Fully integrated quantum method for classical register allocation in LLVM

Brice Chichereau, Stéphane Vialle, Patrick Carribault

Fully Integrated Quantum Method for Classical Register Allocation in LLVM

Brice Chichereau *CEA, DAM, DIF*
F-91297 Arpajon, France
Université Paris-Saclay, CEA,
Laboratoire en Informatique
Haute Performance pour
le Calcul et la simulation
Bruyères-le-Châtel, France
brice.chichereau@cea.fr

Stéphane Vialle *CentraleSupélec*
Gif-sur-Yvette, 91192, France
Université Paris-Saclay, CNRS,
Laboratoire Interdisciplinaire
des Sciences du Numérique
Orsay, France
Stephane.Vialle@centralesupelec.fr

Patrick Carribault *CEA, DAM, DIF*
F-91297 Arpajon, France
Université Paris-Saclay, CEA,
Laboratoire en Informatique
Haute Performance pour
le Calcul et la simulation
Bruyères-le-Châtel, France
patrick.carribault@cea.fr

Abstract

Quantum computing devices are being installed alongside supercomputing clusters to serve as hardware accelerators. This new type of architecture will require an integrated hybrid software stack. With this goal in mind, we have developed a fully integrated hybrid quantum-classical method in the hope of improving Register Allocation in the classical LLVM compiler. We propose a hybrid variational optimization algorithm for the PBQP formulation of Register Allocation. We implemented this algorithm in C++ inside LLVM using the NVIDIA CUDA-Q framework. The performance of the method is evaluated using NVIDIA CUDA-Q noiseless emulators and shows promising results while still needing further optimizations. Our work constitutes a demonstration of an end-to-end tight integration of a quantum subroutine inside an existing classical codebase of interest with potentially interesting performance in fault-tolerant hardware.

Keywords— compiler optimization, quantum computing applications, register allocation, quantum alternating operator ansatz

1 Motivation and goals

Quantum computing is establishing itself as a new paradigm that opens up many new avenues for performing computational tasks. The problems that could benefit from the capabilities of quantum devices range from integer factorization [1] to solving linear systems of equations [2] and combinatorial optimization [3]. Quantum devices used for computation –sometimes called Quantum Computers or Quantum Processing Units (QPU)– leverage the properties of quantum physical systems to perform computations [4].

As QPUs start to be installed in HPC supercomputing centers, the question arises of how to best integrate the existing classical systems with the new quantum computing resources. This new architectural paradigm raises many questions concerning its software stack and toolchain. Quantum computing devices work completely differently from classical computers and thus need specific programming models, compilers, runtimes, etc... Some efforts have been started to develop integrated hybrid quantum-classical software stacks and toolchains [5] but more effort will still be needed.

A central tool in many toolchains is the compiler responsible for converting human-readable code into machine instructions that can be run on a computer. To improve the output executable, a compiler contains many sub-tasks some of which are related to solving computationally hard optimization problems [6]. As combinatorial optimization is a good candidate problem for quantum computers, this motivates exploring the idea of integrating quantum computations in compilers.

Register Allocation is a compiler sub-task dedicated to deciding where to store program variables on the computer. This problem is related to hard optimization problems such as graph coloring [7] and could thus be a good candidate for quantum algorithms. Building upon our previous work on a quantum algorithm for the graph coloring formulation of Register Allocation [8], we propose a hybrid quantum-classical method to solve the more complex Partitioned Boolean Quadratic Problem (PBQP) formulation of Register Allocation. This method will be fully integrated into the LLVM compiler [9] in C++

using the NVIDIA CUDA-Q framework [10].

The main contributions of this work can be summed up as follows:

- Hybrid quantum-classical Register Allocation algorithm with a problem formulation chosen for its expressivity and adaptability to quantum computing.
- Quantum optimization algorithm based on the Quantum Alternating Operator Ansatz with some hard constraints of the problem taken into account in the quantum circuit.
- Integrated implementation of the algorithm in the LLVM compiler using the NVIDIA CUDA-Q framework.
- Experiments and evaluation of the hybrid algorithm using the NVIDIA CUDA-Q emulators for fault-tolerant hardware.

The rest of this paper is organized as follows: Section 2 goes over the background and related previous works. Section 3 describes in detail the modified quantum-classical register allocation algorithm and its integration into LLVM. Section 4 presents simulated experimental results for the performance of our method with and without noise. Finally, Section 5 concludes this paper.

