Anytime Algorithms for Multi-core Architectures

Aminreza Abrahami Saba† Sibin Mohan‡ Rahul Mangharam†
†Dept. Electrical & Systems Engineering
University of Pennsylvania
{aminreza, rahulm}@seas.upenn.edu
‡Dept. of Computer Science
University of Illinois at Urbana Champaign
sibin@cs.uiuc.edu

***Abstract*—Our goal is to investigate the construction, instrumentation and scheduling of time-bounded and anytime algorithms on multi-core architectures such as graphics processing units (GPUs). 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 monotonically increasing utility with the length of execution time. Such imprecise and approximate computing has wide application in prediction algorithms in the domains of vehicle traffic congestion, stock price prediction and weather prediction where a large number of variables and dynamical states must be considered to periodically stream an output. Our investigation focuses on time-bounded anytime algorithms on GPUs for real-time vehicle traffic congestion prediction and route assignment. To explore this, we have designed AutoMatrix, a traffic congestion simulation platform on the Nvidia CUDA-enabled GPU. AutoMatrix is capable of simulating over 16 million vehicles on any US street map and executing traffic estimation, prediction and route assignment algorithms with high-throughput. This research has the potential to extend real-time scheduling on massively parallel GPU architectures to attack a variety of data-driven, interactive and dynamical algorithms with timely operation.**

***Keywords*-Cyber-Physical Systems, real-time systems, timing analysis, GPGPU, CUDA, anytime algorithms, traffic modeling**

I. Introduction

Performance scaling of single-thread processors stopped in 2002 and has fueled the use of multicore GPUs which have been growing in transistor count by 65% annually. For example, the current generation of Nvidia's Fermi GPU consists of 512 cores capable of executing 24,576 concurrent thread kernels for efficient stream processing. Nvidia's graphics circuits 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. The goal of the proposed research is to investigate real-time and time-bounded execution on GPUs. 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. Applications in this category include real-time estimation, prediction and decision making algorithms in weather science, nation-wide traffic management and algorithmic trading.

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 remaining time and dynamically adjust their execution path to optimize the outcome of the computation by a specified deadline. Such *Anytime Algorithms* [2] allow for approximate and imprecise computation with a utility that is monotonic increasing with the available execution time. We focus our investigation in the context of vehicle traffic congestion management where, given origin-destination information of each vehicle, the goal is to periodically stream the fastest routes for millions of vehicles based on the current state of the traffic network. According to the 2007 Urban Mobility Report [3], delays due to traffic congestion cost the nation $78 billion in the form of 4.2 billion lost hours and 2.9 billion gallons of wasted fuel. One of the key strategies recommended by the 2009 Urban Mobility Report is to provide choices to drivers such as alternate paths via on-line traffic information based on probing and estimating the near-term congestion.

A. Anytime Algorithms for Parallel Computing

As the algorithms of interest are largely data-dependent where the real-time constraints are not known a priori and the optimization algorithms improve on the result incrementally, a framework is needed to allow algorithms to adapt to the available time. 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. Our framework has four key elements as follows: (a) profiling the algorithm to partition execution across multiple exploration and exploitation modes, (b) instrumentation for on-line measurement of the progress of the algorithm and the remaining time, (c) interfacing degrees of freedom into the algorithm so that it can adapt it's behavior along those dimensions when needed, and (d) policies for runtime selection of the most appropriate execution path on (a) and (b).

We apply our investigation of anytime algorithms to an in-depth case study on real-time traffic congestion prediction and route assignment using the AutoMatrix traffic simulation GPU platform. The performance of the anytime algorithms

