BLITZCRANK: Factor Graph Accelerator for Motion Planning

Yuhui Hao*
Tianjin University
yuhuihao@tju.edu.cn

Yiming Gan*
University of Rochester
ygan10@ur.rochester.edu

Bo Yu
BeyonCa
bo.yu@beyonca.com

Qiang Liu†
Tianjin University
qiangliu@tju.edu.cn

Shao-Shan Liu
BeyonCa
shaoshan.liu@beyonca.com

Yuhao Zhu
University of Rochester
yzhu@rochester.edu

***Abstract*—Factor graph is a graph representing the factorization of a probability distribution function and serves as a perfect abstraction in many autonomous machine computing stacks, such as planning, localization, tracking and control, which are challenging tasks for autonomous systems with real-time and energy constraints.**

In this paper, we present BLITZCRANK, an accelerator for motion planning algorithms using the abstraction of a factor graph. By formulating motion planning as a factor graph inference, we successfully reduce the scale of the problem and utilize the inherent matrix sparsity. BLITZCRANK is able to realize the user-defined optimal design by finding the optimal order of the factor graph inference. With a domain specific balancing order, BLITZCRANK achieves up to $7.4\times$ speed up and $29.7\times$ energy reduction compared to the software implementation on Intel CPU.

***Index Terms*—factor graph, autonomous machine computing, computer architecture, robotics, motion planning**

I. Introduction

Autonomous robots are increasingly replacing manual-control machines in the coming decades. The rise of autonomous robots such as autonomous vehicles, drones, home service robots and industry robots opens up an entirely new direction of computation. In the various kernels in autonomous robots, motion planning [1], which aims to find the best path to the destination without collision, serves as one of the most critical tasks. In most complex environments, motion planning is frequently activated to search for a new and safe path to avoid surrounding objects. The correctness and efficiency of motion planning contribute significantly to the mission success rate and whether the machine can meet hard real-time limitations.

As the importance of efficient motion planning has been realized for a long time, researchers have been working on building accurate and efficient motion planning systems. A typical CPU-based motion planning system [2] consumes several seconds. GPU accelerations have been proposed as well [3], and are able to reduce the runtime latency to one hundred or several hundred milliseconds. However, GPUs usually bring significant energy consumption overhead to the battery support autonomous systems.

While there have been many recent proposals on building dedicated accelerators for motion planning [4], [5], they all try to directly solve the problem of accelerating motion planning as an integrated problem which usually results in very high chip area or resource consumption when the problem size increases. For example, a scenario in the widely-adopted WAM arm dataset [6] can be translated into solving a system of linear equations with hundreds of variables. An intermediate representation is a form of abstraction for hardware designers to map the complicated applications into several fixed dedicated operators. Without a proper intermediate representation, trying to solve the system could easily lead to a deadlock situation or take much higher latency and energy, even though most of the items in the coefficient matrix we use to solve the system are zero.

In this paper, however, we try to accelerate the motion planning algorithm in an incremental way, which is to separate the huge problem into several steps, by utilizing the abstraction of factor graph [7] for the first time. A factor graph is a graph representing the factorization of a probability distribution function which is the key problem in solving motion planning algorithms. Different orders of traversing factor graph can result in different hardware designs. We provide a framework for users to search for the optimal order of factor graph inference. We also provide a domain specific order that could balance the hardware resource and runtime latency.

We transform the motion planning problem into an incremental optimization problem by using the abstraction of factor graph. By accelerating the motion planning algorithms using factor graph, we achieve up to $7.4\times$ speed up and $29.7\times$ energy reduction compared to a CPU baseline. Furthermore, compared with a dedicated accelerator design that does not utilize factor graph, we achieve $7.6\times$ hardware resources reduction. Our major contributions are:

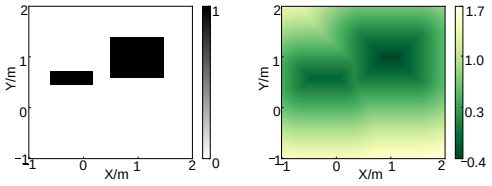
- We propose to leverage factor graph to reduce the size of the optimization processes that exist in motion planning algorithms and solve them in an incremental way. Leveraging factor graph also fully utilizes the potential sparsity in the matrix operations in motion planning.
- We show that the order of factor graph inference will influence the hardware resource utilization and performance. Thus we provide a framework for the users to search for their optimal inference orders. We also propose an order based on domain specific knowledge of motion planning factor graphs.
- We propose BLITZCRANK, a hardware accelerator that exploits the potential parallel computation patterns utilizing factor graph to accelerate motion planning algorithms.

The rest of this paper is organized as follows. Sec. II formulates the motion planning problem into the context of factor graph. Sec. III introduces the order of factor graph inference. Sec. IV delves into the hardware design of the proposed accelerator. Sec. V presents the evaluation results and we conclude in Sec. VI.

