Power and Performance

Abhilash Chaparala
Department of Computer Science
Texas State University
San Marcos, TX
a_c219@txstate.edu

Clara Novoa
Ingram School of Engineering
Texas State University
San Marcos, TX
cn17@txstate.edu

Apan Qasem
Department of Computer Science
Texas State University
San Marcos, TX
apan@txstate.edu

***Abstract*—The Quadratic Assignment Problem (QAP) is an important combinatorial optimization problem with applications in many areas including operations research and chip design. QAP is known to be NP-hard and requires heuristic approaches for most real data sets. Although many algorithms have been proposed for solving QAPs, few have attempted to exploit the fine-grain data parallelism available on accelerator architectures and none have considered aspects of energy efficiency.**

In this paper, we present GPU-accelerated implementations of two QAP algorithms. We parameterize each algorithmic variant along several dimensions including thread granularity, occupancy, shared memory usage and register pressure. We develop an autotuning framework that explores this space of execution parameters and yields CUDA binaries that are efficient in terms of both performance and power consumption. We demonstrate the effectiveness of our tuning framework on an Nvidia Tesla K20c. On a series of experiments on the well-known QAPLIB data sets, our autotuned solutions run at least an order-of-magnitude faster than baseline implementations. The experimental results also provide key insight into the performance characteristics of accelerated QAP solvers. In particular, the results reveal that both algorithmic choice and the shape of the input data sets are key factors in finding an efficient implementation.

***Keywords*-autotuning; energy efficiency; parallel performance;**

I. Introduction

The Quadratic Assignment Problem (QAP) is an NP-hard combinatorial optimization problem. In this problem, the objective is to assign n units to n locations such that the total cost measured as the sum of the products of *flows* between units and *distances* between locations is minimized. QAP maps directly to the facility layout problem in manufacturing systems and plays an important role in the area of operations research. In addition, QAP has wide applicability in many different domains including processor and memory layout optimization, ergonomic design of electronic devices, scheduling problems and economic modeling. Because of its practical and theoretical importance, over the years, many algorithms for solving QAPs have been proposed. In general, instances with $n > 30$ cannot be solved exactly in a reasonable time even on today's high-end platforms. For this reason, the body of work on QAP is dominated by approximation methods, and heuristic and meta-heuristic solutions. The first parallel implementations of QAP were proposed in the early 1990s [1]. The recent rise of GPUs as an integral part of HPC systems has provided the impetus for finding accelerator-based solutions [2].

Although many heuristic algorithms, both sequential and parallel, have been proposed, finding an implementation, that can efficiently solve a variety of problem instances have proven to be particularly challenging. This is because the performance of QAP implementations tend to be highly sensitive to both the size and the *shape* of the input data sets. QAP instances come in many forms. For example, the data can be dense ($flow_{(i,j)}$ defined for all unit pairs (i, j)) or sparse ($flow_{(i,j)}$ defined only for a few units), symmetric ($flow_{(i,j)} = flow_{(j,i)}$) or asymmetric ($flow_{(i,j)} \neq flow_{(j,i)}$) and randomly or non-randomly (i.e., inherent hierarchy) distributed. An implementation can take advantage of one of these properties to quickly find a good solution but can completely collapse for instances where that property does not hold. Of course, in most cases, the issue is not with the specific implementation but rather the algorithm itself. For instance, it has been shown that a genetic algorithm (GA) yields high performance on sparse data sets but its efficiency is substantially diminished when the data set is dense.

Finding good QAP solutions becomes further complicated on GPU architectures. Permutation-based QAPs[1] typically operate on two matrices that contain the flows and the distances. Although the size of these matrices is not prohibitive, the data contained within is needed by each kernel thread and therefore their allocation and access must be carefully orchestrated. The computation patterns in most QAPs resemble those of dense (or sparse) matrix computations. As a result, for best performance both the thread and memory hierarchies must be delicately managed to strike the right balance between occupancy and data locality [3]. These challenges in optimizing QAPs are compounded when we have to account not only for performance but energy efficiency as well.