2 Background & related work

2.1 PBQP Register Allocation

Compilation can be seen as converting a program in human-readable code into an executable file written in machine-readable instructions. Modern compilers such as GCC [11] or LLVM [9] also include many optimization steps to improve the quality, speed, memory footprint, etc... of compiled programs. Register Allocation is a crucial step in modern optimizing compilers: during this step variables are assigned to a storage location on the target device, either:

- A CPU register: very fast access but limited in quantity (usually around 32 in a modern CPU).
- Central memory (RAM): slower but abundant. A variable stored in memory is said to be "spilled".

This problem is constrained, two variables that exist simultaneously cannot be stored in the same register, this is called interference. Trying to minimize the time spent trying to access data while taking these constraints into account has led Register Allocation to be formulated as various optimization problems such as coloring a graph representing the interferences with colors corresponding to CPU registers [7].

Another formulation of Register Allocation is as a Partitioned Boolean Quadratic Problem (PBQP) [12]; this formulation is used in one of the available register allocators in the LLVM compiler. In this formulation, the allocations are represented by binary vectors using a one-hot encoding where the variable has a 1 at the index representing its chosen storage location. Given this encoding of the solution and V the set of all variables, the form of the PBQP optimization problem is:

$$\min_{\boldsymbol{x}} \left(\sum_{v<w} \boldsymbol{x}_v \cdot C_{v,w} \cdot \boldsymbol{x}_w^T + \sum_v \boldsymbol{c}_v \cdot \boldsymbol{x}_v^T \right) \tag{1a}$$

$$\text{s.t. } \forall v \in V,\ \boldsymbol{x}_v \cdot \mathbf{1}^T = 1 \tag{1b}$$

Where $\boldsymbol{x}$ is the set of solution vectors $\boldsymbol{x}_v$ for all variables v and the vectors $\boldsymbol{c}_v$ contain the cost of allocating variable v to each storage location (spill and registers). The matrices $C_{v,w}$ encode, for each pair of variables v, w, the costs of allocating the pair of variables to each specific pair of storage locations.

A PBQP instance can be represented as a graph G_{PBQP} where each node corresponds to a variable $\boldsymbol{x}_v$. Edges in the graph represent non-zero cost matrices $C_{v,w}$ between nodes v and w. An example of such a graph is provided in Figure 1 for the code in Listing 1.

Listing 1: Example pseudo-code.

```
a = 1
b = 2
print(a + b)
c = b
return b + c
```

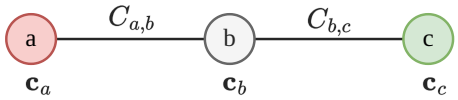


Figure 1: Example of a PBQP Graph. The cost vectors $\boldsymbol{c}_v$ are node weights while the cost matrices $C_{v,w}$ are edge weights.

Below are listed some basic costs and constraints, these are similar to what can be found in the LLVM implementation of PBQP Register Allocation.

Spill and register costs The cost of allocating variable v to a specific location. This cost can be infinite if, for example, the variable v is incompatible with a location due to hardware constraints.

Interference costs Infinite cost added to diagonal elements of $C_{v,w}$ if variables v and w are alive simultaneously to ensure that those are not assigned to the same register.

Coalescing costs An arbitrary negative cost penalty $-b$ can be added to matrix $C_{v,w}$ or vector $\boldsymbol{c}_v$ when we want to encourage storing v and w to the same location or v to a specific location. This is useful when variables are copies of one another or copies of existing registers.

An example of what all those costs combined could look like for the program in Listing 1 is given in Figure 2 for a very simple CPU architecture. In this architecture, "spill" is at index 0 of the cost vectors (similar to how it is in LLVM) and we have 2 hardware registers available.

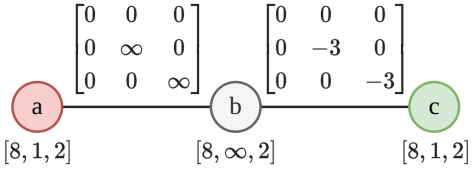


Figure 2: PBQP Graph for Listing 1 with cost vectors and matrices. "Spill" is at index 0 and we have 2 CPU registers available here.

A classical approximate method for solving the PBQP is given in [12] and further expanded in [13]. The problem is solved by removing nodes one by one from the "PBQP graph" G_{PBQP} and adding them to a stack until the graph is empty. The reduction starts with the "easier" nodes of degree ≤ 2 as removing them does not affect the optimality of the solution; "harder" nodes are removed afterward. The nodes are finally popped from the stack and assigned to the available storage location with the lowest cost. LLVM uses this type of solver in its PBQP Register Allocator.

