Anytime Algorithms for GPU Architectures

Rahul Mangharam[†‡] and Aminreza Abrahimi Saba[†]
[†]*Dept. of Computer and Information Science*
[‡]*Dept. of Electrical and Systems Engineering*
University of Pennsylvania
{*rahulm, aminreza*}*@seas.upenn.edu*

***Abstract*—Most algorithms are run-to-completion and provide one answer upon completion and no answer if interrupted before completion. On the other hand, anytime algorithms have a monotonic increasing utility with the length of execution time. Our investigation focuses on the development of time-bounded anytime algorithms on Graphics Processing Units (GPUs) to trade-off the quality of output with execution time. Given a time-varying workload, the algorithm continually measures its progress and the remaining contract time to decide its execution pathway and select system resources required to maximize the quality of the result. To exploit the quality-time trade-off, the focus is on the construction, instrumentation, on-line measurement and decision making of algorithms capable of efficiently managing GPU resources. We demonstrate this with a Parallel A* routing algorithm on a CUDA-enabled GPU. The algorithm execution time and resource usage is described in terms of CUDA kernels constructed at design-time. At runtime, the algorithm selects a subset of kernels and composes them to maximize the quality for the remaining contract time. We demonstrate the feedback-control between the GPU-CPU to achieve controllable computation tardiness by throttling request admissions and the processing precision. As a case study, we have implemented AutoMatrix, a GPU-based vehicle traffic simulator for real-time congestion management which scales up to 16 million vehicles on a US street map. This is an early effort to enable imprecise and approximate real-time computation on parallel architectures for stream-based time-bounded applications such as traffic congestion prediction and route allocation for large transportation networks.**

I. Introduction

Performance scaling of single-thread processors stopped in 2002 and has fueled the use of multicore Graphics Processing Units (GPUs) which have been growing in transistor count by 65% annually. The current generation of NVIDIA's Fermi GPU consists of 512 cores and is capable of executing 24,576 concurrent thread kernels for efficient stream processing. GPUs will, in the year 2015, use 11nm technology and contain around 5,000 cores, which should render them capable of around 20 Teraflops [1]. In the past few years, the GPU has evolved into an increasingly convincing computational platform for non-graphics applications [2]. Our goal is to investigate time-bounded algorithms on GPUs to effectively trade-off execution time with output quality. This will enable a large class of data-dependent and dynamical applications with a large number of variables to leverage the high-throughput concurrent computation of GPUs. Algorithms in this category include real-time estimation, prediction and decision making in weather science, nation-wide traffic management and electronic trading.

As such data-dependent applications are increasingly subject to larger workloads, it is often not possible to compute results in real-time. Furthermore, it is often better to have an imprecise or approximate result within a time-bound rather than a precise answer that is outdated. Algorithms for such applications must therefore adaptively allocate resources appropriate to the goals and loads and evaluate their progress with execution time to deliver intermediate results. Such results may be in the form of estimated values, a range of answers or measurements.

This paper presents an early investigation of time-bounded algorithms on the NVIDIA CUDA General Purpose GPU programming platform [3], [4], [5]. The Compute Unified Device Architecture (CUDA) provides a programming model for general purpose programming on GPUs. The interface uses standard C code with parallel features. Using CUDA, we investigate the design of time-bounded algorithms that continually measure the quality of the output with the remaining time and dynamically adjust their execution path to optimize the outcome of the computation by a specified deadline. Such *Anytime Algorithms* allow for approximate and imprecise computation with an output quality metric that is monotonic with the available execution time [6], [7]. The focus of this investigation is on the construction, instrumentation, runtime progress measurement and adaptive selection of execution paths for anytime algorithms on GPUs.

A. Anytime Algorithms

Anytime Algorithms may be considered in two variations: *Interruptible* algorithms and *Contract-time* algorithms. Interruptible algorithms can be interrupted at *any* time to deliver the best result obtained so far. They are convenient to use as the results may be requested in an on-demand manner. Contract-time algorithms are specified an execution time a priori and decide on the best strategy to maximize the quality of the output within that duration. Contract algorithms are often more intuitive to design, and typically use simpler data structures and control structures, making them easier to implement and maintain. Our focus is on contract-time algorithms for soft real-time stream processing on GPUs. In general, Anytime Algorithms have four properties:

1. Quality & Execution Time trade-off -To measure the runtime performance, anytime algorithms generally introduce a quality function which is a monotonic function of the amount of time available to the algorithm. It is therefore necessary to construct a performance profile of output quality at design-time and select a set of operating points along the

trade-off curve at runtime. A task is monotone if the quality of its intermediate result does not decrease as it executes longer. Such monotone tasks are available in many problem domains, including numerical computation, statistical estimation and prediction [7], heuristic search [8], sorting, and database query processing [9]. In this study, we will consider the parallel execution of a Parallel A* search algorithm that is executed with a specified contract time.

2. Performance Predictability - Anytime Algorithms must expose control or tuning knobs to traverse the quality-time tradeoff. Such knobs may be in the form of sampling rates and iterative improvement functions which affect the quality in terms of metrics such as accuracy, coverage area, certainty, and resolution (level of detail). Quality-time tradeoffs may be achieved by several techniques such as milestone method, sieve functions and selecting between multiple versions of the algorithm [10]. With the *milestone method*, the algorithm executes for a minimum duration and then evaluates its progress at checkpoints within the control-flow graph. Based on the remaining time, the algorithm may decide to execute both the mandatory and optional operations, or just the mandatory operations. *Sieve functions* allow for computation steps to be skipped such that a minimum sampling may be achieved over a shorter duration. Alternatively, the application may be implemented with *multiple versions* of the same algorithm such that computationally intensive implementations may be swapped with quicker but less precise versions. For each of the above techniques, it is necessary that alternative execution approaches have the facility to measure the quality with explicit metrics in the current mode. Languages such as Flex provide language primitives with which the programmer can specify the intermediate result variables and error indicators, as well as the time instants to record them [11], [12].

3. Correctness measure - As the execution pathway of an anytime algorithm is a function of the workload and application goals, it is only determined at runtime. Thus, the outcome may be among a range of possibilities, and it is essential to ensure that outcome is correct. Consider the example in Fig. 1, where the application has a contract-time of 11s. At runtime, the algorithm may choose among any of the alternate execution paths, where paths on the left are more computationally intensive (time consuming) but more accurate than those to the right. In this example, the algorithm chooses path $1A$ which consumes 8.2s. As the remaining time is relatively small for the rest of the execution, the algorithm decides at intermediate checkpoints to execute less accurate and faster paths in later phases (i.e. $2C$ followed by $3D$) to finish by the deadline. In order to check the correctness of combinations of alternate execution paths, one approach would be to verify if the string $1A{-}2C{-}3D$ is a subset in the language of acceptable sequences. More generally, anytime algorithms require a *result evaluator* to ensure the correctness during the course of execution.