In this paper, we address this problem by building an autotuning system centered on QAP solvers. We develop two new GPU implementations for solving QAPs. One employs the 2-opt heuristic while the other uses tabu search. Each algorithm is implemented in a manner that allows it to be invoked with execution time parameters that influence its computation and memory access patterns. In addition, we develop a mechanism that lets us select different algorithms, not just implementations, during tuning. To capture the varying effects of input data, we create a database of QAP instances using publicly available data sets and incorporate it into the tuning process. Similar in spirit to MiDatasets[4], the database intermingles data sets to create entries that reflect a wide range of attributes observed in real occurrences of QAP. Our autotuning system explores the extensive space of alternate variants using heuristic search methods. We implement *Pareto-normal* techniques to provide solutions that are optimized for *both* power and performance.

We conduct extensive experimentation on an Nvidia Tesla K20c and produce QAP solvers that execute an *order-of-magnitude* faster than the best known implementations [5]. The best solutions are

[1]most common formulation and the focus of this work

discovered at a low overhead which speaks to the practicality of such a system. The experimental results also provide key insight about different aspects of the tunable search space including, (i) wide variability in performance and power, (ii) non-orthogonality of search dimensions and (iii) the significance of algorithmic choice.

The main contributions of this paper are as follows:

- we provide two new and efficient parallel GPU implementations for QAP
- we develop an autotuning system that exposes suitable tunable parameters and performs multi-dimensional search to discover algorithmic variants that provide portable performance across different data sets
- we include algorithmic choice in the search space; to our knowledge this is the first such effort in the context of GPU-based autotuning
- we demonstrate the non-orthogonality of the search space and discover synergistic points for performance and power; these results are applicable to general autotuning for GPU, beyond QAP

II. Related Work

Much of the work on application tuning for GPUs have focused on manual tuning where expert programmers devise very specific schemes for extracting performance in particular domains [6]. More recently, there has been work in combining automatic and semi-automatic tuning, mostly through source-level directives. Murthy *et al.* developed a semi-automatic, compile time approach for identifying suitable unroll factors for selected loops [7]. Choi *et al.* present a model-driven framework for automated tuning of sparse matrix-vector multiply (SpMV) [8], which yields huge speedups for matrices block sub-structure. Williams *et al.* have also applied model-based autotuning to sparse matrix computation that has yielded significant performance gains over CPU-based autotuned kernels [9]. The MAGMA project has focused on autotuning dense linear algebra codes, successfully transcending the ATLAS model to achieve as much as a factor of 20 speedup on some kernels [10].

The work in this paper distinguishes itself from earlier work in autotuning in three ways: (i) to our knowledge, this is the first attempt at autotuning GPU algorithms within the domain of combinatorial optimization problems, (ii) we propose the inclusion of both algorithmic choice and input data sets within the search space (parameters not included in other GPU-based autotuning systems) and (iii) we consider autotuning for power as well as performance.

III. GPU acceleration of QAP

The goal of this work is to leverage autotuning to explore the performance potential of an extensive search space and *automatically* discover the best solutions. Thus, we elected to implement two fairly simple schemes, making sure that the implementations provide enough flexibility for parameterization.

Our first algorithm uses the 2-opt heuristic, which starts with a random set of feasible solutions, $\mathcal{P}$ and three $n \times n$ matrices containing flows f, distances d, and decision variables x, where $x_{ij} = 1$ if unit i is assigned to location j and 0 otherwise. The cost of the initial solution can then be computed with

$$cost = \sum_{k=1}^{n}\sum_{l=1}^{n}\sum_{i=1}^{n}\sum_{j=1}^{n} f_{kl}d_{ij}x_{ki}x_{lj} \quad (1)$$