2.2 Existing quantum algorithms for optimization

Multiple quantum algorithms that aim at solving optimization problems like the one in Equation 1 have been developed over the years. Various approaches have been proposed, such as Quantum Annealing [14] or Grover Search-based methods [**?**, 15]. Recently, methods centered around Variational Quantum Algorithms (VQA) have gained significant traction for their relative simplicity and potential advantages [16]. This class of algorithms revolves around using a classical optimization loop to modify a parameterized quantum circuit to produce states encoding interesting solutions.

One of the foundational algorithms for variational quantum optimization amongst the variants of VQA is the Quantum Approximate Optimization Algorithm (QAOA) [3]. Its parameterized circuits consist of alternating layers of gates representing $e^{-i\beta H_C}$ and $e^{-i\gamma H_M}$ where H_C is the "cost Hamiltonian" encoding the problem's costs function and $H_M = \sum_q X_q$ is the "mixer Hamiltonian" driving the evolution of the quantum system. γ and β are parameters to be optimized by the classical loop. This algorithm is simple and foundational, but it has shortcomings that have motivated the development of a large number of variants [17].

These variants include for example the Multi-Angle QAOA [18] where instead of 1 parameter per "layer" in the QAOA circuit, multiple parameters are used, for example: 1 per rotation gate. Another example is the ADAPT-QAOA [19] where the mixer Hamiltonian is selected at each step from a pool of operators by the classical optimizer. An interesting class of QAOA variants is the so-called Quantum Alternating Operator Ansatz (QAOAnsatz) [20]. This aims at generalizing the principles of QAOA to enable it to more efficiently solve problems such as ones with hard constraints. This is achieved by choosing a mixer Hamiltonian H_M that ensures that the evolution of the quantum state is limited to a feasible subspace that respects the problem constraints. We chose it here to solve PBQP Register Allocation due to its ability to natively handle hard problem constraints.

2.3 Related work

The Quantum Alternating Operator Ansatz [20] has previously been applied to various other computational problems using a variety of mixer Hamiltonian and initial state depending on the problem at hand. A first example is an application to Maximum k-Vertex Cover which experiments with classical and Dicke initial states and multiple types of XY gate-based mixers. Other examples include an algorithm for the Minimum Exact Cover problem [21] and also Satisfiability problems such as Max 2-SAT or Max 3-SAT. The latter uses a mixer Hamiltonian based on the principles of the Grover search algorithm [**?**, 22]. QAOAnsatz can be applied to many types of combinatorial optimization problems thanks to its great genericity. In all those examples, usage of the extended QAOAnsatz algorithm over the basic QAOA provides some advantages in terms of convergence speed or solution quality.

Using quantum computing in a classical optimizing compiler is a relatively recent idea. We previously developed a simple experimental quantum-based method for Register Allocation [8]. This method revolved around solving Register Allocation as an interference graph coloring problem [7] in the GCC compiler [11]. This was achieved by successively extracting Maximum Independent Sets[1] from the interference graph for each CPU register using QAOA.

3 Integrated quantum-classical register allocation

The integrated hybrid method is presented in a top-down manner from the high-level algorithm to the QAOAnsatz-based quantum algorithm.

3.1 Full integration into the LLVM compiler

As a reminder, the basic principle of the PBQP solver implemented in LLVM [12] is to "reduce" the PBQP graph G_{PBQP} by removing nodes one-by-one, starting with the ones that do not affect the optimality of the solution. This is implemented in LLVM [9] in the back-end C++ class `RegAllocPBQP` which defines the Register Allocation pass if PBQP is chosen as the allocation method by the user.

Current and future quantum hardware will have constraints that need to be taken into account by algorithms, such as the total number of available qubits or the noise levels. This can be synthesized as a limit on the size of problems that can be sent as inputs to the quantum algorithm. We take this into account by modifying the classical PBQP solver: G_{PBQP} is reduced using the classical algorithm on the CPU until it is small enough to fit into an available QPU. This allows performing more of the allocation on the classical or the quantum side depending on factors such as noise and problem instance characteristics.

Integration of quantum computing into existing C/C++ programs such as LLVM is not a simple problem as Python frameworks dominate the field of quantum computing. Some progress has been achieved recently through the development of quantum computing frameworks with a C++ API such as CUDA-Q [10] or Intel Quantum SDK [23] which enable direct integration into existing C++ applications without external Python routines. We chose to implement the quantum part of the modified solver using the CUDA-Q [10] framework for its simplicity and accessibility.

3.2 Hybrid quantum-classical PBQP solver

We note that a PBQP instance cannot directly be used as input for a quantum algorithm as its variables are binary vectors

[1] A Maximum Independent Set (MIS) of a graph $G = (V, E)$ is a subset S of V such that $(S \times S) \cap E = \emptyset$.

where we need one binary variable per qubit. The conversion is obtained by "flattening" the PBQP instance graph G_{PBQP} into a graph G_{flat} where each node will represent one of those binary variables. To represent the one-hot encoding constraint formulated in Equation 1b, each node (a, b, c) from G_{PBQP} will correspond to a clique[2] in G_{flat} with infinite cost constraints on the edges, shown as solid lines in Figure 3.