II. Factor Graph Formulation For Motion Planning

In this section, we explain how motion planning can be formulated using an abstraction of factor graph. We show motion planning formulation in Sec. II-A and how we can map motion planning into factor graph inference in Sec. II-B.

* indicates equal contribution to the paper.
† indicates the corresponding author of the paper.



(a) The occupied grid map matrix. (b) The SDF matrix.

Fig. 1. Matrix representation of a space example with two rectangular obstacles.

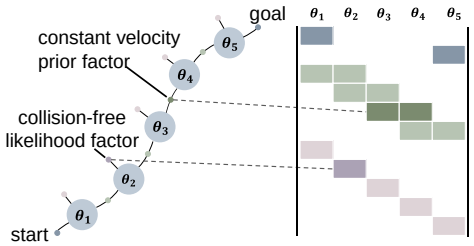


Fig. 2. A toy example of factor graph for motion planning including five states (the left) and the sparsity pattern of matrix $\mathbf{A}$ (the right). The dashed lines indicate the counterparts of the factors and block rows in $\mathbf{A}$.

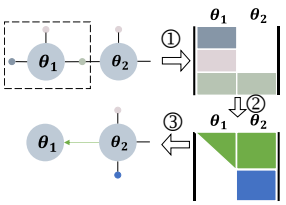


Fig. 3. Operations of factor graph inference. The node being eliminated is θ_1.

A. Motion Planning Formulation

Motion planning algorithms attempt to find trajectories to be both smooth and collision-free. In optimization-based motion planning algorithms, several error factors are introduced to construct a cost function. The motion trajectory satisfying to be both smooth and collision-free is obtained by minimizing the cost function. Due to the duality of optimization and probabilistic inference [8], we can also view motion planning from a probabilistic inference perspective, where the joint probability distribution corresponds to the cost function in the optimization problem, allowing us to map motion planning problems into the form of factor graph.

The trajectory is represented by N discrete states as $\Theta = [\theta_1, \dots, \theta_N]^T$, where each θ_i consists of the system configuration vector (e.g., joint angles of a robot arm) and its derivative with respect to time (e.g., joint angular velocities). In this formulation, our goal is to find the maximum a posterior (MAP) solution of these states, as shown in Equ. 1, given a prior distribution $p(\Theta)$ encouraging smoothness and a likelihood distribution $p(c = 0|\Theta)$ encouraging to be collision-free, where $c = 0$ indicates the trajectory is non-collision. When formulating motion planning into a factor graph, each variable node represents a state, and each factor node denotes the probability distribution of states connected to it.

$$\Theta^* = \arg\max_{\Theta} p(\Theta|c = 0) = \arg\max_{\Theta} p(c = 0|\Theta)p(\Theta) \quad (1)$$

Motion planning factor graph has two basic types of factors. The constant velocity prior factor is defined as:

$$p(\theta_i, \theta_{i+1}) \propto \exp\{-\frac{1}{2}\|\Phi(t_{i+1}, t_i)\theta_i - \theta_{i+1}\|^2_{Q_i}\}, \quad (2)$$

where $\Phi(t_{i+1}, t_i)$ is the state transition matrix between two adjacent states, Q_i is the covariance matrix describing the uncertainty of the distribution, and $\|\cdot\|^2_{Q_i}$ is the Mahalanobis norm that quantifies the error. This factor denotes that the probability is higher when the velocities of two adjacent states are closer, and represents smoothness.

Another factor is the collision-free factor. The likelihood probability is low when the motion agent is close to the obstacle surface or has collided with the obstacle, and vice versa. The traditional hinge loss is used to represent the collision cost:

$$c(x) = \begin{cases} -d(x) + \epsilon & d(x) \le \epsilon \\ 0 & d(x) > \epsilon \end{cases}, \quad (3)$$

where $d(x)$ is the signed distance from any point x in the workspace to the closest obstacle surface, and ϵ is a 'safety distance'. The signed distance $d(x)$ is calculated from a signed distance field (SDF) before factor graph inference. Fig. 1a shows an example of the occupied grid map matrix, where each element denotes the probability of being occupied by obstacles, and Fig. 1b shows its SDF.

To model robots of arbitrary shapes while simplifying computations, several spheres are used to represent the robot's body [9]. In this way, the problem of non-collision is converted from ensuring the distance between robot surface and any obstacles greater than ϵ to ensuring the distance between all sphere's centers and any obstacles greater than ϵ plus sphere radius. The obstacle cost function for each state θ_i is completed by computing the signed distances for each sphere s_j and then collecting their hinge loss into a single vector:

$$h(\theta_i) = [c(k(\theta_i, s_j))]|_{1 \le j \le M}, \quad (4)$$

where $k(\theta_i, s_j)$ is the forward kinematics that maps s_j in state θ_i to the workspace and gets the signed distance. Thus, the collision-free likelihood factor can be defined as:

$$p(c = 0|\theta_i) \propto \exp\{-\frac{1}{2}\|h(\theta_i)\|^2_{\Sigma_i}\}, \quad (5)$$

where Σ_i denotes the uncertainty of the distribution and $\|\cdot\|^2_{\Sigma_i}$ is the Mahalanobis norm, similarly.