The algorithm moves forward by exploring solutions in a neighborhood. To get a single neighborhood solution, two positions, $i, j \in \mathcal{P}$ are randomly selected and a pair-wise exchange of their content is performed. Instead of re-computing the cost using Eq. 1, we apply Burkard and Rendl's formula [14] for computing the *delta* in the objective function, following a swap. This formula which computes the cost in linear time (as opposed to $\mathcal{O}(n^2)$) is given below

$$\begin{aligned}\Delta_{ij} &= (d_{ji} - d_{ij})(f_{\pi_i\pi_j} - f_{\pi_j\pi_i}) \\ &+ \sum_{k\in n\setminus\{i,j\}} ((d_{jk} - d_{ik})(f_{\pi_i\pi_k} - f_{\pi_j\pi_k}) \\ &+ (d_{kj} - d_{ki})(f_{\pi_k\pi_i} - f_{\pi_k\pi_j})) \end{aligned} \quad (2)$$

where π_i is i^{th} permutation.

For the parallel implementation, a set of m initial random permutations of size n is generated and stored in an $m \times n$ matrix, $\mathcal{M}$. Each thread operates on a single row of $\mathcal{M}$ and independently performs all 2-opt interchanges to produce a neighborhood of $n \times (n-1)/2$ new permutations. This neighborhood is examined by computing associated costs using Eq. 2. Once the best value in a neighborhood is found, the process is repeated by each thread as in the sequential case. After maximum iterations is reached, a master thread makes a pass through the best solutions returned by each thread and selects the one with the lowest cost.

Table I
AUTOTUNING SEARCH SPACE

Sub-space	No.	Search parameter	Sample values	Repr.
Compiler Opts.	1	compiler optimization level	O0,O1,O2,03	4 bits
	2	ptxas optimization level	O0,O1,O2,03	4 bits
	3	preserve-relocs	true, false	1 bit
	4	sp-bounds-check	true, false	1 bit
	5	disable-optimizer-constants	true, false	1 bit
	6	allow-expensive-optimizations	true, false	1 bit
	7	fmad	true, false	1 bit
Thread Config.	8	initial solutions	2048	int
	9	threads per block	32	int
	10	blocks per grid	64	int
Other	11	register pressure	64	int
	12	algorithmic variant	*2opt*, *tabu*	
	13	input data set	symmetric, dense	

IV. Constructing the Search Space

Like other GPU codes, performance of QAP solvers is influenced by factors such as occupancy and data reuse. But as we will see later in Section VI, for QAP the performance is impacted at a much higher degree. Moreover, unlike some other domains, QAP performance is highly sensitive to the shape of the input data sets and the particular choice of the solution method. Therefore, to achieve the desired performance it is critical that we expose many different parameters for each algorithmic variant and conduct search along multiple dimensions. To reduce runtime overhead, however, it is also necessary to eliminate portions of the space that are likely to contain mostly bad values. In this section, we describe the parameterization of the search space, provide the rationale

for the inclusion of each search dimension and explain pruning strategies.

A. Compiler optimization

Standard compiler optimizations can influence the performance of QAP codes. Unfortunately, the Nvidia `nvcc` compiler does not make many optimization flags visible to the user. In our framework, we include in the search space all visible compiler and `ptxas` flags from `nvcc` version 6.0 and make provisions for including more optimizations, should more flags be exposed in future versions. Table I lists the flags that form the sub-space of compiler optimizations. This sub-space is represented using a single bitstring. During tuning, the search algorithm selects a point in this sub-space expressed as a bitstring, the bitstring gets converted into the corresponding compiler flags and inserted into the Makefile to build a new variant with the appropriate optimizations. We did not expect the compiler optimization level flags (applied only to the host code) to have much impact but included them for completeness sake. To our surprise, we discovered that in some specific cases these flags can significantly alter the performance of the kernel.