The size of the problem sent to the quantum algorithm is reduced by excluding any binary variable that corresponds to a storage location with infinite cost. These correspond to hardware and problem constraints like register size and type incompatibilities. The cliques are then connected based on the cost matrices for all edges in G_{PBQP}. The inter-variable costs are shown using dashed lines in Figure 3, we can see that the nodes corresponding to spills (cost 8) have no inter-variable constraints as spilled variables do not interfere.

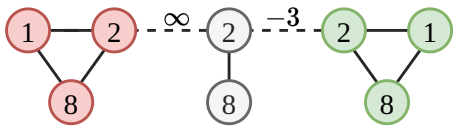


Figure 3: Flattened graph G_{flat} where each node represents a single binary variable of the graph from Figure 2 which corresponds to the code in Listing 1.

The construction of this flattened graph that will serve as input for the quantum algorithm allows us to compute the number of qubits that will be required given a 1-to-1 mapping from binary variable to qubit:

$$N_{\text{qubits}} = \sum_{v \in V} \#\{\text{storage location } a \mid \boldsymbol{c}_v(a) < \infty\} \tag{2}$$

For each variable v in the PBQP Register Allocation, we need 1 qubit per register/spill with finite cost, for example, the PBQP instance shown in Figure 3 would require 8 qubits. This leads to a total number of required qubits roughly proportional to *number of variables* × *number of registers*. With the problem instance having been converted to a quantum-compatible format, the whole hybrid quantum-classical method is detailed in Algorithm 1.

Algorithm 1 Hybrid Quantum PBQP Solver

Require: PBQP graph G_{PBQP}.
1: Connected Components Decomposition.
2: Alloc ← {}
3: **for all** connected subgraphs G'_{PBQP} **do**
4: **while** $N_{\text{qubits}}(G'_{\text{PBQP}}) >$ Threshold **do**
5: Alloc ← Alloc ∪ *Reduce*(G'_{PBQP})
6: **end while**
7: $G_{\text{flat}} \leftarrow$ *Flatten*(G'_{PBQP})
8: Alloc ← Alloc ∪ *QAOAnsatz*(G_{flat})
9: **end for**
10: **return** Alloc

The PBQP graph is first decomposed into its connected components to reduce the size of the problems given as input to the quantum algorithm. This is possible as 2 disconnected subgraphs of G_{PBQP} have no constraints between them and they can thus be considered as separate PBQP instances. The connected components decomposition of G_{PBQP} is obtained by performing a Depth-First Search in the graph.

[2] A clique of a graph G is a fully connected subgraph of G.

3.3 QAOAnsatz for PBQP register allocation

We chose to use a quantum algorithm based on the Quantum Alternating Operator Ansatz (QAOAnsatz) [20] methodology to solve our PBQP instances for its ability to encode hard constraints from the problem directly into the quantum circuits by restricting the quantum states to a feasible subspace. A QAOAnsatz formulation of a problem is given by a cost Hamiltonian H_C, a mixer Hamiltonian H_M and an initial state $|\Psi_0\rangle$. For PBQP Register Allocation we define those as follows:

Cost Hamiltonian

$$\begin{aligned} H_C = &\sum_{v \in V} \sum_{a} \boldsymbol{c}_v(a) Z_a^{(v)} \\ &+ \sum_{v<w} \sum_{a_1, a_2} C_{v,w}(a_1, a_2) Z_{a_1}^{(v)} Z_{a_2}^{(w)} \end{aligned} \tag{3}$$

Where $\boldsymbol{c}_v$ and $C_{v,w}$ are the cost vectors and cost matrices from the PBQP instance. $Z_a^{(v)}$ is the Pauli Z matrix acting on the qubit corresponding to storage location a for variable v. This cost Hamiltonian incorporates the terms from the cost function defined in Equation 1a.

Mixer Hamiltonian The mixer Hamiltonian we have chosen is built to enforce the one-hot encoding for each PBQP variable, i.e. enforce Equation 1b, it can thus be seen as a sum of Hamiltonian $H_M^{(v)}$ for each variable v.