4. Suspend and Resume Execution Anytime algorithms must have the capability to be interrupted either at anytime or at pre-determined checkpoints to output an intermediate result. In addition, they must be able to continue operation using intermediate results.

B. Anytime Algorithms for Parallel Computing

Anytime algorithms have been largely studied on sequential and single execution-path architectures. While early investigations have explored multi-processor architectures [13], our efforts are with parallel GPU computing architectures. This introduces three differences from earlier approaches: (a) The algorithm must be mapped spatially across multiple processing resources (grids, blocks and warps) and this introduces platform-dependent variations in the quality-time tradeoff; (b) Alternative implementations of the algorithm must be compiled into kernels at design time and appropriate kernels must be composed to select an operating point along the quality-time curve; (c) The algorithm specifies the execution of a single thread and the kernel executes hundreds to thousands of threads concurrently. Thus, it is necessary to wait until all threads complete in one kernel till the next kernel is loaded. A primary challenge of kernel composition with the current GPU architectures is to execute similar threads within one kernel and restrict the set of concurrently executing kernels on the GPU.

Our framework has four key elements: (a) profiling the algorithm by timing analysis to partition execution across multiple exploration and exploitation phases, (b) instrumentation for on-line measurement of the progress of the algorithm and the remaining time, (c) interfacing control knobs (or control dimensions) which trade-off output quality with time so that the algorithm can adapt its behavior along those dimensions when needed, and (d) policies for runtime selection of the most appropriate execution path based on (a) and (b).

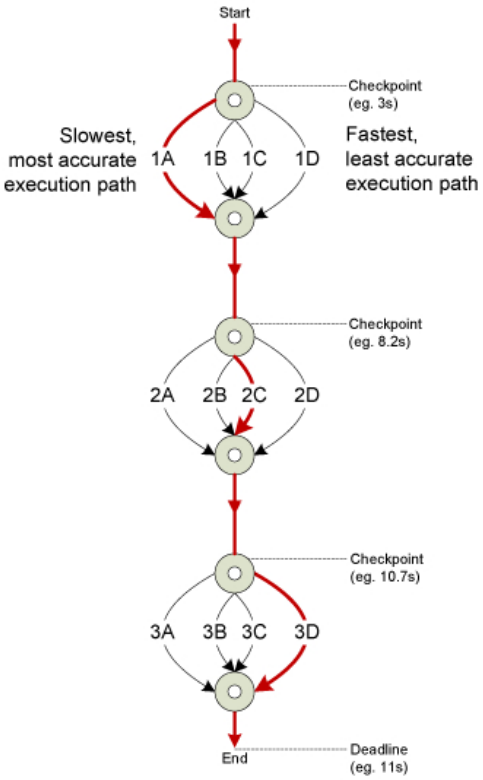


Figure 1. Anytime algorithm with multiple optional paths.

C. Quality and Resource Profiles

To profile and explore the quality-time trade-off, we employ a framework similar to Quality of Service Resource Manager (Q-RAM) [14]. Q-RAM is an optimization framework to maximize the output quality, given multiple resources and multiple control knobs (control dimensions). The quality-time and resource-time profiles are computed at design-time and pruned by extracting the convex hull of the quality-resource-time profile for an application. We apply this meta-approach to time-bound computation of Parallel A* searches executed in parallel (PAP*) across a mesh graph on a GPU. Our current study only explores a single resource (time) and single quality metric (map resolution) as our goal is to stream output for a large number of vehicles continually as the weights of the graph change due to congestion. Other quality metrics such as route update rate and the incremental length (e.g. 0.5, 1, 3km) for which the route must be calculated, may be included.

D. Organization

This paper is organized as follows: The next section provides a brief background on the GPU architecture and timing analysis for sample algorithms. Section 3, describes the construction of the PAP* algorithm followed by its quality-time profiling and runtime kernel composition. Section 4 presents performance results of the anytime implementation of the PAP* algorithm and its effectiveness in exploiting the quality-time trade-off. We conclude with a case-study followed by a summary of the related work and future directions.

II. General Purpose GPU Timing Analysis

In recent years graphics processing units (GPUs) have provided us inexpensive highly parallel computation power [2], [5]. Harnessing this power was relatively difficult until GPU architectures started to support general purpose programming languages (e.g. C, C++, Python) in addition to graphics APIs [3], [4], [15]. So instead of expressing algorithms in terms of programmable shaders, they can be constructed using more general programming models like CUDA.

A. CUDA GPU Architecture

CUDA provides parallel programming extensions to the C programming language. It gives the programmer direct access to the virtual instruction set and explicit control of the memory in the device. The programmer specifies the CPU and GPU functions in terms of thread and thread-block layouts. The finest unit of execution in CUDA is one thread. Threads are grouped into blocks and blocks are organized into a grid (Figure 2). Each multiprocessor executes one block at a time, and each block runs on just one multiprocessor. The CPU and GPU run different code. The CPU executes the main program and sends tasks to GPU in form of *kernel functions*. While there can be many kernel functions defined in a program, there can be only one kernel running on the device at any point in time. This restriction may be relaxed with newer GPUs.

In our measurements, from the CPUs perspective, kernel invocation endures small overhead (<0.12ms). Thus, in the rest the experiments we ignore this overhead by repeating the experiments 100 times on the device with just one invocation and then divide the measured time by 100. This way, the overhead is amortized and excluded from our measurements.

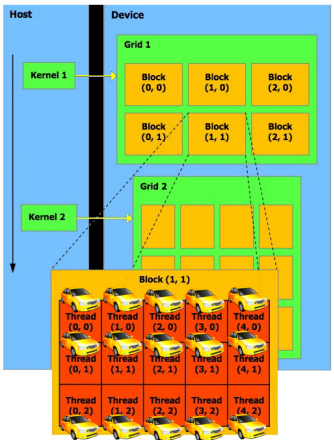


Figure 2. CUDA programming model: showing CPU (host) to GPU (device) communication. Kernels are mapped to grids on the device, which execute blocks of threads concurrently.

B. An Example of GPU WCET Measurement

We now discuss the specifics of Worst Case Execution Time (WCET) measurement for an example CUDA program for vehicle routing across a street map. For the timing analysis, we use the AutoMatrix vehicle traffic simulator (described later in the case-study section) which executes on the GPU. This step is required to construct the quality and execution time profile of an algorithm at design-time. Figure 3(a) depicts the high-level Control Flow Graph (CFG) which is largely the decision paths a vehicle takes when it reaches an intersection. The left side of the control flow graph, which consists of the outer and inner loops, depicts the *worst-case* execution path for the program. The outer loop determines the high level routing for all the vehicles in the grid, while the inner loops calculate the finer routing details for each vehicle in the grid. The execution time for the program is largely influenced by the behavior of these two loops, since most of the time is spent performing these calculations. Hence, there is a need to obtain good estimates of the WCETs for both of these loops.