B. Thread configuration

It has been shown that effectively managing the GPU thread hierarchy is instrumental in producing high-performing GPU codes [16], [17]. On one hand, not having enough computation per thread can inhibit parallelism. On the other hand, thread coarsening or block fusion can lead to problems with poor memory reuse. The QAP solvers we implemented follows a fairly simple scheme of task decomposition, where each thread works on a separate instance of the search and all of the computation is done on a single grid. Nevertheless, even for this simple scheme, the choice of the number of threads per block (and consequently, total number of blocks) can have a huge impact on performance, making it necessary to parameterize these attributes.

In our implementations, the total number of threads across all blocks equals the number of initial solutions. The range of the initial solutions in the search space is determined by the quality of the solution produced. Prior studies have shown that fewer than 512 instances can hamper the solution quality while more than 2^{14} instances starts to produce diminishing returns [18]. Based on this result, we set the lower and upper bounds for initial solutions to 512 and 2^{14}, respectively.

The number of blocks is determined by evenly dividing the total number of threads. We also ensure that each block contains threads in multiples of the warp size, otherwise it inevitably leads to inefficient use of GPU resources. The maximum number of threads per block is further constrained by the maximum number of threads allowed per block on the target platform (1024 on Tesla K20c). Thus, the thread configuration is a three-dimensional space that can be expressed as a set as follows

$$\begin{aligned}\mathcal{T} = \{(p,t,b) \mid\ & P_{MIN} \leq p \leq P_{MAX} \text{ and } p \bmod w_s = 0,\\ & w_s \geq t \geq t_{MAX} \text{ and } t \bmod w_s = 0,\\ & b = p/t \ \text{ where } p \bmod t = 0\ \} \end{aligned} \quad (1)$$

where p = initial solutions (permutations), b = number of blocks, t = number of threads, w_s = warp size, t_{MAX} = maximum threads per block in architecture, and P_{MIN} and P_{MAX} are lower and upper bounds for the initial solutions.

In each iteration, the search generates values for p and t that meet the above constraints. The value of b is computed from p and t. The generated values are inserted into the CUDA source by the code restructurer.

C. Register pressure

Although current GPUs provide a large shared register space per block, it has been shown that for some kernels, excessive *register pressure*, the ratio between required and available registers, can cause significant loss in performance. An attribute that has a direct impact on register pressure is thread granularity. Since the number of required registers in a thread cannot be modified arbitrarily, we use the `maxrregcount` flag in `nvcc` to control the number of allocated registers, and thereby the register pressure, in each kernel. The number of allocated registers is always chosen to be a multiple of 16, with a lower bound of four registers per thread and an upper bound enforced by the system (64K per block on K20c).

D. Algorithmic choice

It has been shown that algorithmic choice can be helpful for certain classes of CPU code. But the issue has not been explored in the context of GPUs or in the domain of combinatorial optimizations. To provide algorithmic choice in our tuning system, we develop two algorithms for QAP, as described in Section III. To make the space more interesting, we implement two other variants of 2-opt.

The first variant, *2opt-shared*, attempts to exploit inter-thread data locality. In this variation, portions of the flow and distance matrices that comprise a neighborhood are copied into shared memory and the computation is adjusted accordingly. The idea is to restrict neighborhood explorations by each thread to shared memory, thereby reducing cost calculation time.

For the second variant, *2opt-dyn*, we exploit dynamic parallelism available on the Kepler compute architecture. Since the exploration of a region does not always involve the same amount of work and can lead to unused cycles, we implement a version where each kernel thread starts with an initial solution as before, but then divides the neighborhood into smaller chunks and delegates a child thread to perform a local search within that space. This process has the effect of applying *intensification* within each instance of 2-opt.

We develop a CUDA code template that allows a specific algorithm to be invoked based on a runtime parameter. The I/O, setup and clean-up code are identical for all variants. The only difference is in the kernel function invocation and the parameters that are passed into the kernel. At each step during the tuning process one of the four algorithmic variants is selected and the specific CUDA source is compiled with the requisite parameters.