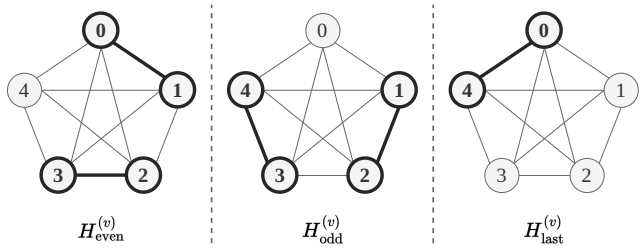


Figure 4: Visual representation of the ring XY mixer for the clique corresponding to a variable v. The bold edges correspond to applications of a XY gate.

$H_M^{(v)}$ needs to maintain the one-hot encoding at all times which corresponds to keeping the Hamming weight for the qubits "clique" of this variable constant at 1. This can be achieved using what's called a "XY ring" mixer Hamiltonian [20]. The basic idea of this type of mixer Hamiltonian is to apply 2-qubit XY interactions between each pair of qubits in the set in which we want the hamming weight to be constant in a ring pattern. The XY interaction is given by the Hamiltonian $X_{q_1}X_{q_2} + Y_{q_1}Y_{q_2}$ for qubits q_1 and q_2. To maximize the number of parallel 2-qubit operations, the XY interactions are first applied to pairs of qubits starting with an even index, then the ones with an odd index, and finally an interaction

is applied between the first and last qubits if there is an odd number of qubits. A visual representation of the action of this Hamiltonian is shown in Figure 4.

The XY gate that will be used in the final QAOAnsatz circuit will be of the form $e^{-i\beta\left(X_{a_i}^{(v)}X_{a_j}^{(v)}+Y_{a_i}^{(v)}Y_{a_j}^{(v)}\right)}$, this can be implemented in CUDA-Q using the `exp_pauli` function which is used to define exponential of Pauli gates. The code for the XY gate is shown in Listing 2 where `__qpu__` indicates that the `XY` function defines a quantum circuit and `cudaq::qubit` is the type for qubits.

Listing 2: XY-Gate definition using CUDA-Q.

```
__qpu__ static void XY(double Beta, cudaq::
    qubit &Q0, cudaq::qubit &Q1) {
  cudaq::exp_pauli(-Beta, "XX", Q0, Q1);
  cudaq::exp_pauli(-Beta, "YY", Q0, Q1);
}
```

Initial State We need to choose an initial state that is compatible with our mixer Hamiltonian, ie. one that respects the problem constraints. Here we set each group of qubits corresponding to a PBQP variable to the state "$|100\ldots\rangle$" which corresponds to spilling the variable to memory in LLVM. This state can easily be constructed starting from a zero state by applying X gates to all qubits that need to be set to 1.

The code for the full QAOAnsatz circuit in CUDA-Q is given in Listing 3. The quantum circuit here is declared using a `__qpu__` decorator on the struct's `()` operator. The `ZZ` gate is constructed using cascades of CNOT around a R_Z gate. In the framework, `cudaq::spin_op` designates a Hamiltonian and `cudaq::qvector<2>` a list of qubits (2-level qudits).

Listing 3: QAOAnsatz circuit using CUDA-Q.

```
struct QAOAnsatzCircuit {
  unsigned NbQubits, NbLayers;
  cudaq::spin_op HCost, HMixer;
  // Quantum Circuit definition
  __qpu__ void operator()(std::vector<double>
    Beta, std::vector<double> Gamma) {
    // Initialize Qubits Register
    cudaq::qvector<2> Q(NbQubits);
    InitialState(Q);
    // Main circuit
    for (int p = 0; p < NbLayers; p++) {
      // Cost Hamiltonian
      for (auto &Term: HCost)
        ZZ(Gamma[p], Term, Q);
      // Mixer Hamiltonian
      for (auto &Term: HMixer)
        XY(Beta[p], Term, Q);
    }
  }
}
```

4 Experimental results

4.1 Benchmarking methodology

We evaluate the performance of our integrated QAOAnsatz-based allocator by using it to compile a couple of classical mini-applications: LULESH [**?**] and miniFE [24] which are representative of typical High-Performance Computing (HPC) workflows. Our goal is to measure the impact of our algorithm on the quality of the compiled codes.

The experiments are performed by simulating the quantum circuits using the emulators provided in CUDA-Q [10]. The simulations are performed on NVIDIA A100 GPUs and the target architecture for our modified compiler is `x86_64` on AMD EPYC CPUs.

Modern CPU architectures usually have dozens of hardware registers available, meaning that the algorithm would need hundreds of qubits to fully compile typical programs according to Equation 2. To focus our study on the capabilities of our new quantum method we limit the number of available registers for allocation to a fixed value N_{regs}. We apply this restriction on the number of available registers for all the subgraphs allocated using the QPU.