B. MAP Formulation with Factor Graph

The above MAP can be formulated to minimize the negative log of Equ. 1 and then convert the factor graph inference to a nonlinear least squares optimization problem:

$$\begin{aligned} \Theta^* &= \arg\max_{\Theta}\{p(c = 0|\Theta)p(\Theta)\} \\ &= \arg\min_{\Theta}\{-\log(p(c = 0|\Theta)p(\Theta))\} \\ &= \arg\min_{\Theta}\{\sum_i \|\Phi(t_{i+1}, t_i)\theta_i - \theta_{i+1}\|^2_{Q_i} + \sum_i \|h(\theta_i)\|^2_{\Sigma_i}\}. \end{aligned} \quad (6)$$

Directly solving the nonlinear least square problem has tremendous high complexity and is not affordable. Typical nonlinear solvers, such as the Gaussian-Newton method [10], follow the iterative process. They start with an initial value Θ_0. At each iteration, an increment Δ is computed and applied to the next estimate $\Theta = \Theta \oplus \Delta$. The iteration stops when specific convergence criteria are reached, such as Δ falling below a small threshold. Δ can be obtained by solving a linear least squares problem at each step as:

$$\Delta^* = \arg\min_{\Delta} \|\mathbf{A}\Delta - \mathbf{b}\|^2, \quad (7)$$

where the matrix $\mathbf{A}$ collects all Jacobian matrices (the partial derivative of the error with respect to the state), and the vector $\mathbf{b}$ integrates all error factors. The covariance matrices are also multiplied into $\mathbf{A}$ and $\mathbf{b}$, where $\mathbf{A}$ is a large yet sparse matrix.

Solving the equation directly requires significant costs on computation and memory. Factor graph allows us to solve it in an incremental way. The structure of the factor graph directly corresponds to the sparsity pattern of $\mathbf{A}$ [7]. Performing factor graph inference is equivalent to solving the system of linear equations $\mathbf{A}\Delta = \mathbf{b}$ in an incremental way.

Fig. 2 shows a toy example of the motion planning factor graph and the sparsity pattern of matrix $\mathbf{A}$. Each variable node (circle) corresponds to a state to be optimized and each factor node (dot) corresponds to a block row (several rows) in $\mathbf{A}$. For each state, one or more factors can be observed/calculated. The smoothness factor will be observed by two states at the same time but the collision-free factor can be observed by only one state.

III. Factor Graph Inference

We introduce how to infer on a factor graph in this section. We explain the factor graph inference basics (Sec. III-A). We then propose an algorithm to identify the optimal factor graph inference order, which is key to our speedup (Sec. III-B).

A. Factor Graph Inference Basics

Factor graph inference starts from eliminating all variable nodes with a specific order [8], which is equivalent to QR decomposition on matrix $\mathbf{A}$.

We will use the toy example in Fig. 2 to illustrate this process. If the forward elimination order $(\theta_1, \theta_2, \ldots, \theta_5)$ is selected, Fig. 3 shows the operations of the factor graph and matrix when eliminating θ_1. First, the adjacent factors are constructed as a small matrix $\overline{\mathbf{A}}$, where the block rows are composed of neighboring factor nodes and the block columns correspond to the neighboring variable nodes. Second, partial QR decomposition is performed on $\overline{\mathbf{A}}$ to zero out the elements below the diagonal of the first block column. Third, the first block row of the updated matrix indicates that the solution of θ_1 is dependent on θ_2, as represented by the arrow in the graph, which points from θ_2 to θ_1, and a new factor represented by the second block row is inserted into the factor graph.

The above steps are then iterated for subsequent variables. After eliminating all variables, $\mathbf{A}$ is transformed into an upper triangular matrix, where we perform back substitution at the end.

B. Optimal Order of Factor Graph Inference

Why Does Order Matter? Factor graph inference is a graph traversal problem. For a factor graph with N variable nodes, given an arbitrary starting point, there are up to $N!$ number of inference orders, which represents when different nodes are visited. Every order could lead to a different hardware design with different latency and memory requirements. For example, if we start inference from θ_1, the constructed matrix $\overline{\mathbf{A}}$ will contain two block columns, as shown in Fig. 3. However, if we start inference from θ_2, $\overline{\mathbf{A}}$ will contain three block columns.

Exhausting all the designs to identify the optimal design takes years. Our idea is to identify the approximate optimal order purely in software without actually synthesizing the hardware. We resort to common software metrics closely related to specific hardware metrics and provide a software framework to compute them. Our framework works by considering only basic software measurements that can be obtained from a software implementation of the factor graph inference. The order we find is an approximation to the optimal order of factor graph inference. Although the order is not guaranteed to be optimal, we save significant time of synthesizing real hardware.