E. Data sets

To facilitate data-set specific tuning, we build a database of test instances from data made available at QAPLIB [19]. We concentrate on three attributes of the data, (i) density, (ii) symmetry and (iii) distribution. Each data set is assigned a value for each attribute. Each attribute has one of three values: high, low, normal. For instance, for density, high implies dense, low implies sparse and normal implies neither. The ranked data sets are then systematically combined to produce instances exhibiting a range of different characteristics. Our scheme allows for 27 distinct combinations but only 18 could be extracted from QAPLIB data.

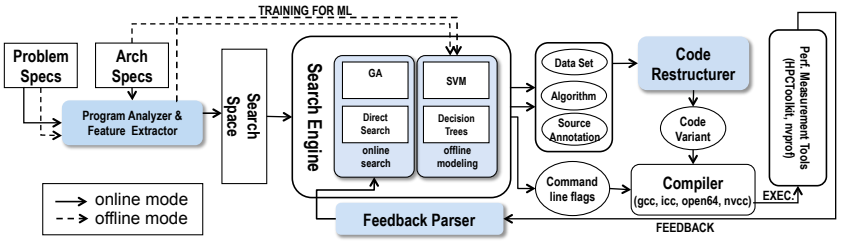


Figure 1. Overview of autotuning framework

During tuning, the code is run with all 18 instances from the database and a simultaneous search is performed to find the best variant in each space. At completion, 18 different binaries are presented to the user to be used with different data sets. Sometimes the same binary works well with multiple data sets, in which case fewer variants are generated.

V. Autotuning Framework

Fig. 1 provides an overview of MATS[2], our autotuning framework. Although the system can be used to autotune other types of codes, here we describe it in the context of tuning QAP solvers written in CUDA only.

MATS can operate in two modes: *offline* and *online*. In the offline mode, machine learning algorithms are developed to characterize program behavior and the learned models are used to select the best variant. We do not utilize the offline mode in this paper[3]. In the online mode, the user provides the location of the CUDA source and the input data. The code is then analyzed and a search space is constructed. One of several heuristic search algorithms is selected to explore this space. During each iteration of the search, a point is evaluated. An evaluation or trial run involves building the CUDA executable using the parameters selected by the search and executing the code on the target GPU. The execution time and average power consumption is recorded during each trial run. The search algorithm uses this feedback to select the next point to be evaluated. This process continues until a pre-specified tuning time limit is reached or the search algorithm converges to a local minima. Next, we briefly discuss the main components of MATS.

VI. Experimental Results

A. Experimental setup

We evaluated our tuning strategy on a Kepler-based Nvidia Tesla K20c which has 13 MPs, each with 192 cores. The board allows 48KB shared memory and 64K registers per block. The GPU is hosted on a six-core Intel Sandybridge server which runs Ubuntu 14.01. All program variants in the experiments are compiled with `nvcc` 6.0.

All input data sets are collected from QAPLIB [19]. Some data sets in QAPLIB are synthetically constructed to simulate specific scenarios while others contain data from real-world instances. The problem sizes range between 20 and 100. The results presented in this section only use the Lipa [20] and Taillard [1] data sets.

For brevity, we use several short-hand terms in the following discussion. *2opt*, *2opt-shared*, *2opt-dyn*, *tabu* refers to the four algorithmic variants included in the search space. Except where noted, *baseline* refers to *2opt*, with 2 blocks, 1024 threads per block, 32 max registers, compiled at level *-O2* with all other optimizations turned off. Cost is the ratio of *trial* runs over *production* runs, expressed as a percentage. We assume 500 production runs in these experiments. Search algorithm refers to a hill climber. None of the search methods had a clear advantage over the others. In the interest of space, we only present numbers from one search.