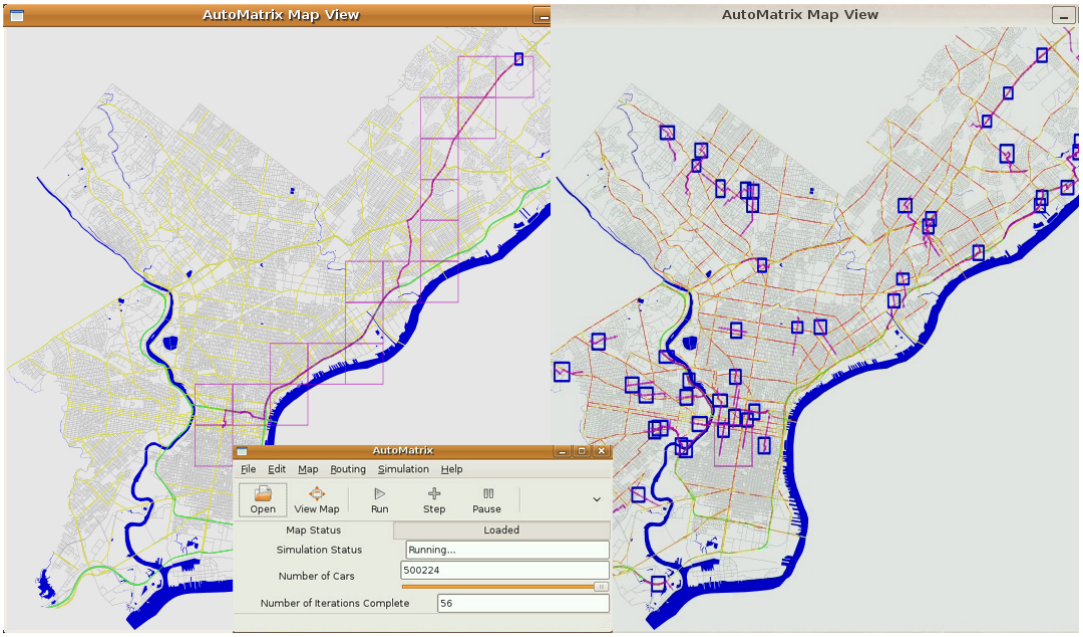

Figure 1: AutoMatrix traffic simulator with 500K vehicles executing on the Nvidia CUDA platform. (a) Shows hierarchal A* routing in coarse and fine granularity. (b) A* routing for vehicles with unique origin and destination pairs.

are measured by the quality of the fastest travel path computed and the error between the computed estimated travel time and the actual travel time from the origin to destination.

II. AutoMatrix GPU-based Traffic Simulation Platform

As preliminary work for the proposed research we developed AutoMatrix (Fig. 1), a hybrid traffic simulator capable of simulating over 16 million vehicles on any US street map. AutoMatrix executes on the Nvidia GPU platform using the CUDA API and is capable of executing algorithms for traffic estimation, prediction and route assignment. Vehicle mobility is modeled by first-order Lighthill-Whitham-Richards (LWR) model [4] and is also extensible to integrate off-line and on-line traffic data. The primary purpose of AutoMatrix is to execute parallel algorithms for real-time traffic prediction and capacity-constrained All-Pairs-Shortest-Path (APSP) vehicle routing with various optimizations such as minimum delay, min-max delay distribution, even-use of road network resources, and for minimizing the error between actual travel time and estimated travel time. AutoMatrix includes a basic set of traffic congestion models, routing algorithms, trip models and mobility models.

A. Congestion Modeling

The LWR traffic model provides the fundamental equation for *free flow* traffic speed as the traffic density increases. On the other hand, *traffic incidents* (i.e. accidents, construction, detours) are responsible for over 35% of the overall travel time delay. AutoMatrix supports point-based congestion (See Fig. 2) and also allows us to impose certain weather conditions in specific areas, and see the effect on the rest of the network. Point-congestion can be used to analyze and find the bottlenecks of a network. It can also be used for congestion planning by simulating traffic scenarios that would result from specific long-term urban construction plans or short-term detours.

Planned blockades: We can manually mark a set of road segments as blocked and define the corresponding detours. Also AutoMatrix can automatically suggest optimal detours for a set of defined point congestions. This can help both in design of transportation networks and rerouting traffic in exceptional circumstances.

Traffic hotspot identification: Given a network and a traffic induction scenario, AutoMatrix can find traffic hotspots by analyzing street diversity along heavily loaded paths. High levels of street diversity with a good mix of different road types facilitates a larger number of equivalent alternate paths and better resilience to traffic congestion.

Impact of weather condition: AutoMatrix can also analyze traffic behavior in a network after imposing a specific weather condition on an area. This exhibits itself as the changes in conduction speed of the road segments, and probability of occurrence of traffic incidents.

B. Routing Algorithms

Currently AutoMatrix structures the street map as a graph and supports three routing algorithms. Here we briefly explain each of them from a high-level view point.

1) Adaptive Routing: After computing the initial route for a source/destination pair, AutoMatrix updates the route over time based on the traffic density distribution across the

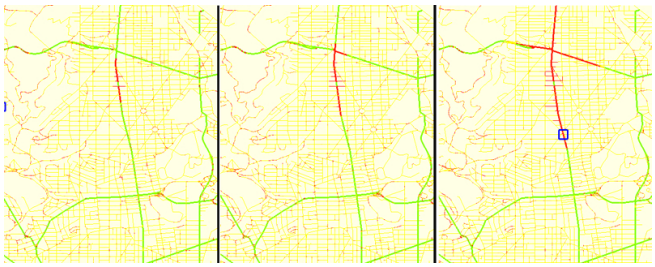

Figure 2: Traffic incident modeling with backlog

street map graph. Thus, the routed vehicles can avoid newly created congestions zones and hotspots.