We achieve this by instrumenting the code and observing the clock during the instrumentation points. The CUDA architecture has a special register, named $\%physid$, that keeps track of the multiprocessor on which the current thread is executing. Each multiprocessor has its own individual clock that can be easily read from the kernel using the $clock()$ system call. The instrumentation checkpoints (timestamp capture) are shown in Figure 3(a) by the red triangles placed at various points along the CFG. Each instance of the inner loop was forced to execute for eight iterations since each vehicle's routing has at most eight road segments to choose from at an intersection. The outer loop was run for 100 iterations to capture the average and worst case. When we plotted the behavior of the application for a varying number of iterations of the outer loop we noticed that a run of 100 iterations captured most of the vehicle routing scenarios.

C. Timing results

Fig. 3(b) shows the execution times for the inner loops for each thread that executes on the CUDA processor. While the graph shows a fair distribution of the execution times for the inner loop (largely due to map data fetches from global memory on the device), the results are fairly well bounded within 0.5ms. Hence this can be considered the WCET profile for the inner loop. As was the case for the inner loop, we find an upper bound for the execution of the outer loop is approximately 3.6ms. Fig. 3(c) shows the absolute execution times from the start of the kernel for all warps. A warp is a group of 32 threads scheduled concurrently on a single stream processor. The execution profile of the thread closely follows that of the outer loop, except that the time taken is two orders of magnitude higher. The execution times for the complete thread is dominated by the the times for the outer loop. We obtain an upper limit for the execution times for entire program to be 349.4ms. Since we schedule 850 threads at a time on the CUDA processor (this is a constraint of this particular processor model), the first batch takes at most 349.4ms, the second takes a further 349.4 ms, and so on. Using this approach, we can instrument the entire application and extract timing measurements (see Fig. 3(d)) for both design-time analysis and runtime adaptation.

We adopt the same measurement technique for the Parallel A* search algorithm we use as a case-study for the remainder of this paper.

III. PAP* Parallel Search Algorithm

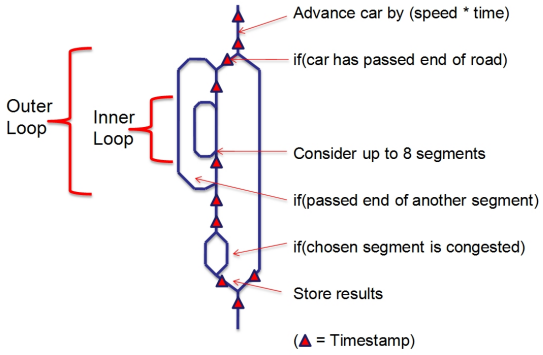
We step through the construction of a Parallel A* algorithm that is capable of trading-off execution time for output quality. To enable the contract-time capability, the algorithm must expose one or more control knobs (control dimensions) such that the execution time and quality are profiled for all operating points at design time. At runtime, the algorithm must be capable of adjusting the operating point based on the remaining time and goals of the application. This case study provides a simple walkthrough of this process. A fundamental limitation of the CUDA device is that each of its multiprocessors follows a SIMD architecture. This limits the efficiency for algorithms with divergent thread processes. Our goal is to adapt the A* algorithm to run efficiently on GPU given these limitations. The adaptation of this algorithm to CUDA was inspired by the work done in [16] and [17].

A. Graph representation

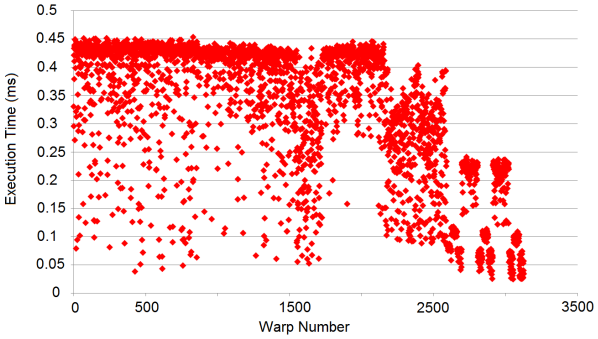
Graphs are commonly represented by their adjacency matrices. However since we are mostly dealing with sparse graphs of street topologies, such a representation will be very wasteful in terms of memory space. We therefore store the graph in memory using adjacency lists which are more compact. Adjacency lists of the vertices of each graph are packed into a single array. In addition, different graphs are packed together into a single compact adjacency list. So on a higher level it may look like we a have a graph with several connected components, i.e. a jungle. Each query is confined to one of these connected components. Each vertex stores the starting index of of its adjacency list in the global compact adjacency list. Vertices of all graphs $G_i(V_i, E_i)$ are represented as an array V. An array E stores the adjacency lists of all vertices of all graphs. Each entry of the edge array E refers to a vertex in vertex array V.

B. Parallel A* searches in Parallel (PAP*)

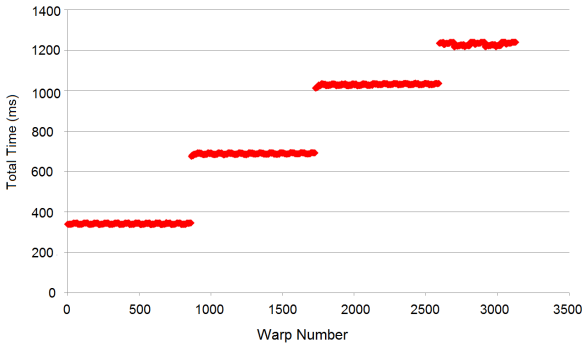
In PAP*, the goal is to run multiple instances of the A* search algorithms while each of them can take advantage of the parallel nature of the GPU. The problem is thus: given



(a) Structure of example code showing instrumentation points



(b) Measured Execution Time for the Inner Loop



(c) Execution Time from Beginning of Kernel

Piece Measured	Maximum Time (ms)
Beginning	0.122
One Inner Loop	*(0.454)*
Eight Inner Loops	3.631
Outer Loop	*(3.483)*
Middle	0.168
Last Branch	2.618
End	0.020
Total single thread WCET	**6.558**

(d) Total execution time of instrumented code sections

Figure 3. GPU Worst Case Execution Time Measurement

a weighted directed graph $G(V, E, W)$ with non-negative weights, and a set of source vertices S, and a set of the corresponding destination vertices D, find the shortest path from each vertex in S to its corresponding destination in D.

C. CUDA implementation of PAP*

In our implementation we use a set of arrays for each of the graphs we have: a vertex array V, and edge array E, boolean masks F and C of size $|V|$ which record the search frontier and the finalized nodes, and also a weight array W of size $|E|$. Also we have a cost array C which keeps record of the shortest path from the source to the expanded nodes.