[2]Model-driven Adaptive Tuning System

[3]use of ML is ongoing work, see Section VII

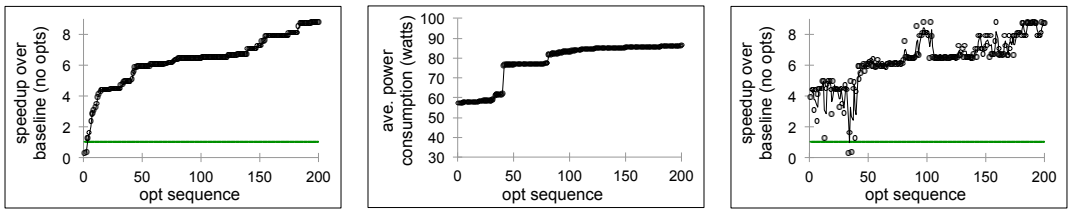
B. Performance variations in search space

We first validate the construction of the tuning space by examining performance patterns within certain sub-spaces. Fig. 2 shows execution time and power variations for 200 randomly selected optimization sequences. For these experiments, we fixed the thread configuration to $(6144, 64, 96)$ and ran the basic *2opt* algorithm. We observe in Fig. 2(a) that even though we are tuning for a small set of compiler optimizations, the search space encompasses a wide span of performance points. The best sequence yields a massive 878% improvement over *baseline* while the worst causes an 80% degradation. Although many sequences offer a decent speedup (about a factor of 6 for sequences 50-150), there are very few really good sequences (speedup > 8). Not surprisingly, some of the best sequences contained the `ptxas` -O3 option but there were several that did not.

Fig. 2(b) shows the average power consumption for the same random sequences, sorted in increasing order of power. The optimizations have less of an impact on power consumption with all points being clustered at three distinct levels. And although the absolute difference between the best and the worst cluster is substantial (20 watts), it does not appear that search is necessary to find the best values. However, a very different pattern emerges in Fig. 2(c) which shows the *speedup values* sorted in increasing order of power consumption. As one would expect, there are many points where high power consumption coincides with high performance. But interestingly, there are several points that produce good speedup without incurring high penalty in power consumption. The wide variability and non-intuitive patterns in performance re-affirm the need for autotuning in this space.

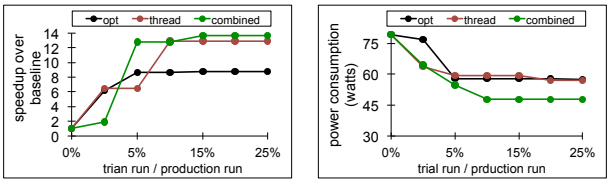
C. Non-orthogonality of the search space

We present experimental results in Fig. 3 that demonstrate that not only are the search spaces for optimizations and thread configurations important but they are also non-orthogonal. Fig. 3(a) shows speedup obtained over *baseline* for three different search methods. In `opt`, we first search the optimization space, find a suitable sequence and then search the thread configuration space whereas



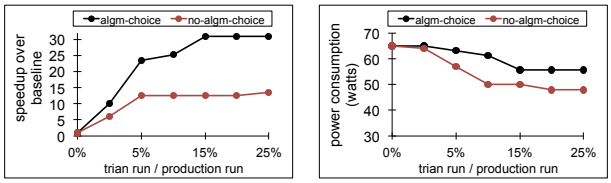
(a) performance (inc. order) (b) power (inc. order) (c) performance (inc. order of power)

Figure 2. Performance and power variations in the compiler optimization sub-space



(a) performance (b) power

Figure 3. Non-orthogonality of search dimensions



(a) performance (b) power

Figure 5. Impact of algorithmic choice

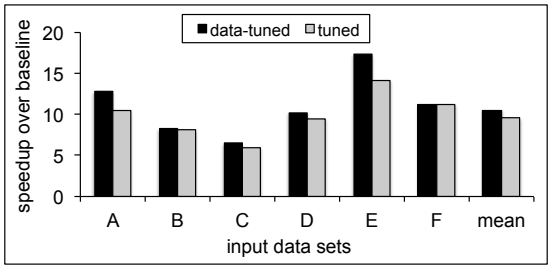


Figure 4. Impact of data-set specific autotuning

in `thread` we do the opposite. For `combined`, we apply a multi-dimensional search to explore both spaces in concert. Settling on a good thread configuration before exploring the optimization space appears to be more profitable. But exploring both spaces together proves most beneficial in the long run.