We use three software based metrics that could reflect hardware constraints as examples. The software metrics are easy to get when traversing the factor graph. The first software based metric we use is the maximum matrix size along factor graph inference. Each step of factor graph inference is a set of matrix operations. The maximum matrix size along all the steps equals the hardware resource requirements upper bond needed to finish the factor graph inference.

The second metric we use is the average matrix size during factor graph inference. This metric is closely related to runtime latency.

Algorithm 1 Computing the matrix size and density given an inference order.

Input: Factor graph $\Phi_{1:N}$ and an inference order η
Output: Maximum matrix size S_{max}, average matrix size S_{ave} and average matrix density D_{ave}

for θ_i in η **do**:
  Search for the neighboring factor nodes $f_{1:n}$.
  $R_i = \sum_{i=1}^{n} f_i^{row}$
  $C_i = \#(\bigcup_{i=1}^{n} f_i^{var}) \times 2 \cdot \mathcal{DOF}$
  $S_i = R_i \times C_i$
  $D_i = \sum_{i=1}^{n} f_i^{size} / S_i$
  $f_{new}^{row} = R_i - 2 \cdot \mathcal{DOF}$
  $f_{new}^{col} = C_i - 2 \cdot \mathcal{DOF}$
  $f_{new}^{var} = \bigcup_{i=1}^{n} f_i^{var} \setminus \theta_i$
  Remove $f_{1:n}$ and add f_{new} to $\Phi_{1:N}$.
end for
$S_{max} = max(S_{1:N})$
$S_{ave} = mean(S_{1:N})$
$D_{ave} = mean(D_{1:N})$

Smaller average matrix size usually results to lower runtime latency. With carefully designed pipelining, reducing the size of the matrix operation directly reduces the latency of each pipe.

The third software based metric we use is the average matrix density. Similar to average matrix size, average matrix density is also related to runtime latency and energy consumption. Higher average density represents better utilization of the existing sparsity.

Algo. 1 shows the algorithm for computing the above three metrics given a factor graph and an inference order. The three metrics computation follows the order of factor graph inference. For each variable node θ_i entering the inference order, all neighboring factor nodes $f_{1:n}$ are searched first. Then the total number of matrix rows R_i and columns C_i, total matrix size S_i, and total matrix density D_i are calculated from the information in f_i, including the number of each matrix row f_i^{row}, the neighboring variables f_i^{var}, and each matrix size f_i^{size}. Finally, the new factor information is updated, and the neighboring factors $f_{1:n}$ are removed from the factor graph alongside the new factor f_{new} is added. After traversing all variable nodes, we can get the maximum matrix size S_{max}, average matrix size S_{ave}, and average matrix density D_{ave}. Users are also able to apply their own software metrics using our framework.

Domain Specific Balancing Order. We also propose a domain specific order based on the observation of domain specific factor graph characteristics that tries to balance resource utilization and runtime efficiency. We have two observations of factor graphs in motion planning algorithms.

- First, starting from the side of the factor graph usually will have less maximum matrix size compared to starting from the middle of the factor graph. This is because the middle states can usually observe multiple nearby states with multiple factors.
- Second, the factor graph of the motion planning algorithm is symmetric.

Based on the two observations we make on the motion planning factor graph, we propose a domain specific balancing order of factor graph inference that tries to improve performance under specific hardware resource constraints. We propose to start from the side of the factor graph and utilize the symmetry of the factor graph to perform inference in a parallel way. On the one hand, we try

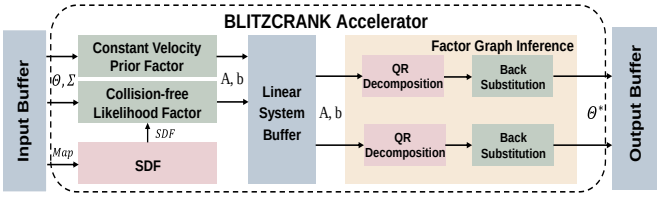


Fig. 4. BLITZCRANK accelerator architecture. Red blocks indicate blocks with high performance requirements and resource consumption, so we focus on optimizing them by parallelism; Green blocks represent multiplexed blocks and we pipeline them; Blue blocks denote storage buffer; Arrows direct the data flow.

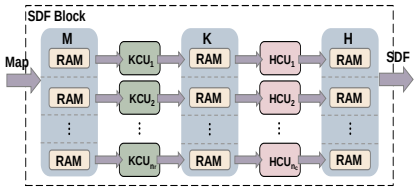


Fig. 5. The architecture of the SDF block. We design $\mathbf{n_r}$ number of $\mathbf{K}$ computing units (KCU) and $\mathbf{n_c}$ number of $\mathbf{H}$ computing units (HCU) to accelerate the SDF computing process.