4.2 Performance of the QAOAnsatz-based algorithm

We first compare the performance of our QAOAnsatz-based Register Allocation method to a basic QAOA where the hard constraints are converted into cost penalties added to the cost Hamiltonian. The basic QAOA will use the simple $H_M = \sum_q X_q$ mixer Hamiltonian and a fully superposed $|+\rangle^{\otimes n}$ initial state.

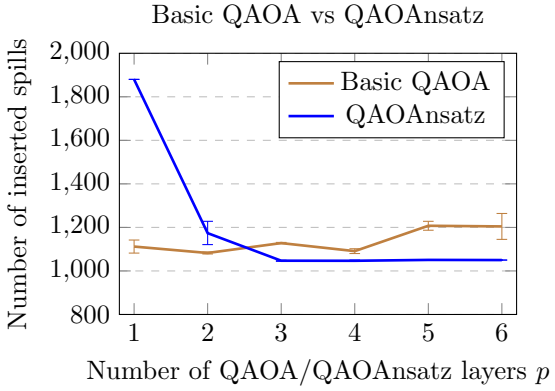
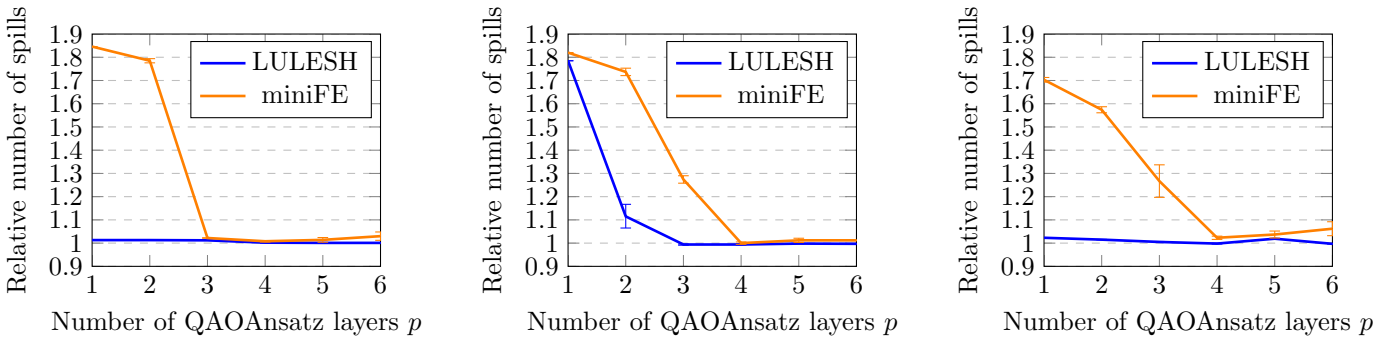


Figure 5: Comparison between the performance of a basic QAOA solver and our QAOAnsatz-based solver for the PBQP Register Allocation problem. The compiled code is LULESH, with $N_{\text{qubits}} = 16$ and $N_{\text{regs}} = 3$, the number of spills is averaged over 3 compilations.

We compile the LULESH mini-app with $N_{\text{qubits}} = 16$ and $N_{\text{regs}} = 3$ for an increasing number of circuit layers p for each algorithm. We use the COBYLA optimizer [25] for the variational algorithms. The results of these experiments are shown in Figure 5. The QAOAnsatz-based method struggles here for very shallow ($p \leq 2$) circuits compared to the basic QAOA implementation but as more layers are added, it overtakes the basic one. The QAOAnsatz behaves as expected: increasing the number of circuit layers improves the performance. We do not observe this behavior for the basic QAOA implementation, possibly due to the soft problem constraints leading to a large number of measured "illegal" states that saturate the results, thus hiding the "good" solutions regardless of circuit depth.

We evaluated the relative number of inserted spills compared to the classical allocation for the LULESH [**?**] and miniFE [24]



(a) $N_{\text{qubits}} = 12, N_{\text{regs}} = 3$ (b) $N_{\text{qubits}} = 16, N_{\text{regs}} = 3$ (c) $N_{\text{qubits}} = 20, N_{\text{regs}} = 3$

Figure 6: Performance of the QAOAnsatz PBQP Register Allocation method for various N_{qubits}, p and compiled classical codes. The performance is measured in the number of spills inserted in the code relative to the number of spills found with the fully classical allocator, as given by LLVM. The number of spills is averaged over 3 compilations

codes. All allocation, quantum and classical, were obtained using the same N_{regs} constraint and for varying numbers of available qubits. The results of those experiments are displayed in Figure 6. In all cases, the number of inserted spills trends down when the number of circuit layers p increases. For $N_{\text{qubits}} \in \{12, 20\}$ on the LULESH code, the decrease is slower. The allocation problem characteristics at those sizes are possibly leading to a faster convergence.