Fig. 3(b) shows the power consumption numbers for the same three strategies and here the effect of *combined* is more pronounced. Multi-dimensional search produces almost a 25% improvement in power consumption over orthogonal search.

D. Sensitivity to data sets

As mentioned, performance of QAP is substantially impacted by the size and the characteristics of input data sets. In our tuning system, we tackle this problem by producing autotuned binaries for each class of data set. Fig. 4 compares the performance of this strategy (`data-tuned`) with that of the basic version (`tuned`) on six classes of input data. To obtain speedup numbers for data set *A*, we compare the execution of the autotuned variants with that of *baseline* when run with data set *A*. As we can see, `data-tuned` produces significant benefits for some input instances while for others the gain is minimal, with an overall improvement of 10%. In post-mortem analysis, we found that `data-tuned` is particularly effective for data sets that are both sparse *and* symmetric. We are currently investigating the reason behind this.

E. Impact of algorithmic choice

Algorithmic choice provided the biggest gains in our tuning process. Fig. 5 illustrates the impact of algorithmic choice on QAP solutions. Speedup numbers with and without algorithmic tuning are presented in Fig. 5(a). Incorporating algorithmic choice into the search space leads to almost a three-fold performance improvement. Significant gains are achieved at low cost values of $< 5\%$. The improvements in power consumption, as shown in Fig. 5(b), is less dramatic but still significant.

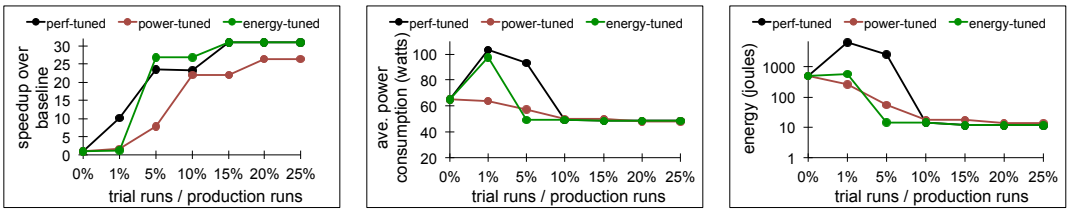
F. Power

Here, we discuss the effectiveness of three approaches to tuning for power and performance. The first approach uses execution time as the feedback, the second uses average power consumption and the third explores the *Pareto-normal* frontier. We refer to these three approaches as *PFT*, *PT* and *ET*, respectively. Figs. 6(a)-6(c) show the effects of these approaches when tuning for performance, power and energy.

When the objective is to obtain the best performance (Fig. 6(a)), both *PFT* and *ET* are able to achieve similar speedup at about 15% cost. *ET* has a slight edge at lower cost values (5-10%), while *PFT* has an advantage at very low cost values (1%). *PT* lags behind the other two all throughout. When tuning for power consumption (Fig. 6(b)), understandably, the situation is reversed. *PT* reaches better values much sooner than *PFT*. *ET* is able to match *PT* for $cost > 1\%$. The situation is very similar when treating energy as the main objective. *PFT* trails both *PT* and *ET* for costs $<$ 10%. Considering, these three scenarios, we conclude that given sufficient number of trial runs all three approaches can eventually find the best values in the search space. When considering the cost of evaluation it is apparent that a Pareto-normal search is the best option, providing the best results at low cost for all three objectives.

VII. Conclusions

This paper presented two new GPU implementations for QAP and described a method for autotuning these implementations for



(a) performance (b) power (c) energy

Figure 6. Tuning for power and energy

both performance and power. The results show that autotuning can provide dramatic improvements in performance over a naive GPU implementation. This performance is only achieved by carefully exploring many different search dimensions including algorithmic variants and input data sets. Furthermore, the tuning strategy can be easily configured to target multiple objectives such as performance, power and energy.