2) Hierarchical Routing: In general, road networks are very large graphs (e.g. Washington D.C. contains over 800K segments) and computing shortest path for a large number of cars is computationally very demanding. In hierarchical routing, we first make a coarser estimation of the network and perform all-pairs shortest paths (APSP) on it, then the routes in this network are translated into routes in the actual network by local A* searches (as seen in Fig. 1(a).

3) Parallel A searches in Parallel (PAP):* The PAP algorithm can run thousands of A* search instances in a parallel manner on NVIDIA GPUs. This enables AutoMatrix to scale routing for tens of thousands of cars in real-time. AutoMatrix combines PAP with hierarchical routing. Thus we first calculate APSP on the GPU in parallel and then local A* searches are done using PAP. This framework allows us to move from a smaller number of large searches to a larger number of smaller searches and vice versa. Having a large number of light-weight queries is generally more favorable as it allows for more adaptive scheduling of the tasks.

C. Performance Evaluation

For any instance of routing, AutoMatrix calculates an estimated travel time (ETT) and measures the actual travel time (ATT). Thus, we are able to evaluate how well can a specific routing algorithm adapt to congestions, and in general, changes in the traffic conditions of the network (sensitivity analysis). Small ATT/ETT error indicates the ability of the routing algorithm to predict future travel times along a route based on projected traffic conditions.

III. GPU Timing Analysis

As instrumentation and on-line performance and timing measurement are key building blocks toward anytime algorithms, we have conducted extensive dynamic timing analysis of AutoMatrix on CUDA. The CUDA architecture [5] abstracts away the internal details of the processors, such as pipeline structure, functional units, register table, *etc.* This coupled with the fact that there are no guarantees provided about whether these internal structures will retain their design across generations of the CUDA platform means that the task of performing accurate timing analysis becomes difficult and the process of static timing analysis becomes un-viable. Hence, we must resort to using *dynamic techniques* in a smart manner.

A. Control Flow behavior of AutoMatrix

Figure 3 depicts the high-level structure of the AutoMatrix program. These are largely the decision paths a vehicle takes when it reaches an intersection. The left side of the control flow graph depicts the "worst-case" path for the program. This section consists of two loops, the *inner* and *outer* loops. 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 worst-case execution times (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 system calls (named "*clock()*" in this case). Hence, by observing this register and the clock values from time to time, on the worst-case path, we can capture the execution times along that part of the control flow graph (CFG). The instrumentation (timestamp capture) points are shown in Figure 3 by the red triangles placed at various points along the CFG. The clock is sampled at these points and the deltas are calculated to obtain the execution times for those code segments.

Each instance of the inner loop executed for *eight* iterations since each vehicle's routing has at most eight road segments to choose from. The outer loop was run for 100 iterations. When we plotted the behavior of AutoMatrix for a varying number of iterations of the outer loop we noticed that a run of 100 iterations captured most of the simulations.

B. Early Timing Results

Figure 4 shows the execution times for the inner loops for each thread that executes on the CUDA processor. While the

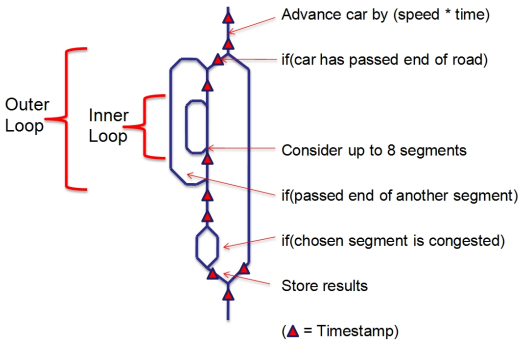

Figure 3: High Level Structure of AutoMatrix Code showing Instrumentation Points

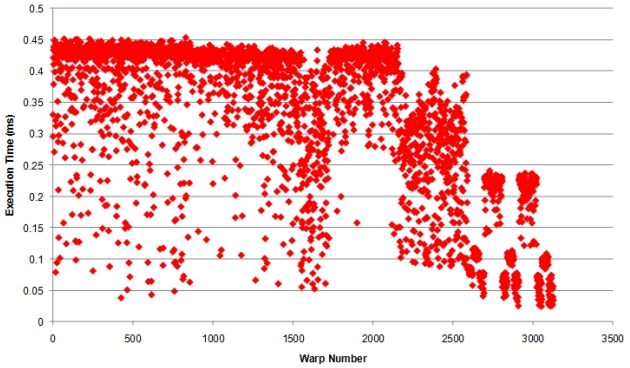

Figure 4: Measured Execution Time for the Inner Loop

graph shows a fair distribution of the execution times for the inner loop, the results are fairly well bounded within 0.5 ms. Hence this can be taken to be the worst-case execution time profile for the inner loop.