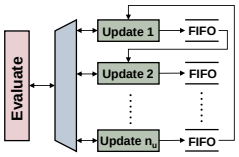


Fig. 6. The QR decomposition block uses one Evaluate unit and $\mathbf{n_u}$ time-multiplexed Update units to ensure a balanced pipeline.

not to exceed the hardware resource constraints by applying for an order with a small maximum matrix size. On the other hand, the parallelization ensures runtime efficiency. We show the comparison among different orders in Sec. V-B.

IV. BLITZCRANK Hardware

We propose to accelerate the motion planning algorithms given an order of factor graph inference. Sec. IV-A gives an overview of BLITZCRANK. Sec. IV-B to Sec. IV-D present the design details of sub-blocks.

A. Hardware Deign

Fig. 4 shows the BLITZCRANK hardware architecture of the accelerator, which consists of a collection of optimized hardware blocks. The top-level architecture contains four sub-blocks: the SDF block, the constant velocity prior factor block, the collision-free likelihood factor block, and the factor graph inference block.

The data flow of the accelerator is as follows. The map matrix is loaded from the input buffer to the SDF block to compute the SDF matrix. The state variables Θ and covariance matrix Σ are loaded from the input buffer to the two factor blocks. The two factor blocks compute the Jacobian matrix $\mathbf{A}$ and error vector $\mathbf{b}$, which will be used to solve the linear equations. The factor graph inference block performs matrix decomposition and back substitution for the linear equations to solve the result Δ, and adds it to the starting point for evaluation. The process is iterative. Whether the iteration continues or outputs the optimal value Θ^* will be decided by checking if the convergence requirements are satisfied.

B. SDF Block

The calculation of SDF matrix is summarized in four steps: First, transform the occupancy grid map matrix $\mathbf{G}$ into a binary map matrix $\mathbf{M}$ according to the given threshold. Second, invert the elements in $\mathbf{M}$, i.e., $\mathbf{0}$ becomes $\mathbf{1}$ and vice versa, to obtain the matrix $\mathbf{M}'$. Third, for each $\mathbf{1}$ in $\mathbf{M}$ and $\mathbf{M}'$, find the distance to the closest $\mathbf{0}$ to get the matrix $\mathbf{H}$ and $\mathbf{H}'$. Finally, $\mathbf{H}'$ minus $\mathbf{H}$ to get the SDF matrix $\mathbf{S}$.

The most time-consuming step above is the third step, which is to compute matrix $\mathbf{H}$. Given a 2D map, this step can be further divided into two sub-steps [11]. First, find the distance on each row to get the matrix $\mathbf{K}$. Then on each column of $\mathbf{K}$, add the vertical distance to the remaining rows and find the minimum value to get the nearest distance on the plane, forming the matrix $\mathbf{H}$. For a 3D map, $\mathbf{H}$ can be solved by adding a step to find the minimum distance on the third dimension. Since these steps perform the same operation for all rows and columns, respectively, it is possible to increase the number of computing units to improve parallelism and, thus, performance.

Fig. 5 shows the architecture of the SDF block. A detailed parallelization can be performed when calculating $\mathbf{K}$ and $\mathbf{H}$. To be specific, when computing $\mathbf{K}$, $\mathbf{n_r}$ number of rows can be unrolled at the same time. When calculating $\mathbf{H}$, $\mathbf{n_c}$ number of columns can be unrolled at the same time. Thus $\mathbf{n_r}$ and $\mathbf{n_c}$ number of computing units can be designed to accelerate the computing process.

C. Factor Graph Inference Block

The factor graph inference block contains QR decomposition blocks and back substitution blocks. Since the domain specific balancing order we apply to infer the factor graph, two sets of QR decomposition blocks and back substitution blocks are designed for parallel inference from two sides of the factor graph.

The QR decomposition starts from the first column of the input matrix $\overline{\mathbf{A}}$. Two phases are needed. In the Evaluate phase, the Householder matrix $\mathbf{P}$ is constructed from this column. In the Update phase, the entries below the diagonal of this column are set to zeros by left multiplying $\mathbf{P}$, and the following columns are updated. The updated matrix in the lower right corner serves as the input for the next iteration. The iteration continues until the first block column is eliminated.

Based on our analysis of the data dependencies, the Evaluate-Update phase can be pipelined. The current iteration of the Update phase and the next iteration of the Evaluate phase have no data dependency and thus can be paralleled. We show the architecture design in Fig. 6. As the Update phase is the bottleneck in the pipeline, we design $\mathbf{n_u}$ number of time-multiplexed Update units, each of them is connected through a FIFO due to the sequential data read/write relationship between the front and back units. All of the Update units are connected to the Evaluate unit. The performance improves and converges as we increase the $\mathbf{n_u}$, alongside the increase of the cost of hardware resources.