Each thread on the device is assigned to one node of a graph (see Fig. 4). In each iteration, each vertex checks to see if it is in the frontier list. If yes, it fetches its current cost from the cost array C and its neighbors weights from the weight array W. The cost of each neighbor is updated if it is greater than the cost of the current vertex plus the edge weight to that neighbor (relaxation). At the end of the execution of the kernel, a second kernel compares cost C with updating cost. It updates the cost C only if its cost is higher. The updating cost array reflects the cost array after each kernel execution for consistency.

The second stage of kernel execution, shown in Fig. 4, is required as there is no synchronization between CUDA blocks. To find the minimum cost vertices in the search frontier of each graph we first find the minimums per CUDA block and then find the global minimums in a separate kernel execution.

Another decision that should be made in the runtime configuration of the kernels is the way the threads are organized into blocks and the grid. The next section briefly explains the idea.

D. Multiple parallel queries

For the experiments in this paper, we use fixed and identical queries with the source and destination being the two opposite corners of the mesh (worst case), while the edge weights are chosen randomly and are different in different meshes. There are different modes in which each query can be run. For example, we can run a single query with maximum parallelism (maximum number of blocks). Another option is to pack multiple queries together and run them together as if we are running one query on a larger graph. The implementation structure of the PAP* allows us to do this on a single graph or multiple graphs seamlessly. This decreases the amount of parallelism as seen by each individual query (see Fig. 5), but as will be explained later, it might increase the overall performance of the system, as there might be more parallelism to exploit in multiple queries than in a single one.

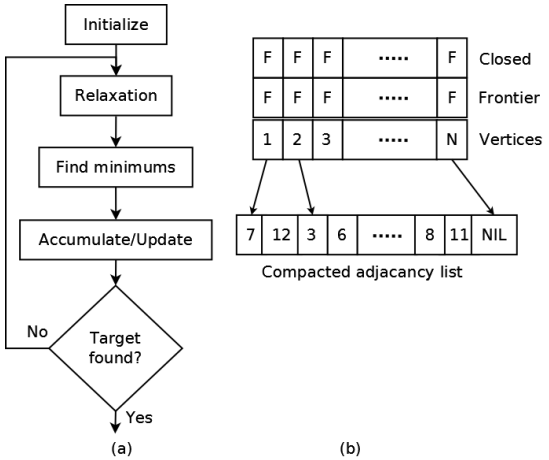


Figure 4. PAP* consists of four CUDA kernels. Three of them, relaxation, find minimums, and accumulate/update run in a tight loop until the target node is found.

IV. PAP* - Control Dimensions

For the query processing routine to adapt to the available time, we provide degrees of freedom along which the query processor can change the operating point in the algorithm to trade-off route accuracy and execution time.

A. Platform Independent Control Dimensions

In this case study, we have used a regular mesh graph, along with three lower resolution versions of it ranging from 1024 nodes down to 16 nodes, as shown in Fig. 6. Level 0, is the original graph and Levels 1, 2, and 3 are lower resolution meshes derived from the original graph. These abstractions speed up execution and provide partial results in lieu of a complete path. The higher level meshes hide the details of the graph and thus shrink the search space. Lower levels with more details lead to a larger search space, but provide more accurate search results. In this experiment, we assigned scores 6, 4, 2, and 1 for routes calculated in the layers 0 to 3 respectively. These scores are application dependent but generally illustrate that the more detailed paths are assigned higher scores to maintain monotonic quality with increasing processing. For each search query we consider these different versions of that graph. It is possible to change the level at runtime. The experiments are repeated for a number of runs to distinguish the measurements from the noise.

B. Platform-dependent Control Dimensions

The PAP* algorithm is very flexible in terms of mapping execution to the thread-block level architecture on the device. We observe that for the GPU, architecture-specific constraints provide a large dynamic range for effectively scheduling threads for a given workload. It can be run with different thread/block numbers given any search graph size. This becomes a problem of optimization parameters to schedule parallel searches together on the device. Fig. 7, shows the variation in execution time of a single A* search with

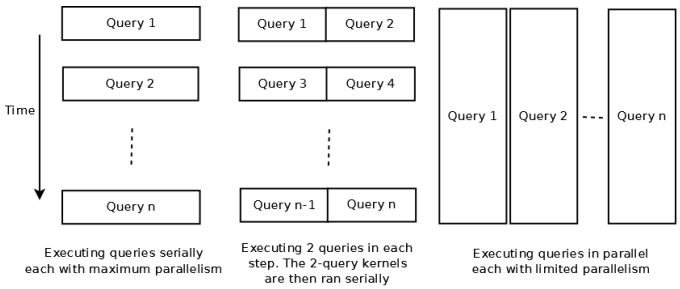


Figure 5. Merging queries together and running them together in parallel versus running maximally parallelized single queries, serially.

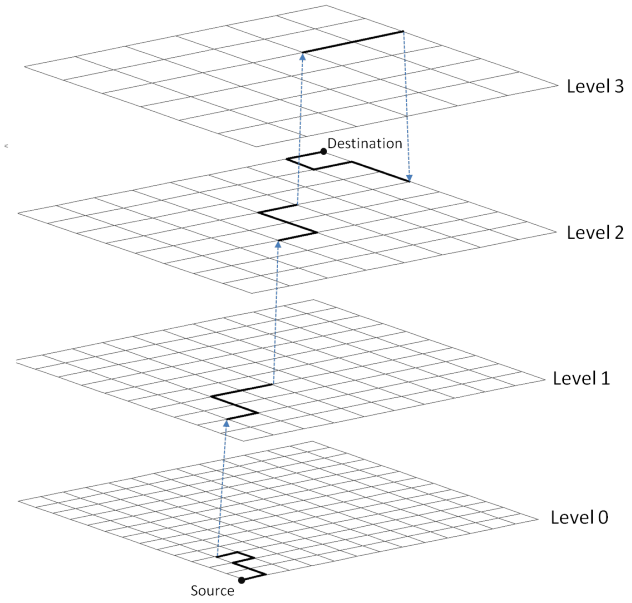
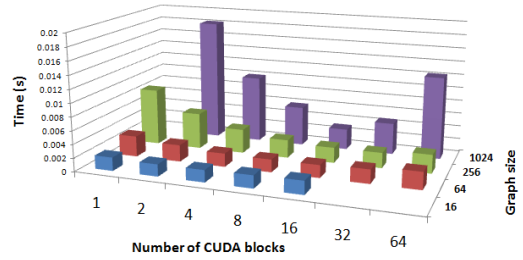


Figure 6. Multi-level graph: we maintain 4 different layers for the graph that we are using. Each higher layer represents a coarser level approximation of the lower level graph. This figure is not to the scale and is just for demonstration of the idea.

different graph sizes and blocks numbers. The sweet spot for a single isolated search is when we have as many blocks as possible, provided that the number of threads on each multiprocessor doesn't go below the number of cores per multiprocessor. In this figure the number of multiprocessors is 27, with 8 cores per multiprocessor. Thus, spreading the computation to maximally use all multiprocessors while running at least 8 threads per multiprocessor allows for the highest throughput for this algorithm. Fig. 8 is a plot of the algorithm's execution time using different number of blocks. Here the number of threads per multiprocessor would be number of vertices divided by the block count.