Finally, a question that arises naturally is whether or not the number of inserted spills is a good metric for the final performance of the compiled program in our case. We verify this by measuring the execution time of the compiled programs using both quantum and classical allocators with the same N_{regs} constraints. The results of this analysis are displayed in Figure 7 for the execution of the compiled LULESH codes.

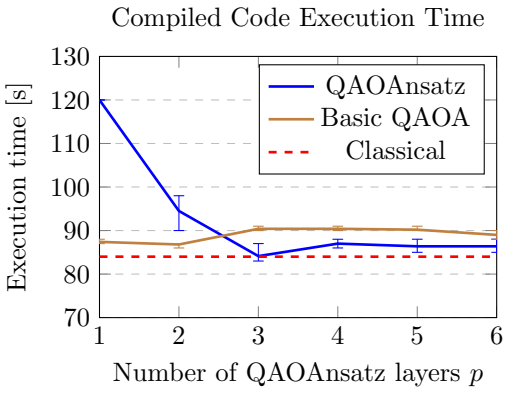


Figure 7: Execution time of the compiled LULESH code for varying numbers of QAOA layers, averaged on 5 sequential runs with problem size 40. Compiled with $N_{\text{qubit}} = 16$ and $N_{\text{regs}} = 3$.

We can conclude the following from these results:

- The evolution of the compiled code's execution time roughly follows the one we observed for inserted spills in Figure 5, validating the number of inserted spills as a metric for the quality of the compiled code.
- While in most cases the classical solver outperforms our new method, there are instances such as with $p = 3$ where it sometimes has a slight advantage. Further analysis of the influence of all hyperparameters and problem characteristics is a path toward further gains over the classical methods.

5 Conclusion and Future Work

In this work, we presented an end-to-end integration of a quantum method to solve the Register Allocation problem in the classical LLVM compiler [9]. This was achieved by integrating a quantum subroutine into the existing classical solver for the PBQP formulation of the problem [13]. The hybrid algorithm allows deciding the size of the problem that is sent as input to the quantum part of the algorithm, which is based on the Quantum Alternating Operator Ansatz (QAOAnsatz) [20] framework, a generalization of the well-known QAOA [3]. We used a mixer Hamiltonian for the algorithm that fits the constraints of the PBQP formulation. We implemented our hybrid method using the NVIDIA CUDA-Q [10] framework in C++ directly in the LLVM compiler allowing a tight integration in the existing code which did not rely on calling external Python routines.

We evaluated the performance of our hybrid method by compiling multiple classical codes and measuring the quality of the obtained compiled applications. This quality was measured through metrics such as the number of inserted spills and the execution time of the compiled application. Our QAOAnsatz-based method displayed expected behavior such as the quality of the resulting application increasing as we added "QAOAnsatz layers" to our quantum circuits. The quality of the compiled applications outperformed the one obtained by using LLVM's classical PBQP allocator in some specific cases and obtained similar results most of the time.

We believe this work constitutes a new step forward on the path to creating an integrated software stack and toolchain for hybrid quantum-classical systems. This builds on our previous preliminary work on GCC's register allocator [8] and presents a fully integrated quantum method in an existing large C++ application that is a very important part of many toolchains. Further work following this goal of an integrated HPC/QC software stack will include improvements to the presented method; extension to larger, state-of-the-art Register Allocators such as LLVM's Greedy allocator; and exploration of analog quantum

methods for this problem.

Acknowledgments

This work is part of HQI initiative (www.hqi.fr) and is supported by France 2030 under the French National Research Agency award number "ANR-22-PNCQ-0002".

References

[1] P. Shor, "Algorithms for Quantum Computation: Discrete Logarithms and Factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, Nov. 1994, pp. 124–134.

[2] A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum Algorithm for Solving Linear Systems of Equations," *Physical Review Letters*, vol. 103, no. 15, p. 150502, Oct. 2009.

[3] E. Farhi, J. Goldstone, and S. Gutmann, "A Quantum Approximate Optimization Algorithm," Nov. 2014.

[4] M. A. Nielsen and I. L. Chuang, "Quantum Computation and Quantum Information: 10th Anniversary Edition," Dec. 2010.

[5] P. Seitz, A. Elsharkawy, X.-T. M. To, and M. Schulz, "Toward a Unified Hybrid HPCQC Toolchain," in *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 02, Sep. 2023, pp. 96–102.

[6] M. O. Beg, "Combinatorial Problems in Compiler Optimization," Doctoral Thesis, University of Waterloo, Apr. 2013.