D. Other Blocks

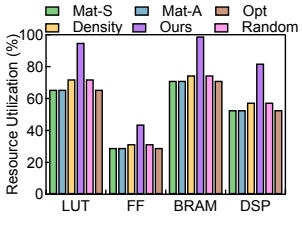
The constant velocity prior factor block and the collision-free likelihood factor block can compute the Jacobian matrix and error vector for one factor at a time. We pipeline the two blocks to accelerate the process further.

V. Evaluation

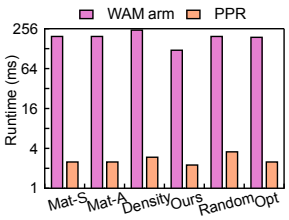
To evaluate the proposed accelerator, we conduct a series of experiments. Sec. V-A introduces the experimental setup. We compare variants with different inference orders in Sec. V-B. Sec. V-C evaluates the accuracy of our accelerator. We evaluate the matrix size and density in Sec. V-D and demonstrate the speedup and energy reduction of the proposed hardware accelerator compared to software implementation. Sec. V-E compares the resource consumption of the proposed hardware accelerator with a large accelerator that does not utilize factor graph as an abstraction.

A. Experimental Setup

Hardware Setup. We synthesize the accelerator using Vitis-HLS, and we run it on the Xilinx Zynq-7000 SoC ZC706 FPGA. The accelerator operates at a fixed frequency of 167 MHz. The FPGA power consumption is estimated by the Vivado power analysis tool

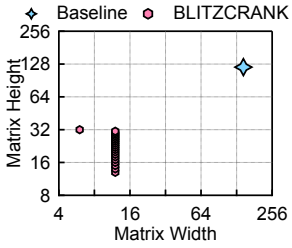


(a) Resource utilization comparison between variants with different inference orders.

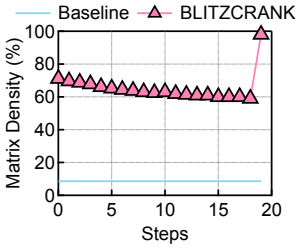


(b) Runtime latency comparison between variants with different inference orders.

Fig. 7. Resource utilization and latency comparison between variants with different inference orders.



(a) The matrix size is reduced by using factor graph.



(b) The matrix density is increased by using factor graph.

Fig. 8. Matrix size reduction and density increase by using factor graph in BLITZCRANK.

using real workloads under test. All power and resource utilization data are obtained after the design passes the post-layout timing analysis.

Software Setup. A software implementation of motion planning, GPMP2 [12], is used as a baseline, which uses GTSAM [13] to implement factor graph inference. The software is evaluated on the 11th Intel processor that has 16 cores and operates at 2.5 GHz. The Intel CPU power is measured through a power meter.

Datasets. We evaluate the accelerator with two different datasets. The first one is the 7-DOF WAM arm dataset [6] consisting of 24 unique planning problems. The second one is the 3-DOF Planar Point Robot (PPR) dateset [12] consisting of 30 unique planning problems.

Inference Order Variants. We test with six different orders. **Mat-S** represents the order results in the smallest maximum matrix size. **Mat-A** represents the order with the smallest average matrix size. **Density** represents the order results in the highest average matrix density. **Ours** represents the balancing order we propose. We also test a random inference order **Random** starting from a random point. We also infer with an order calculated by [14], which tries to calculate an order that can result in minimum fill-in, and we refer to **Opt**.

B. Comparison among Variants with Different Inference Orders

As we have discussed in Sec. III-B, we exhaustively search all the possible orders to find different optimal orders to meet different hardware constraints or goals.

We compare variants with different inference orders in Fig. 7. Fig. 7a shows the resource utilization of different variants. Among all the variants, **Mat-S** results in the least consumption in all categories of hardware resources. This indicates that using the smallest maximum matrix size is an effective metric to meet the hardware resource constraints. **Mat-A** results in the exact same hardware design compared to **Mat-S**. We find that in our dataset, the average matrix size is positively correlated to the maximum matrix size. **Density** is shown to be a less effective metric. It results in high hardware resource consumption. **Random** order has the worst hardware resource consumption of all the searched orders. **Opt** performs surprisingly well on hardware resource consumption. As a pure software metric, it tries to minimize the fill-in degree. **Ours** has the highest hardware resource consumption as we try to process the factor graph inference in a parallel way from two sides of the graph, which results in one more factor graph inference unit in hardware.

Fig. 7b shows the runtime latency comparison among all variances. We show the runtime latency on two datasets on the y-axis (in log scale). In all the searched orders, **Opt** has the shortest runtime latency in both datasets, followed by **Mat-S** and **Mat-A**. **Density** has the worst latency in WAM arm dataset and **Random** has the worst latency in PPR dataset. **Ours** outperforms all the other orders in both datasets due to the parallelism. **Ours** reduces the latency by 36.1% compared to **Opt**. In all the following evaluation sections, we will use **Ours** as the inference order.