C. Kernel Composition

For an anytime application, a pool of kernels with fixed time-quality trade-offs are aggregated in the host at design-time. At runtime, based on the load and quality-level goals of the application, the appropriate kernel is selected on-demand and queued to execute on the device. As shown in Fig. 9(a), when we compose a kernel in this manner from several smaller kernels, the running time of the composed kernel is equal to the sum of the running times of the kernels in



	1	2	4	8	16	32	64
16	0.002	0.0018	0.0018	0.0019	0.002		
64	0.0031	0.0025	0.002	0.0019	0.0019	0.0021	0.0026
256	0.0085	0.0054	0.0036	0.0027	0.0023	0.0023	0.0028
1024		0.0182	0.01	0.0059	0.0031	0.0047	0.0123

Figure 7. PAP* execution time for the multilevel graph of Fig. 6.

the longest path from start to completion (critical path). So, on a device that has M multiprocessors, a kernel composed of M parallel 1-block search kernels takes the same amount of time to complete as a single 1-block search. Thus, given the information in Fig. 7 the best way to run 32 searches is to compose a new kernel from 32 1-block searches, rather than running 32 32-block searches sequentially. However, if we have just a single query then we are better off running a single kernel with 32 blocks.

Each of these searches is one CUDA kernel and the CUDA device doesn't allow us to run more than one kernel at a time. So, to run multiple kernels (e.g. 1-block A* searches) concurrently, we would have to make a new kernel that can effectively emulate multiple parallel kernels at runtime, while from the viewpoint of the device there is just one kernel executing [18].

In this scheme the number of blocks is equal to or less than the number of multiprocessors on the device. So, to assign blocks of a kernel to different multiprocessors, we can use block indices. So we'll have a nested $IF \ldots THEN \ldots ELSE$ structure that runs specific code based on the block ID. We don't assign more than one kernel to a single block as far as possible. More specifically, the overall structure of each thread looks like the following pseudo code:

Algorithm 1: Kernel composition example.

```
input: K_i(.);
bid ← blockIdx.x;
if bid< K1.size then
    K1();
    K2();
    if bid< K3.size then
        K3();
    else if bid≥ K3.size then
        K4();
        K5();
    end
else if bid≥ K1.size then
    K6();
end
```

So, assuming that $K_i.size$ is the number of blocks that kernel $K-i$ requires, blocks $0, 1, \ldots, K1.size-1$ run kernel K1

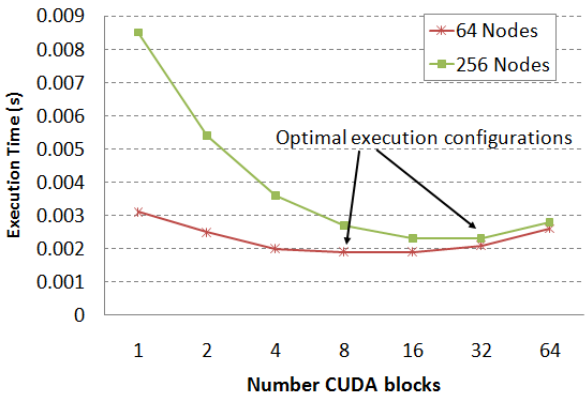
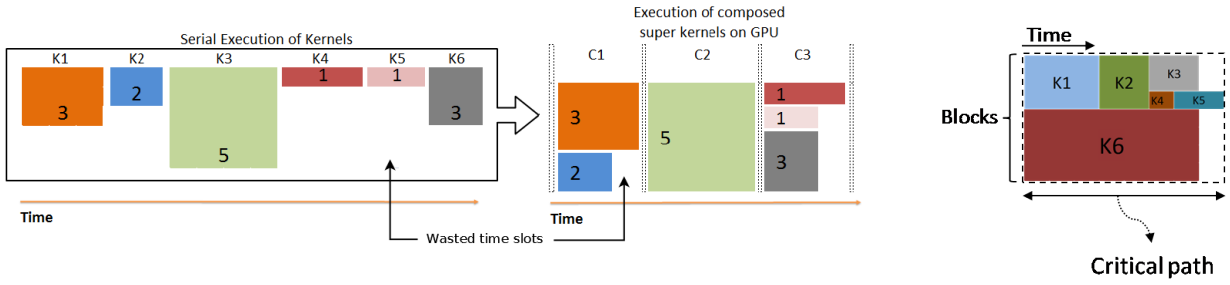


Figure 8. PAP* execution time for different block numbers



(a) (b)

Figure 9. (a) Each block represents one kernel queued on the host. Kernels are executed sequentially. The same kernels are then combined together to form composed kernels that emulate simultaneous execution of kernels. (b) Demonstration of critical path in a composed kernel.

and then K2 and then blocks $0, 1, \ldots, K3.size-1$ run kernel K3, and so on. Fig. 9(b) demonstrates how these base kernels are packed together. So we have effectively made a new kernel that contains at its core six basic kernels. Creating combinations of kernels incurs a static compilation cost plus a negligible constant run-time cost. So at any point of time we can invoke one of these six kernels, or any kernel composed using a subset of them. Using these combinations increases the flexibility of the scheduling. We can run kernels in parallel to increase device utilization when a single kernel cannot guarantee high occupancy of the device.

V. Anytime PAP* Evaluation

A. *Offline scheduling: quality vs. execution time trade-off*

Based on the timing information for different operation modes as summarized in Fig. 6 and Fig. 7, we developed a dynamic CUDA thread-kernel scheduler which selects parameters for different queries based on the available time, to maximize the overall query response quality. We have designed a scheduler that given the total amount of time available, finds the best scheme to combine and schedule the required kernels to run the queries. This scheduler, named `offline`, creates batches of queries and runs all queries in that batch in one of the levels of the multilevel graph.

However, due to variations in actual kernel execution times and those estimated at design time we need to be able to adapt to the variation in the available time. So, after each batch completion, the scheduler optimizes the current schedule given the actual amount of time left to finish the queries. Fig. 10 shows the quality versus contract time for scheduler `offline`, along with the overall quality score that static schedules can achieve, while running a total of 6,144 search queries. The static level scores are consistently lower than the `offline` scheduler as the scheduler cannot adapt and misses the deadlines for queries with contract time below the minimum time required for the specified level of the graph. `offline` finds the optimal graph levels and composition to run the kernels so as to meet the deadline while maximizing the overall quality of results.