As was the case for the inner loop, we can find an upper bound for the execution of the outer loop as well: approximately 3.6 ms. Figure 5 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 bound for the execution times for entire threads: 349.4 ms. Since we schedule 850 threads at a time on the CUDA processor, the first batch takes at most 349.4 ms, the second takes a further 349.4 ms, and so on.

All experiments were conducted on an NVidia GeForce GTX 260 card. It contains 216 1.2 GHz cores and has a peak throughput of 805 GFlops. We simulated $100,000$ cars under the forced worst-case scenario presented in section III-A. The $100,000$ cars were split into $3,125$ *warps* of 32 threads each. As mentioned before, the total WCET for batches of 850 wraps is 349.4 ms. Hence, the WCET for individual threads: 6.558 ms.

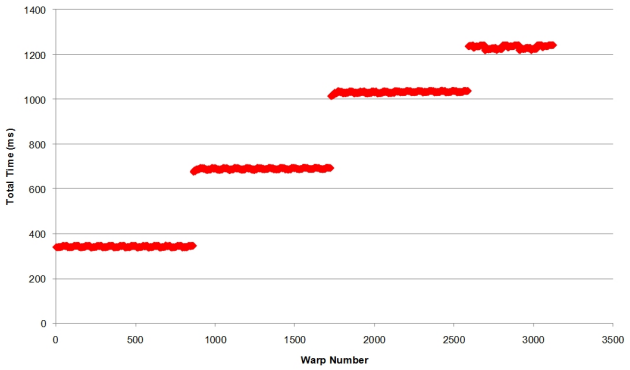

Figure 5: Execution Time from Beginning of Kernel

IV. Research Plan

As part of our efforts for creating and using anytime algorithms on general purpose GPU's, we intend to execute along these research directions:

1) *Algorithm Profiles*: It is necessary to profile algorithms in terms of exploration/exploitation phases, scatter/-gather operations and determine the best mapping of state to constant, shared and global memory. This allows the algorithm to determine the default block sizes, overall number of threads and work per thread.
2) *Runtime Diversity*: In order to exploit the different degrees of freedom within the algorithm it is essential to identify or construct algorithms with a large dynamic range of execution times across different execution paths. In search algorithms, we plan to accomplish this by using hierarchal search, adaptation based on the length of the look ahead path from the current position, route re-computation interval and search across different road types.
3) *Timing Analysis*: of such algorithms to find their true behavior, resource requirements, *etc.* Obtain a parameterized mathematical transformation of the algorithm based on effects on running time. A true challenge is to translate these mathematical transformations into timing constraints and vice-versa
4) *Instrumentation*: Devise methods for runtime performance and time measurement to identify the best runtime operating mode.
5) *Kernel Scheduling*: Develop mechanisms and policies to switch between operating modes to optimize operation for the remaining time. With CUDA, we are currently restricted to scheduling kernels at runtime. [6], [7] describe one method for run-to-completion algorithms to obtain better throughput.
6) *Optimization*: Devise techniques and *utility* metrics to determine the best mode of operation for the remaining time. Construct a multi-objective multi-resource quality trade-offs of the algorithm along the lines of the QRAM [8].

References

[1] EETimes. Nvidia to EDA: Give us power tools , 2009.
[2] M. Likhachev et. al. Anytime Search in Dynamic Graphs. *Artificial Intelligence*, 172(14), 2008.
[3] T. Lomax and D. Schrank. Urban Mobility Report. *Texas Transportation Institute*, 2007.
[4] B. Kerner. The Physics of Traffic. *Springer*, 2004.
[5] NVidia CUDA architecture. http://www.nvidia.com/object/cuda_home_new.html.
[6] M. Guevara, C. Gregg, K. Hazelwood, and K. Skadron. Enabling Task Parallelism in the CUDA Compiler. In *Proc. Workshop on Prog. Models for Emerging Arch.*, 2009.
[7] J. Nikolls, I. Buck, and M. Garland. Scalable parallel programming with cuda. 2008.
[8] C. Lee, J. Lehoczky, R. Rajkumar, and D. Siewiorek. On Quality of Service Optimization with Discrete QoS Options. In *IEEE RTAS*, 1999.