[7] G. J. Chaitin, "Register Allocation & Spilling via Graph Coloring," *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 98–101, Jun. 1982.

[8] B. Chichereau, S. Vialle, and P. Carribault, "Experimenting with Hybrid Quantum Optimization in HPC Software Stack for CPU Register Allocation," in *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 02, Sep. 2023, pp. 134–140.

[9] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, Mar. 2004, pp. 75–86.

[10] J.-S. Kim, A. McCaskey, B. Heim, M. Modani, S. Stanwyck, and T. Costa, "CUDA Quantum: The Platform for Integrated Quantum-Classical Computing," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, Jul. 2023, pp. 1–4.

[11] "GCC, the GNU Compiler Collection - GNU Project."

[12] B. Scholz and E. Eckstein, "Register Allocation for Irregular Architectures," in *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, ser. LCTES/SCOPES '02. New York, NY, USA: Association for Computing Machinery, Jun. 2002, pp. 139–148.

[13] L. Hames and B. Scholz, "Nearly Optimal Register Allocation with PBQP," in *Modular Programming Languages*, ser. Lecture Notes in Computer Science, D. E. Lightfoot and C. Szyperski, Eds. Berlin, Heidelberg: Springer, 2006, pp. 346–361.

[14] T. Kadowaki and H. Nishimori, "Quantum Annealing in the Transverse Ising Model," *Physical Review E*, vol. 58, no. 5, pp. 5355–5363, Nov. 1998.

[15] D. Bulger, W. P. Baritompa, and G. R. Wood, "Implementing Pure Adaptive Search with Grover's Quantum Algorithm," *Journal of Optimization Theory and Applications*, vol. 116, no. 3, pp. 517–529, Mar. 2003.

[16] M. Cerezo, A. Arrasmith, R. Babbush, S. C. Benjamin, S. Endo, K. Fujii, J. R. McClean, K. Mitarai, X. Yuan, L. Cincio, and P. J. Coles, "Variational Quantum Algorithms," *Nature Reviews Physics*, vol. 3, no. 9, pp. 625–644, Sep. 2021.

[17] K. Blekos, D. Brand, A. Ceschini, C.-H. Chou, R.-H. Li, K. Pandya, and A. Summer, "A Review on Quantum Approximate Optimization Algorithm and Its Variants," *Physics Reports*, vol. 1068, pp. 1–66, Jun. 2024.

[18] R. Herrman, P. C. Lotshaw, J. Ostrowski, T. S. Humble, and G. Siopsis, "Multi-Angle Quantum Approximate Optimization Algorithm," *Scientific Reports*, vol. 12, no. 1, p. 6781, Apr. 2022.

[19] L. Zhu, H. L. Tang, G. S. Barron, F. A. Calderon-Vargas, N. J. Mayhall, E. Barnes, and S. E. Economou, "Adaptive Quantum Approximate Optimization Algorithm for Solving Combinatorial Problems on a Quantum Computer," *Physical Review Research*, vol. 4, no. 3, p. 033029, Jul. 2022.

[20] S. Hadfield, Z. Wang, B. O'Gorman, E. G. Rieffel, D. Venturelli, and R. Biswas, "From the Quantum Approximate Optimization Algorithm to a Quantum Alternating Operator Ansatz," *Algorithms*, vol. 12, no. 2, p. 34, Feb. 2019.

[21] S.-S. Wang, H.-L. Liu, Y.-Q. Song, F. Gao, S.-J. Qin, and Q.-Y. Wen, "Quantum Alternating Operator Ansatz for Solving the Minimum Exact Cover Problem," *Physica A: Statistical Mechanics and its Applications*, vol. 626, p. 129089, Sep. 2023.

[22] A. Bärtschi and S. Eidenbenz, "Grover Mixers for QAOA: Shifting Complexity from Mixer Design to State Preparation," in *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Oct. 2020, pp. 72–82.

[23] X.-C. Wu, S. P. Premaratne, and K. Rasch, "A Comprehensive Introduction to the Intel Quantum SDK," in *Proceedings of the 2023 Introduction on Hybrid Quantum-Classical Programming Using C++ Quantum Extension*, ser. HybridQC '23. New York, NY, USA: Association for Computing Machinery, Aug. 2023, pp. 1–2.

[24] M. Heroux, "miniFE Finite Element Mini-Application," Jun. 2022.

[25] M. J. D. Powell, "A Direct Search Optimization Method That Models the Objective and Constraint Functions by Linear Interpolation," in *Advances in Optimization and Numerical Analysis*, ser. Mathematics and Its Applications, S. Gomez and J.-P. Hennart, Eds. Dordrecht: Springer Netherlands, 1994, pp. 51–67.