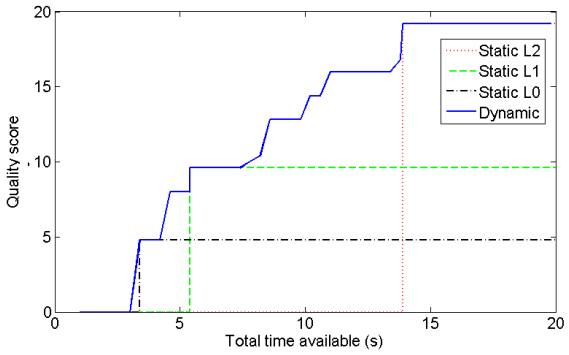


Figure 10. The *offline* scheduling algorithm can change the execution parameters of the query execution routines to adapt to the available resources, and maximize the overall quality of the query responses.

B. *Online Scheduling with Quality Control*

Since the scheduler might create very large kernels in its optimization of the quality score, variation in actual kernel execution times can be large. So if for some reason one of the batches takes much longer than was anticipated by the scheduler, then the deviation from the optimal plan might be so large that forces us to unduly decrease the quality of the subsequent batches which will hurt the overall quality of the results. So the scheduler will have more flexibility if it had the option of modifying a schedule while it is running. We design a new scheduler, named `online` that can do this. If the scheduler detects that a query is taking longer than the projected amount of time, it can adapt to the new situation and make sure that the system will succeed in responding to all queries in a timely manner while having a larger search space to optimize the schedule. For this purpose, `online` probes the the system continually and determines when the projected time bounds have been violated by some predefined margin. In that case, a new schedule is composed based on the new time bounds, and put into effect immediately. So, it can change the execution time of a query batch even after it has been dispatched by the host CPU by changing the *level* in the graph from the current location on the map.

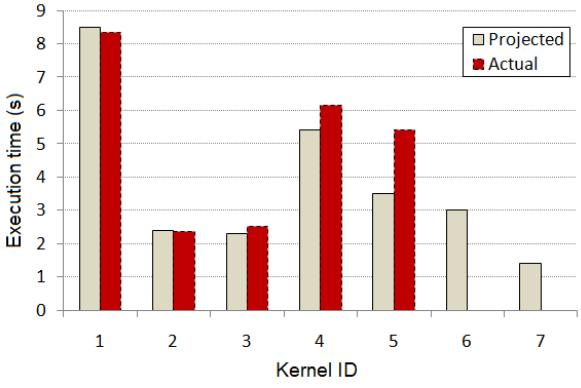
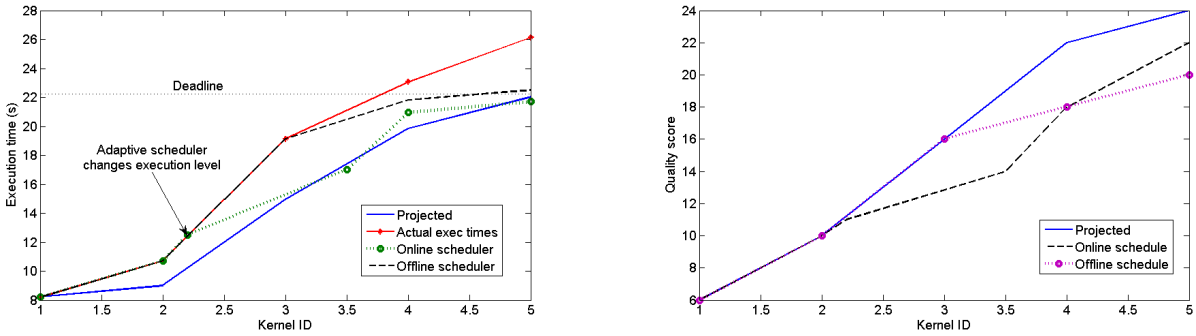


Figure 11. Projected versus the actual execution time of the 7 sub-kernels to execute three batches of 6,144 queries each.



(a) (b)

Figure 12. (a) Execution times by *offline* and *online* schedulers. (b) Quality scores of schedules composed by *offline* and *online* schedulers.

Fig. 11 shows the projected and the actual execution times for three query batches each consisting of 6,144 queries. We have manually forced the actual execution of the kernels to take longer to demonstrate the ability of the system to adapt to new time bounds. Fig. 12(a) shows the accumulated execution times of kernels running in the order proposed by the scheduler. As can be seen in Fig. 12(a), when dispatching kernel 3, `offline` detects that we are short of time based on the actual execution times, we make a new execution plan that finishes the query processing within the given time bounds. However, `online` detects the situation while running kernel 2 and changes the execution plan accordingly to meet the time bounds. As we expect, the sooner we detect an execution plan that has gone awry the better we can fix it because more time is left and we have more candidate execution plans to choose from. Thus as can be seen in Fig. 12(b), `online` has been able to score higher than `offline`.

VI. Online Scheduling with Feedback Control

In the previous section we described the approach to trade-off application-level quality for execution-time. In several classes of applications the response time is a primary measure of service quality. Consider for example a stock market application where users request portfolio updates as well as make buy/sell transactions. In this case, if the server is overloaded it is better not to process additional requests to guarantee a low maximum response time. In this overload case, the server refuses service to a fraction of requests to maintain a tardiness bound [19]. Similarly, in the case of the GPU we have developed a feedback-controller in the CPU which periodically measures the processing time and quality of the process on the GPU and accordingly throttles the admission of new requests.

We have implemented a method where a Proportional Integral (PI) controller executes on the CPU in closed-loop operation with the GPU (see Fig. 13). For a given task, we can set the quality set-point and tardiness set-point and the controller will throttle the input request rate. Consider the example in Fig. 14, where we request route queries to the mesh-based PAP* routing engine on the GPU. In this setup, routing queries can request a minimum quality level for each route. In other words, if a request wants a detailed map on Level0 and Level1 only, then it will result in heavier processing. In the figure, after the system has settled to the first phase of stream queries, there is a sudden spike in the number of queries and a majority of requests now request heavier processing per request. As the system is now overloaded, you can observe the tardiness overshoots for a few cycles, and then the controller fixes the situation by changing the set-point of the load.