TABLE I
THE SUCCESS RATE ON INTEL CPU AND BLITZCRANK EVALUATING ON THE WAM ARM DATASET AND THE PPR DATASET.

	WAM arm	**PPR**
Intel CPU	95.8%	93.3%
BLITZCRANK	95.8%	93.3%

C. Accuracy

The data types of both BLITZCRANK and baseline are single-precision floating point numbers. In our experiments, all initial states are initialized by a constant-velocity straight line trajectory in configuration space. The initial state setting is common among different motion planning algorithm frameworks [9], [12], [15]. Both the software baseline and our hardware strictly follow the same initial states. We use success rate as the metric of accuracy.

We summarize the accuracy results in Tbl. I. We show that the accuracy of BLITZCRANK is exactly the same as the software version, indicating our method does not lose accuracy by solving the problem in an incremental way.

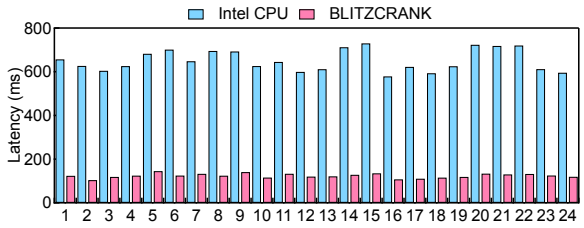
D. Performance Evaluation

Comparison on Matrix Size and Density. Our major contribution is to utilize factor graph as an abstraction to solve the motion planning algorithm in an incremental way. By doing so, we significantly reduce the size of the matrix operation and fully utilize the sparsity inside the matrix. We show the results in Fig. 8. We demonstrate it with one example in PPR dataset.

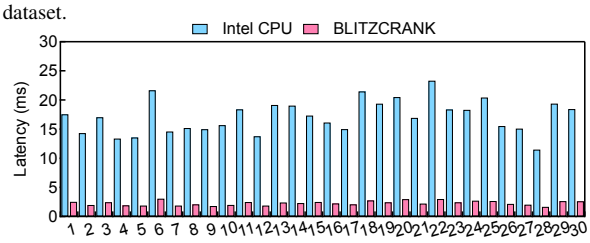
Fig. 8a shows the size reduction in matrix operations with x-axis representing the matrix width and y-axis representing the matrix height. Without using factor graph as an abstraction, the baseline needs to solve a linear equation with matrix size to be 146×120. BLITZCRANK successfully reduces the scale of the problem by orders of magnitude. BLITZCRANK in total takes 20 steps to finish the process, where the largest matrix size in one step is 31×12.

At the same time, the resource utilization is significantly improved by skipping the sparsity in the matrix. Fig. 8b shows the original problem has a matrix density (non-zero elements) of 8.6%. BLITZCRANK has an average matrix density of 65.1% with the largest density to be 98.4%.

Comparison on Runtime and Energy Efficiency. We evaluate the runtime latency and energy efficiency of BLITZCRANK hardware compared with the Intel CPU baseline. We show the runtime improvements on two datasets in Fig. 9. Fig. 9a shows that BLITZCRANK significantly reduces the planning latency compared to the Intel CPU baseline on WAM arm dataset. BLITZCRANK has an average speed up of $5.2\times$ on the runtime latency with a maximum



(a) Performance improvement compared to Intel CPU on WAM arm dataset.



(b) Performance improvement compared to Intel CPU on PPR dataset.
Fig. 9. Performance improvement brought by BLITZCRANK.

TABLE II
RESOURCE UTILIZATION (UTILIZATION PERCENTAGES AND ABSOLUTE NUMBERS) OF BLITZCRANK AND THE BASELINE ACCELERATOR.

	LUT	FF	BRAM	DSP
BLITZCRANK	95% (208029)	43% (191482)	99% (540)	82% (738)
Baseline	827% (1809722)	496% (2170775)	572% (3120)	368% (3314)

speed up of $5.6\times$. The average latency of motion planning has been reduced to 124.4 ms. Fig. 9b shows similar trends on PPR dataset. The average speed up on PPR dataset is $7.4\times$ with a maximum speed up of $7.7\times$.

BLITZCRANK also shows significant improvements in energy efficiency. We show the energy consumption comparison in Fig. 10. Compared to Intel CPU, BLITZCRANK hardware reduces the energy consumption by $29.7\times$ and $21.9\times$ on two datasets.

E. Comparison with Large Accelerator

One of the most important benefits of using factor graph as an abstraction is to reduce the problem size. We also compare BLITZCRANK with an accelerator that does not apply factor graph as an abstraction and directly solves the optimization process.