VIII. Acknowledgements

The material is based upon work supported by the National Science Foundation through grant no. CNS-1305302 and a CAREER award no. CNS-1253292. Equipement support was provided by Nvidia.

References

[1] E. Taillard, "Robust taboo search for the quadratic assignment problem," *Parallel Computing*, vol. 17, no. 3-4, pp. 443–455, 1991.

[2] M. Czapinski, "An effective parallel multistart tabu search for quadratic assignment problem on CUDA platform," *Journal of Parallel and Distributed Computing*, vol. 73, pp. 1461–1468, 2013.

[3] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proc. of the 2008 ACM/IEEE conf. on Supercomputing*, 2008.

[4] G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam, "Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization," in *Proc. of the 2nd Int'l Conf. on High Performance Embedded Architectures and Compilers*, 2007, pp. 245–260.

[5] W. Zhu, J. Curry, and A. Marquez, "SIMD tabu search for the quadratic assignment problem with graphics hardware acceleration," *Int'l Journal of Production Research*, vol. 48, no. 4, pp. 1035–1047, 2010.

[6] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proc. of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008.

[7] G. Murthy, M. Ravishankar, M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for gpgpu programs," in *IEEE Int'l Symposium on Parallel Distributed Processing*, 2010.

[8] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in *Proc. of the 15th ACM SIGPLAN symposium on principles and practice of parallel programming*, 2010.

[9] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Comput.*, vol. 35, no. 3, pp. 178–194, 2009.

[10] R. Nath, S. Tomov, and J. Dongarra, "Accelerating GPU kernels for dense linear algebra," in *Proc. of 9th Int'l Meeting on High Performance Computing for Computational Science*, 2010.

[11] R. Miceli, G. Civario, A. Sikora, E. Csar, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, and F. Bodin, "Autotune: A plugin-driven approach to the automatic tuning of parallel applications," *Applied Parallel and Scientific Computing*, vol. 7782, pp. 328–342, 2013.

[12] A. Davidson and J. Owens, "Toward techniques for auto-tuning GPU algorithms," in *Applied Parallel and Scientific Computing*. Springer, 2012, vol. 7134, pp. 110–119.

[13] L. Yixun, E. Z. Zhang, and X. Shen, "A cross-input adaptive framework for GPU program optimizations," in *Proc. of the 2009 IEEE Int'l Symposium on Parallel&Distributed Processing*, 2009.

[14] R. Burkard and F. Rendl, "A thermodynamically motivated simulation procedure for combinatorial optimization problems," *European Journal of Operational Research*, vol. 17, no. 2, pp. 169–174, 1984.

[15] Z. Drezner, "A new genetic algorithm for the quadratic assignment problem," *Informs Journal on Computing*, vol. 15, no. 3, pp. 320–330, 2003.

[16] A. Magni, C. Dubach, and M. F. P. O'Boyle, "A large-scale cross-architecture evaluation of thread-coarsening," in *Proc. of the 2013 ACM/IEEE conf. on Supercomputing*, 2013.

[17] S. Unkule, C. Shaltz, and A. Qasem, "Automatic restructuring of GPU kernels for exploiting inter-thread data locality," in *Proc. Int'l. Conf. on Compiler Construction (CC12)*, 2012, pp. 21–40.

[18] A. Chaparala, C. Novoa, and A. Qasem, "A SIMD solution for the quadratic assignment problem with GPU acceleration," in *3rd Annual XSEDE Conf.*, 2014.

[19] "QAPLIB - a quadratic assignment problem library," http://anjos.mgi.polymtl.ca/qaplib/, 2014.

[20] Y. Li and P. Pardalos, "Generating quadratic assignment test problems with known optimal permutations," *Computational Optimization and Applications*, vol. 1, pp. 163–184, 1992.