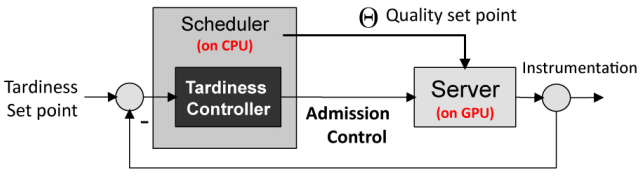


Figure 13. PI control loop executes on the CPU to throttle requests on the GPU so the desired tardiness may be maintained on the GPU

From the PI controller's perspective (see Fig. 15), the controller initially settles on a set-point of 39 requests per time step. After the spike in the request load, the controller throttles the request rate to settle at 31. In this experiment the setpoint for tardiness is 0.9, Proportional controller gain is 0.3 and the Integral controller gain is 0.01. We can observe that even though the controller and plant execute on two separate processors, the response time for the applications of interest are both effective and controllable to finer granularity. In this example, we use tardiness as opposed to deadline miss rate as the applications of interest are both soft real-time and a continuous metric is more useful than a binary one. If we employ a deadline miss as the metric, we observe in Fig. 16 the deadline misses initially settle down to about 10% and then temporarily rises to a peak of 50% during the onset of the spike and settles to the minimum miss rate shortly afterwards. In general, the execution time is a stochastic process at runtime and statistical approaches to adaptive control would be more effective. The example shown is to

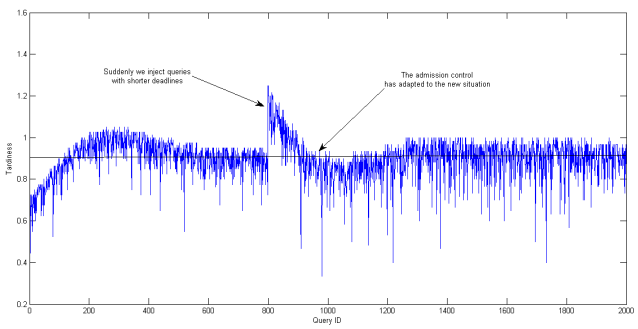


Figure 14. Feedback-based admission control maintains tardiness set-point of 0.9. There is a spike in the request rate at the 800th request. The controller restores tardiness by throttling the admission.

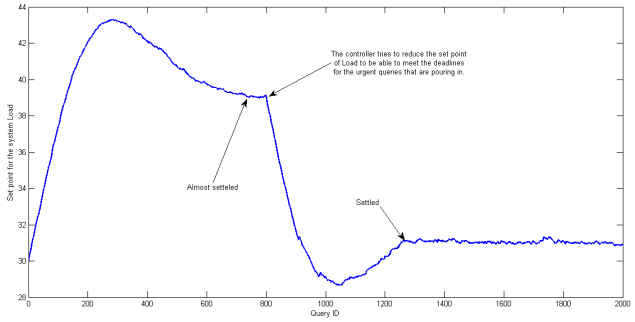


Figure 15. The PI controller's adapts its set-point for admission rate after there is a spike in requests

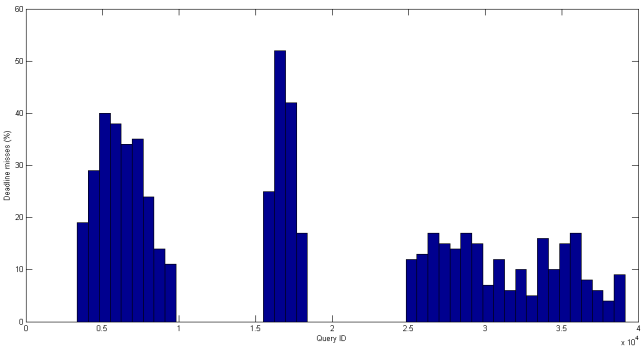


Figure 16. The deadline miss rate for routing requests during initialization, a request spike and after settling to a steady state

demonstrate a feedback control knob to anytime algorithms on the GPU. We thus observe that both quality and admission control can be combined in a single feedback controller to provide effective soft real-time performance on GPUs.

VII. Case Study: AutoMatrix Traffic Congestion Simulator

We have developed AutoMatrix, a vehicle traffic congestion simulator to further explore anytime algorithms on GPUs (see Fig. 17). AutoMatrix is implemented in CUDA and can simulate over 16 million vehicles on any US street map while operating faster than wall-clock time. It is a microsimulation such that each vehicle is represented by a thread. Periodically all vehicles report their speed, position and direction so that the speed weight may be determined for each street segment. We use the fundamental traffic flow diagram [20] to determine the weight of each segment based on the number of vehicles on that segment. Vehicles are well-behaved in the sense that they use a car-following algorithm and specify random walk or origin-destination routing requests. The goal of AutoMatrix is to explore real-time traffic prediction and fastest-path routing based on the current state of the street network.

AutoMatrix implements the parallel A* routing algorithm and is able to route vehicles with hierarchal routing and adaptive routes. In the former case, vehicles can request for a coarse-grained route based on a current snapshot of the network at the initial time. Once the vehicle begins the journey, the street network weights are periodically updated (e.g. every 5sec) and an incremental adapted route for a short distance ahead (e.g. 1Km) is computed. As the vehicle traverses the network, it continually receives updated routes based on the current traffic conditions. We observe in Fig. 17(left), a microsimulation of 800,000 vehicles in Washington D.C. representing the congested segments in red color. In Fig. 17(middle), a single vehicle has been assigned a course-grained path from North Philadelphia to downtown. In this and the next case, there are 500,000 vehicles randomly walking to provide background traffic. We also observe a short 1Km fine-grained path from the vehicle's current position that is computed every 5sec to provide an adaptive route based on real-time traffic congestion. In Fig. 17(right) we observe hundreds of PAP* routed vehicles with fine-grained adaptive routes being updated every 5sec.

We are currently using AutoMatrix to explore resource-adaptive routing that can scale to hundreds of thousand of vehicles. As all the above experiments utilize synthetically generated traffic, we plan to use the real-time traffic data from INRIX to execute on the AutoMatrix setup.

VIII. Related work

The term "Anytime algorithm" was introduced by Dean and Boddy[6] in their work on time-dependent planning during the late 1980's. Horvitz[7] introduced the flexible computing model for time-critical decision making and planning algorithms in Artificial Intelligence. Following this, several studies in the AI community focused on composing anytime algorithms to more complex systems for sensor interpretation and path planning[21], [22], search[8], and evaluation of belief networks [23]. In the early 1990s, the real-time community explored scheduling approaches for imprecise and approximate computing[10], [13]. The Flex language was developed with time as a first-class citizen to specify timing and performance requirements within the algorithm[11], [12]. This contributed to approaches for performance polymorphism[24] such that mandatory and optional computations may be selectively executed based on the available execution time. These efforts were seldom evaluated on computing platforms or applied to complex problems.