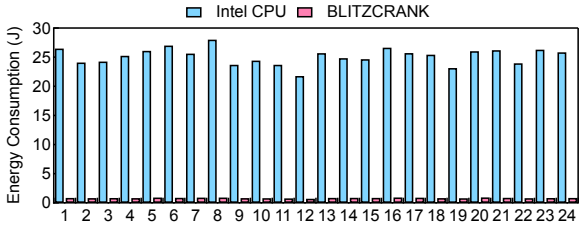
We also try our best to implement the baseline accelerator. Except for the hardware units we use for factor graph inference, the rest of the hardware design is kept for the baseline accelerator. We show the resource utilization of BLITZCRANK and the baseline accelerator in Tbl. II. The baseline accelerator consumes $8.7\times$ higher LUT, $11.3\times$ higher FF, stores $5.8\times$ more data and uses $4.5\times$ more DSPs. Even the largest FPGA in Xilinx Zynq family can not support such a large accelerator.

VI. CONCLUSION

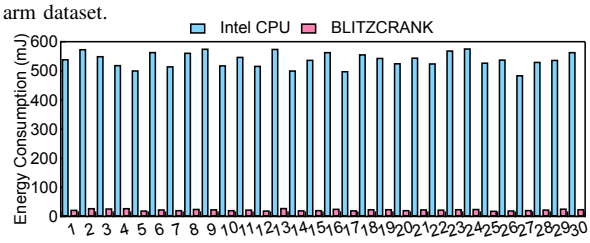
This paper takes the first step of referring to factor graph as an abstraction to build an accelerator for motion planning algorithms. The key contribution of BLITZCRANK is to formulate motion planning as factor graph inference and try to find the user-defined optimal order of it. We demonstrate factor graph is an ideal abstraction for hardware designers to reduce the scale of the hardware accelerators and utilize the potential sparsity inside autonomous machine applications.

ACKNOWLEDGEMENT

The authors would like to thank the support of the National Natural Science Foundation of China under Grant U21B2031.



(a) Energy efficiency improvement compared to Intel CPU on WAM arm dataset.



(b) Energy efficiency improvement compared to Intel CPU on PPR dataset.
Fig. 10. Energy efficiency improvement brought by BLITZCRANK.

REFERENCES

[1] Jean-Claude Latombe. *Robot Motion Planning*, volume 124. Springer Science & Business Media, 2012.
[2] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.
[3] Jia Pan, Christian Lauterbach, and Dinesh Manocha. g-planner: Real-time motion planning and global navigation using gpus. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.
[4] Sean Murray, William Floyd-Jones, Ying Qi, George Konidaris, and Daniel J Sorin. The microarchitecture of a real-time robot motion planning accelerator. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12. IEEE, 2016.
[5] Shiqi Lian, Yinhe Han, Xiaoming Chen, Ying Wang, and Hang Xiao. Dadu-p: A scalable accelerator for robot motion planning in a dynamic environment. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.
[6] Mustafa Mukadam, Xinyan Yan, and Byron Boots. Gaussian Process Motion planning. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9–15. IEEE, 2016.
[7] Frank Dellaert. Factor graphs: Exploiting structure in robotics. *Annual Review of Control, Robotics, and Autonomous Systems*, 4:141–166, 2021.
[8] Frank Dellaert and Michael Kaess. Factor graphs for robot perception. *Foundations and Trends® in Robotics*, 6(1-2):1–139, 2017.
[9] Nathan Ratliff, Matt Zucker, J. Andrew Bagnell, and Siddhartha Srinivasa. CHOMP: Gradient optimization techniques for efficient motion planning. In *2009 IEEE International Conference on Robotics and Automation*, pages 489–494. IEEE, 2009.
[10] Yong Wang. Gauss–Newton method. *WIREs Computational Statistics*, 4:415–420, 2012.
[11] Ricardo Fabbri, Luciano Da F. Costa, Julio C. Torelli, and Odemir M. Bruno. 2D Euclidean distance transform algorithms: A comparative survey. *ACM Computing Surveys*, 40:1–44, 2008.
[12] Jing Dong, Mustafa Mukadam, Frank Dellaert, and Byron Boots. Motion Planning as Probabilistic Inference using Gaussian Processes and Factor Graphs. In *Robotics: Science and Systems XII*. Robotics: Science and Systems Foundation, 2016.
[13] Georgia Institute of Technology. GTSAM. https://github.com/borglab/gtsam. Accessed: 2022-11-10.
[14] Timothy A. Davis, John R. Gilbert, Stefan I. Larimore, and Esmond G. Ng. A column approximate minimum degree ordering algorithm. *ACM Transactions on Mathematical Software*, 30:353–376, 2004.
[15] John Schulman, Yan Duan, Jonathan Ho, Alex Lee, Ibrahim Awwal, Henry Bradlow, Jia Pan, Sachin Patil, Ken Goldberg, and Pieter Abbeel. Motion planning with sequential convex optimization and convex collision checking. *The International Journal of Robotics Research*, 33:1251–1270, 2014.