Within the computer architecture and programming languages communities, general techniques for execution time speed-up have been proposed. Loop perforation[25], increments loop iterators to globally expedite computation while trading accuracy in return for (speed) performance. This approach executes every n-th iteration and does not require customized or surgical modification of the algorithm. While the class of applications this can be applied to is limited, loop perforation has been used to speed-up video and audio encoders, Monte Carlo simulations, and machine learning. The results show that the transformed applications can run as much as two to three times faster than the original applications while distorting the output by less than 10%. Other approaches, such as PetaBricks, offer runtime algorithm choice [26], [27], [28] where multiple implementations of multiple algorithms are selectively exploited to solve a problem is the natural way of programming. Choices also include different automatic parallelization techniques, data distributions, algorithmic parameters, transformations, and blocking. These approaches do not adhere to any contract-time requirements and are equivalent to a 'better than best effort' speedup approach in trading-off execution time for

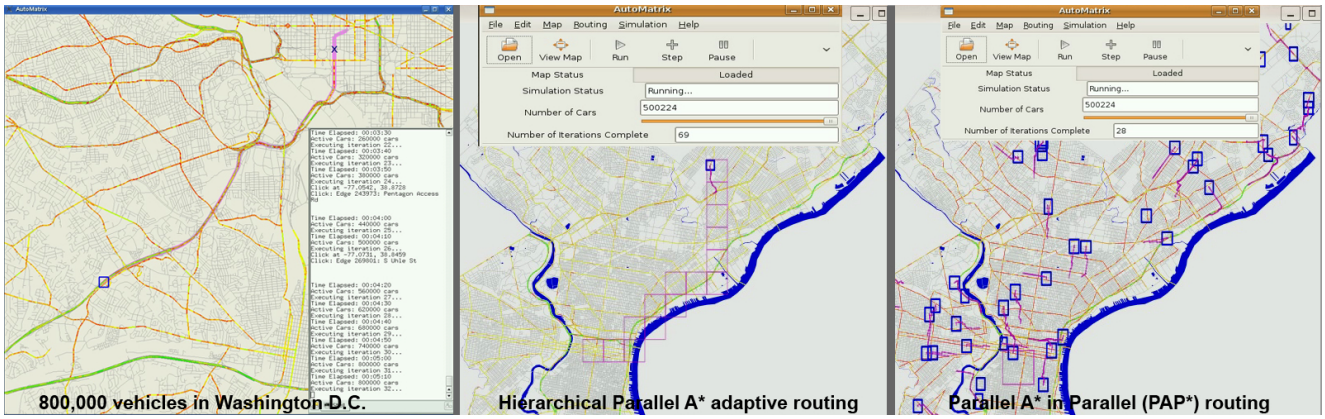


Figure 17. AutoMatrix CUDA-based vehicle traffic simulator showing scalable, hierarchal and adaptive parallel routing for real-time congestion management

quality. More recently, [18] used static compilation techniques to create efficient kernel implementations for multiple tasks. Our approach builds on this for runtime adaptation to contract time constraints.

IX. Conclusion

We present early efforts in time-bounded algorithm execution on a GPU's parallel architecture. The focus of this work was on the construction of anytime algorithms which effectively trade-off output quality and admission rate for execution time to provide approximate results within a contract-time. We present the construction, instrumentation, on-line measurement and runtime scheduling of such anytime algorithms. Through a case study with Parallel A* search, we demonstrate the feasibility and effectiveness of the proposed approach. For a case-study, we developed AutoMatrix, a GPU-based vehicle traffic simulator that can scale to over 16 million vehicles. We use AutoMatrix to explore implementations of the PAP* algorithm on real street maps. We have currently considered feedback-based quality-controlled and admission-controlled parallel algorithms and plan to extend our efforts to auto-tuning algorithms. This effort presents a new approach to real-time computing on very parallel processor architectures.[1]

References

[1] EETimes. NVIDIA to EDA: Give us power tools , 2009.
[2] S. Che and et. al. A performance study of general-purpose applications on graphics processors using CUDA. *J. Parallel and Distributed Comp.*, 68(10):1370–1380, 2008.
[3] NVIDIA. CUDA Programming Guide 2.1. 2008.
[4] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 2008.
[5] J. Nickolls, I. Buck, and M. Garland. Scalable Parallel Programing with CUDA, 2008.
[6] M. Boddy and T.L. Dean. Solving Time-dependent Planning Problems. *Joint Conf. on AI*, pages 979–984, 1989.
[7] E. J. Horvitz, H. J. Suermondt, and G. F. Cooper. Bounded Conditioning: Flexible inference for decision under scarce resources. *Workshop on Uncertainty in AI*, 1989.
[8] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime Search in Dynamic Graphs. *Artif. Intell.*, 172(14):1613–1643, 2008.
[9] J. Grass, S. Zilberstein, and E. Moss. Anytime Database Algorithms. *U. Amherst, Tech. Report.*, 1995.
[10] J. W. S. Liu and et. al. Algorithms for Scheduling Imprecise Computations. *Computer*, 24(5):58–68, 1991.
[11] K-J. Lin and S. Natarajan. Expressing and Maintaining Timing Constraints in Flex. In *Proc. Ninth IEEE Real-Time Systems Symp*, pages 96–105, 1988.
[12] K. B. Kenny and K. J. Lin. Building Flexible Real-Time Sys. using the Flex Language. *ACM Computer*, 24(5):70–78, 1991.
[13] J. Y-T. Leung. A Survey of Scheduling Results for Imprecise Computation Tasks. 1995.
[14] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A Resource Allocation Model for QoS Mgmt. *IEEE RTSS*, 1997.
[15] A. Munshi. OpenCL - Beyond Programmable Shading Course. *SIGGRAPH*, 2008.
[16] Pawan Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU using CUDA. In *HiPC'07: Proc. Conf. on High Perf. Computing*, pages 197–208, 2007.
[17] P. J. Martín, R. Torres, and A. Gavilanes. CUDA Solutions for the SSSP Problem. In *Proc. Intl. Conf. on Computational Sc.*, pages 904–913, 2009.
[18] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. Enabling Task Parallelism in the CUDA Scheduler. *Workshop on Programming Models for Emerging Architectures*, 2009.
[19] Y. Zhou and KD Kang. Deadline Assignment and Tardiness Control for Real-Time Data Services. In *22nd Euromicro Conference on Real-Time Systems (ECRTS '10)*, 2010.
[20] A. May. Traffic Flow Fundamentals. In *Prentice Hall, NJ*, 1990.
[21] S. Zilberstein and S.J Russel. Anytime sensing, planning and action: A Practical model for robot control. *Joint Conf. on AI*, pages 1402–1407, 1993.
[22] S. M LaValle. Planning algorithms, 2004.
[23] M.P. Wellman and C. L. Liu. State-Space Abstraction for Anytime Evaluation of Probabilistic Networks. *Conf. on Uncertainty in AI*, 1994.
[24] K. B. Kenny. *Structuring Real-Time Systems using Performance Polymorphism*. PhD thesis, 1990.
[25] M. Rinard and et. al. Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures. In *MIT Tech. Report MIT-CSAIL-TR-2009-042*, 2009.
[26] G. F. Diamos and S. Yalamanchili. Harmony: an Execution Model and Runtime for Heterogeneous Many Core Systems. In *Proc. High Perf. Distr. Comp.*, pages 197–200, 2008.
[27] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ASPLOS XIII*, pages 287–296, 2008.
[28] J. Ansel and et. al. PetaBricks: A Language and Compiler for Algorithmic Choice. In *ACM SIGPLAN Conf. Programming Language Design and Impl.*, 2009.

[1] An earlier version of this effort was presented at the AVICPS Workshop, 2010 (at RTSS) but was not published.